Chapter 12

Instruction Scheduling

CHAPTER OVERVIEW

The execution time of a set of operations depends heavily on the order in which they are presented for execution. Instruction scheduling attempts to reorder the operations in a procedure to improve its running time. In essence, it tries to execute as many operations per cycle as possible.

This chapter introduces the dominant technique for scheduling in compilers: greedy list scheduling. It then presents several methods for applying list scheduling to larger scopes than a single basic block.

Keywords: Instruction Scheduling, List Scheduling, Trace Scheduling, Software Pipelining

12.1 INTRODUCTION

On many processors, the order in which operations are presented for execution has a significant effect on the length of time it takes to execute a sequence of instructions. Different operations take different lengths of time. On a typical commodity microprocessor, integer addition and subtraction require less time than integer division; similarly, floating-point division takes longer than floating-point addition or subtraction. Multiplication usually falls between the corresponding addition and division operations. The time required to complete a load from memory depends on where in the memory hierarchy the value resides at the time that the load is issued.

The task of ordering the operations in a block or a procedure to make effective use of processor resources is called *instruction scheduling*. The scheduler takes as input a partially ordered list of operations in the target machine's assembly language; it produces as output an ordered version

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00012-8

of the same list. The scheduler assumes that the code has already been optimized and it does not try to duplicate the optimizer's work. Instead, it packs operations into the available cycles and functional unit issue slots so that the code will run as quickly as possible.

Conceptual Roadmap

The order in which the processor encounters operations has a direct impact on the speed of execution of compiled code. Thus, most compilers include an instruction scheduler that reorders the final operations to improve performance. The scheduler's choices are constrained by the flow of data, by the delays associated with individual operations, and by the capabilities of the target processor. The scheduler must account for all these factors if it is to produce a correct and efficient schedule for the compiled code.

The dominant technique for instruction scheduling is a greedy heuristic called list scheduling. List schedulers operate on straightline code and use a variety of priority ranking schemes to guide their choices. Compiler writers have invented a number of frameworks to schedule over larger regions in the code than basic blocks; these regional and loop schedulers simply create conditions where the compiler can apply list scheduling to a longer sequence of operations.

Overview

On most modern processors, the order in which instructions appear has an impact on the speed with which the code executes. Processors overlap the execution of operations, issuing successive operations as quickly as possible given the finite (and small) set of functional units. In principle this strategy makes good utilization of hardware resources and decreases execution time by overlapping the execution of successive operations. The difficulty arises when an operation issues before its operands are ready.

Stall
the delay caused by a hardware *interlock* that prevents a value from being read until its defining operation completes

An *interlock* is the mechanism that detects the premature issue and creates the actual delay.

Statically scheduled
A processor that relies on compiler insertion of `NOP`s for correctness is a *statically scheduled* processor.

Dynamically scheduled
A processor that provides interlocks to ensure correctness is a *dynamically scheduled* processor.

Processor designs handle this situation in one of two ways. The processor can *stall* the premature operation until its operands are available. On a machine that stalls premature operations, the scheduler reorders the operations in an attempt to minimize the number of such stalls. Alternatively, the processor can execute the premature operation, albeit with the incorrect operands. This approach relies on the scheduler to maintain enough distance between a value's definition and its various uses to maintain correctness. If insufficient useful operations are available to cover the delay associated with some operation, the scheduler must insert `nop`s to fill the gap.

Commodity microprocessors often have operations that have different latencies. Typical values might be one cycle for an integer add or subtract, three for an integer multiply or a floating-point add or subtract, five for a floating-point multiply, 12 to 18 for a floating-point divide, and 20 to 40 for an integer divide. As a further complication, some operations have variable latencies. The latency of a `load` depends on where in the memory hierarchy it finds the value; those latencies can range from a few cycles, say one to five for the nearest cache, to tens or hundreds of cycles for values in main memory. Arithmetic operations can have variable latencies as well. For example, floating-point multiply and divide units may take an early exit when they recognize that the actual operands render some stages of processing irrelevant (e.g. multiply by zero or one).

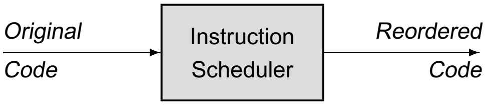
To further complicate matters, many commodity processors have the property that they can initiate execution of more than one operation in each cycle. So-called *superscalar* processors exploit parallelism at the instruction level—independent operations that can run concurrently without conflict. In a superscalar environment, the scheduler's job is to keep as many functional units busy as possible. Because the instruction dispatch hardware has a limited amount of lookahead, the scheduler may need to pay attention to both the cycle in which each operation issues and the relative ordering of operations within each cycle.

Superscalar
A processor that can issue distinct operations to multiple distinct functional units in a single cycle is considered a *superscalar* processor.

Instruction level parallelism (ILP)
the availability of independent operations that can execute concurrently

Consider, for example, a simple processor with one integer functional unit and one floating-point functional unit. The compiler wants to schedule a loop that consists of 100 integer operations and 100 floating-point operations. If the compiler orders the operations so that the first 75 operations are integer operations, the floating-point unit will sit idle until the processor finally reaches some work for it. If all the operations are independent (an unrealistic assumption), the best order might be to alternate operations between the two units.

Informally, instruction scheduling is the process whereby a compiler reorders the operations in the compiled code in an attempt to decrease its running time. Conceptually, an instruction scheduler looks like:



The instruction scheduler takes as input a partially ordered list of instructions; it produces as output an ordered list of instructions constructed from the same set of operations. The scheduler assumes a fixed set of operations; it does not rewrite the code (other than adding `nop`s to maintain correct execution). The scheduler assumes a fixed allocation of values to registers; while it may rename registers, it does not change allocation decisions.

MEASURING RUNTIME PERFORMANCE

The primary goal of instruction scheduling is to improve the running time of the generated code. Discussions of performance use many different metrics; the two most common are

Instructions per second The metric commonly used to advertise computers and to compare system performance is the number of instructions executed in a second. This can be measured as instructions issued per second or instructions retired per second.

Time to complete a fixed task This metric uses one or more programs whose behavior is known and compares the time required to complete these fixed tasks. This approach, called *benchmarking*, provides information about overall system performance, both hardware and software, on a particular workload.

No single metric contains enough information to allow evaluation of the quality of code generated by the compiler's back end. For example, if the measure is instructions per second, does the compiler get extra credit for leaving extraneous (but independent) instructions in code? The simple timing metric provides no information about what is achievable for a given program. Thus, it allows one compiler to do better than another but fails to show the distance between the generated code and what is optimal for that code on the target machine.

Numbers that the compiler writer might want to measure include the percentage of executed instructions whose results are actually used and the percentage of cycles spent in stalls and interlocks. The former gives insight into some aspects of predicated execution, while the latter directly measures some aspects of schedule quality.

The instruction scheduler has three primary goals. First, it must preserve the meaning of the code that it receives as input. Second, it should minimize execution time by avoiding stalls or `nop`s. Third, it should avoid increasing value lifetimes past the point where additional register spills are necessary. Of course, the scheduler should operate efficiently.

Many processors can issue multiple operations per cycle. While the mechanisms vary across architectures, the underlying challenge for the scheduler is the same: make good utilization of the hardware resources. In a very long instruction word (VLIW) processor, the processor issues an operation for each functional unit in each cycle, all gathered into a single fixed-format instruction. (The scheduler packs `nop`s into the slots for idle functional units.) A packed VLIW machine avoids many of these `nop`s with a variable-length instruction.

Superscalar processors look over a small window in the instruction stream, pick out operations that can execute on available units, and assign them to functional units. A dynamically scheduled processor considers operand availability; a statically scheduled processor only considers functional unit availability. An out-of-order superscalar processor uses a much larger window to scan for operations to execute; the window might be a hundred or more instructions.

This diversity of hardware dispatch mechanisms blurs the distinction between an operation and an instruction. On VLIW and packed VLIW machines, an instruction contains multiple operations. On superscalar machines, we usually refer to a single operation as an instruction and describe these machines as issuing multiple instructions per cycle. Throughout this book, we have used the term *operation* to describe a single opcode and its operands. We use the term *instruction* only to refer to an aggregation of one or more operations that all issue in the same cycle.

In deference to tradition, we still refer to this problem as *instruction scheduling*, although it might be more precisely called *operation scheduling*. On a VLIW or packed VLIW architecture, the scheduler packs operations into instructions that execute in a given cycle. On a superscalar architecture, either in order or out of order, the scheduler reorders operations to let the processor issue as many as possible in each cycle.

This chapter examines scheduling and the tools and techniques that compilers use to perform it. Section 12.2 provides a detailed introduction to the problem. Section 12.3 introduces the standard framework used for instruction scheduling: the list-scheduling algorithm. Section 12.4 presents several techniques that compilers use to extend the range of operations over which they can apply list scheduling. The "Advanced Topics" section presents an approach to loop scheduling.

12.2 THE INSTRUCTION-SCHEDULING PROBLEM

Consider the small example code shown in Figure 12.1; it reproduces an example used in Section 1.3. The column labelled "Start" shows the cycle in which each operation begins execution. Assume that the processor has a single functional unit, loads and stores take three cycles, a multiply takes two cycles, and all other operations complete in a single cycle. With these assumptions, the original code, shown on the left, takes 22 cycles.

The scheduled code, in Figure 12.1b, executes in many fewer cycles. It separates long-latency operations from operations that reference their results. This separation allows operations that do not depend on these results to

Start	Operations
1	loadAI r_{arp}, @a ⇒ r_1
4	add r_1, r_1 ⇒ r_1
5	loadAI r_{arp}, @b ⇒ r_2
8	mult r_1, r_2 ⇒ r_1
10	loadAI r_{arp}, @c ⇒ r_2
13	mult r_1, r_2 ⇒ r_1
15	loadAI r_{arp}, @d ⇒ r_2
18	mult r_1, r_2 ⇒ r_1
20	storeAI r_1 ⇒ r_{arp}, @a

(a) Original Code

Start	Operations
1	loadAI r_{arp}, @a ⇒ r_1
2	loadAI r_{arp}, @b ⇒ r_2
3	loadAI r_{arp}, @c ⇒ r_3
4	add r_1, r_1 ⇒ r_1
5	mult r_1, r_2 ⇒ r_1
6	loadAI r_{arp}, @d ⇒ r_2
7	mult r_1, r_3 ⇒ r_1
9	mult r_1, r_2 ⇒ r_1
11	storeAI r_1 ⇒ r_{arp}, @a

(b) Scheduled Code

■ **FIGURE 12.1** Example Block from Chapter 1.

execute concurrently with the long-latency operations. The code issues load operations in the first three cycles; the results are available in cycles 4, 5, and 6, respectively. This schedule requires an extra register, r_3, to hold the result of the third concurrently executing load operation, but it allows the processor to perform useful work while waiting for the first arithmetic operand to arrive. The overlap among operations effectively hides the latency of the memory operations. The same idea, applied throughout the block, hides the latency of the mult operation. The reordering reduces the running time to 13 cycles, a 41 percent improvement.

All of the examples we have seen so far deal, implicitly, with a target machine that issues a single operation in each cycle. Almost all commodity processors have multiple functional units and issue several operations in each cycle. We will introduce the list-scheduling algorithm for a single-issue machine and point out how to extend the basic algorithm to handle multioperation instructions.

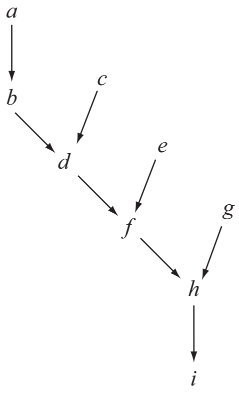
Dependence graph
For a block b, its dependence graph $\mathcal{D} = (N, E)$ has a node for each operation in b. An edge in $\mathcal{D}$ connects two nodes n_1 and n_2 if n_2 uses the result of n_1.

The instruction scheduling problem is defined over the *dependence graph* $\mathcal{D}$ of a basic block. $\mathcal{D}$ is sometimes called a *precedence graph*. Edges in $\mathcal{D}$ represent the flow of values in the block. Additionally, each node has two attributes, an *operation type* and a *delay*. For a node n, the operation corresponding to n must execute on a functional unit specified by its operation type; it requires *delay*(n) cycles to complete. Figure 12.2b shows the dependence graph for the code in our running example. We have substituted concrete numbers for @a, @b, @c, and @d to avoid confusion with the labels used to identify operations.

Nodes with no predecessors in $\mathcal{D}$, such as a, c, e, and g in the example, are called *leaves* of the graph. Since the leaves depend on no other operations,

```
a:  loadAI   rarp,@a ⇒ r1
b:  add      r1,r1   ⇒ r1
c:  loadAI   rarp,@b ⇒ r2
d:  mult     r1,r2   ⇒ r1
e:  loadAI   rarp,@c ⇒ r3
f:  mult     r1,r2   ⇒ r1
g:  loadAI   rarp,@d ⇒ r2
h:  mult     r1,r2   ⇒ r1
i:  storeAI  r1      ⇒ rarp,@a
```

(a) Example Code



(b) Its Dependence Graph

■ **FIGURE 12.2** Dependence Graph for the Example.

they can be scheduled as early as possible. Nodes with no successors in $\mathcal{D}$, such as i in the example, are called *roots* of the graph. The roots are, in some sense, the most constrained nodes in the graph because they cannot execute until all of their ancestors have executed. With this terminology, it appears that we have drawn $\mathcal{D}$ upside down—at least with regard to the trees, ASTs, and DAGs used earlier in the book. Placing the leaves at the top of the figure, however, creates a rough correspondence between placement in the drawing and eventual placement in the scheduled code. A leaf is at the top of the tree because it can execute early in the schedule. A root is at the bottom of the tree because it must execute after each of its ancestors.

$\mathcal{D}$ is not a tree. It is a forest of DAGs. Thus, nodes can have multiple parents and $\mathcal{D}$ can have multiple roots.

Given a dependence graph $\mathcal{D}$ for a code fragment, a schedule S maps each node $n \in N$ to a nonnegative integer that denotes the cycle in which it should be issued, assuming that the first operation issues in cycle 1. This provides a clear and concise definition of an instruction, namely, the i^{th} instruction is the set of operations $\{n \mid S(n) = i\}$. A schedule must meet three constraints.

1. $S(n) \geq 1$, for each $n \in N$. This constraint forbids operations that issue before execution starts. A schedule that violates this constraint is not well formed. For the sake of uniformity, the schedule must also have at least one operation n' with $S(n') = 1$.
2. If $(n_1, n_2) \in E$ then $S(n_1) + delay(n_1) \leq S(n_2)$. This constraint enforces correctness. An operation cannot issue until its operands have been defined. A schedule that violates this rule changes the flow of data in the code and is likely to produce incorrect results on a statically-scheduled machine.

3. Each instruction contains no more operations of each type t than the target machine can issue in a cycle. This constraint enforces feasibility, since a schedule that violates it contains instructions that the target machine cannot possibly issue. (On a typical VLIW machine, the scheduler must fill unused slots in an instruction with `nops`.)

The compiler should only produce schedules that meet all three constraints.

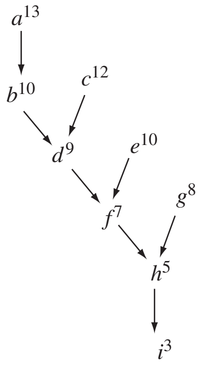
Given a well-formed schedule that is both correct and feasible, the length of the schedule is simply the cycle number in which the last operation completes, assuming the first instruction issues in cycle 1. Schedule length can be computed as:

$$L(S) = \max_{n \in N} (S(n) + delay(n)).$$

If we assume that *delay* captures all the operational latencies, schedule S should execute in $L(S)$ time. With a notion of schedule length comes the notion of a *time-optimal* schedule. A schedule S_i is time optimal if $L(S_i) \leq L(S_j)$ for all other schedules S_j that contain the same set of operations.

Critical path

the longest latency path through a dependence graph



Dependence Graph
Annotated with Latencies

The dependence graph captures important properties of the schedule. Computing the total delay along the paths through the graph exposes additional detail about the block. Annotating the dependence graph $\mathcal{D}$ for our example with information about cumulative latency yields the graph shown in the margin. The path length from a node to the end of the computation is shown as a superscript on the node. The values clearly show that the path *abdfhi* is longest—it is the *critical path* that determines overall execution time for this example.

How, then, should the compiler schedule this computation? An operation can only be scheduled into an instruction when its operands are available. Since a, c, e, and g have no predecessors in the graph, they are the initial candidates for scheduling. The fact that a lies on the critical path strongly suggests that it be scheduled into the first instruction. Once a has been scheduled, the longest path remaining in $\mathcal{D}$ is *cdefhi*, suggesting that c be scheduled as the second instruction. With the schedule *ac*, b and e tie for the longest path. However, b needs the result of a, which will not be available until the fourth cycle. This makes e followed by b the better choice. Continuing in this fashion leads to the schedule *acebdgfhi*. This matches the schedule shown in Figure 12.1b.

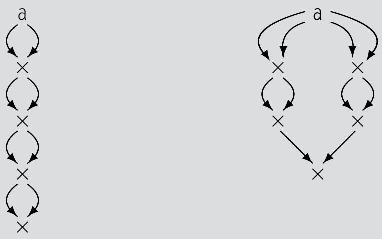
However, the compiler cannot simply rearrange the instructions into the proposed order. Recall that both c and e define `r2` and d uses the value that c stores in `r2`. The scheduler cannot move e before d unless it renames the

LIMITATIONS TO SCHEDULING

The scheduler cannot cure all problems with instruction order. Consider the following code to compute a^{16}

Start	Operations		
1	loadAI	r_{arp},@a	⇒ r_1
4	mult	r_1,r_1	⇒ r_1
6	mult	r_1,r_1	⇒ r_1
8	mult	r_1,r_1	⇒ r_1
10	mult	r_1,r_1	⇒ r_1
12	storeAI	r_1	⇒ r_{arp},@x

The mult operations each need two cycles. The chain of dependences, shown on the left below, between the multiplies prevents the scheduler from improving the code. (If other independent operations are available, the scheduler could place them between the multiplies.)



Original Code Refactored Code

The issue is one of code shape that must be addressed earlier in compilation. If the optimizer refactors or reshapes the code into $(a^2)^2 \cdot (a^2)^2$, as shown on the right, the scheduler can overlap some of the multiplications and achieve a shorter schedule. If the processor can only issue one multiply per cycle, the refactored schedule saves one cycle. If the processor can issue two multiplies per cycle, it saves two cycles.

result of e to avoid the conflict with c's definition of r_2. This constraint arises not from the flow of data, as with the dependences modelled by edges in $\mathcal{D}$. Instead, it prevents an assignment that would change the flow of data. These constraints are often called *antidependences*. We denote the antidependence between e and d as $e \rightarrow d$.

Antidependence

Operation *x* is *antidependent* on operation *y* if *x* precedes *y* and *y* defines a value used in *x*. Reversing their order of execution could cause *x* to compute a different value.

The scheduler can produce correct code in at least two different ways. It can discover the antidependences that are present in the input code and respect them in the final schedule, or it can rename values to avoid them. The example contains four antidependences, namely, $e \rightarrow c$, $e \rightarrow d$, $g \rightarrow e$, and $g \rightarrow f$. All of them involve redefinition of r_2. (Constraints exist based on r_1 as well, but each antidependence on r_1 duplicates a dependence based on the flow of values.)

Respecting antidependences changes the set of schedules that the compiler can produce. For example, it cannot move *e* before *c* or *d*. This forces it to produce a schedule such as *acbdefghi*, which requires 18 cycles. While this schedule is an 18 percent improvement over the unscheduled code (*abcdefghi*), it is not competitive with the 41 percent improvement obtained by renaming to produce *acebdgfhi*, as shown on the right side of Figure 12.1.

As an alternative, the scheduler can systematically rename the values in the block to eliminate antidependences before it schedules the code. This approach frees the scheduler from the constraints imposed by antidependences, but it creates the potential for problems if the scheduled code requires spill code. Renaming does not change the number of live variables; it simply changes their names and helps the scheduler avoid violating antidependences. Increasing overlap, however, can increase demand for registers and force the register allocator to spill more values—adding long-latency operations and forcing another round of scheduling.

The simplest renaming scheme assigns a new name to each value as it is produced. In the ongoing example, this scheme produces the following code. This version of the code has the same pattern of definitions and uses.

```
a:  loadAI   rarp,@a  ⇒ r1
b:  add      r1,r1    ⇒ r2
c:  loadAI   rarp,@b  ⇒ r3
d:  mult     r2,r3    ⇒ r4
e:  loadAI   rarp,@c  ⇒ r5
f:  mult     r4,r5    ⇒ r6
g:  loadAI   rarp,@d  ⇒ r7
h:  mult     r6,r7    ⇒ r8
i:  storeAI  r8       ⇒ rarp,@a
```

However, the dependence relationships are expressed unambiguously in the code. It contains no antidependences, so naming constraints cannot arise.

12.2.1 Other Measures of Schedule Quality

Schedules can be measured in terms other than time. Two schedules S_i and S_j for the same block might produce different demands for registers—that is, the maximum number of live values in S_j may be less than in S_i. If the processor requires the scheduler to insert nops for idle functional units, then S_i might have fewer operations than S_j and might fetch fewer instructions as a result. This need not depend solely on schedule length. For example, on a

INTERACTIONS BETWEEN SCHEDULING AND ALLOCATION

Antidependences between operations can limit the scheduler's ability to reorder operations. The scheduler can avoid antidependences by renaming; however, renaming creates a need for the compiler to perform register allocation after scheduling. This example is but one of the interactions between instruction scheduling and register allocation.

The core function of the scheduler is to reorder operations. Since most operations both use and define values, changing the relative order of two operations *x* and *y* can change the lifetimes of values. Moving *y* from below *x* to above *x* lengthens the lifetime of the value *y* defines. If one of *x*'s operands is a last use, moving *x* below *y* lengthens its lifetime. Symmetrically, if one of *y*'s operands is a last use, moving *y* above *x* shortens its lifetime.

The net effect of reordering *x* and *y* depends on the details of both *x* and *y*, as well as the surrounding code. If none of the uses involved is a last use, then the swap has no net effect on demand for registers. (Each operation defines a register; swapping them changes the lifetimes of specific registers, but not the aggregate demand for registers.)

In a similar way, register allocation can change the instruction-scheduling problem. The core functions of a register allocator are to rename references and to insert memory operations when demand for registers is larger than the register set. Both these functions affect the ability of the scheduler to produce fast code. When the allocator maps a large virtual name space to the name space of target-machine registers, it can introduce antidependences that constrain the scheduler. Similarly, when the allocator inserts spill code, it adds operations to the code that must, themselves, be scheduled into instructions.

We know, mathematically, that solving these problems together might produce solutions that cannot be obtained by running the scheduler followed by the allocator or the allocator followed by the scheduler. However, both problems are complex enough that most real-world compilers treat them separately.

processor with a variable-cycle `nop`, bunching `nop`s together produces fewer operations and, potentially, fewer instructions. Finally, S_j might require less energy than S_i to execute on the target system because it never uses one of the functional units, it fetches fewer instructions, or it causes fewer bit transitions in the processor's fetch and decode logic.

12.2.2 What Makes Scheduling Hard?

The fundamental operation in scheduling is gathering operations together into groups based on the cycle in which those operations will begin execution. For each operation, the scheduler must choose a cycle. For each cycle, the scheduler must choose a set of operations. In balancing these two viewpoints, it must ensure that each operation issues only when its operands are available.

When the scheduler places an operation *i* in cycle *c*, that decision affects the earliest possible placement of any operation that relies on the result of *i*—any operation in $\mathcal{D}$ that is reachable from *i*. If more than one operation can legally execute in cycle *c*, then the scheduler's choice can change the earliest placement of many operations—all those operations dependent (either directly or transitively) on each of the possible choices.

Local instruction scheduling is NP-complete for all but the simplest architectures. In practice, compilers produce approximate solutions to scheduling problems using greedy heuristics. Almost all the scheduling algorithms used in compilers are based on a single family of heuristic techniques, called *list scheduling*. The following section describes list scheduling in detail. Subsequent sections show how to extend the paradigm to larger scopes.

SECTION REVIEW

A local instruction scheduler must assign an execution cycle to each operation. (These cycles are numbered from the entry to the basic block.) In the process, it must ensure that no cycle in the schedule has more operations than the hardware can issue in a single cycle. On a statically scheduled processor, it must ensure that each operation issues only after its operands are available; that may require it to insert `nop`s into the schedule. On a dynamically scheduled processor, it should minimize the expected number of stalls that execution will cause.

The key data structure for instruction scheduling is the dependence graph for the block being processed. It represents the flow of data in the block. It is easily annotated with information about operation-by-operation delays. The annotated dependence graph exposes important information about constraints and critical paths in the block.

Review Questions

1. What parameters of the target processor might the scheduler need? Find these parameters for the processor in your own computer.
2. It is well known and widely appreciated that instruction scheduling interacts with register allocation. How does instruction scheduling interact with instruction selection? Are there modifications to the instruction selection process that we might make to simplify scheduling?

12.3 LOCAL LIST SCHEDULING

List scheduling is a greedy, heuristic approach to scheduling the operations in a basic block. It has been the dominant paradigm for instruction scheduling since the late 1970s, largely because it discovers reasonable schedules and it adapts easily to changes in computer architectures. However, list scheduling is an approach rather than a specific algorithm. Wide variation exists in how it is implemented and how it attempts to prioritize instructions for scheduling. This section explores the basic framework of list scheduling, as well as a couple of variations on the idea.

12.3.1 The Algorithm

Classic list scheduling operates on a single basic block. Limiting our consideration to straightline sequences of code allows us to ignore situations that can complicate scheduling. For example, when the scheduler considers multiple blocks, an operand might depend on prior definitions in different blocks, which creates uncertainty about when the operand is available for use. Code motion across block boundaries creates another set of complications. It can move an operation onto a path where it did not previously exist; it can also remove an operation from a path where it is necessary. Restricting our consideration to the single-block case avoids these complications. Section 12.4 explores cross-block scheduling.

To apply list scheduling to a block, the scheduler follows a four-step plan.

1. *Rename to avoid antidependences* To reduce the set of constraints on the scheduler, the compiler renames values. Each definition receives a unique name. This step is not strictly necessary. However, it lets the scheduler find some schedules that the antidependences would have prevented and it simplifies the scheduler's implementation.
2. *Build a dependence graph, $\mathcal{D}$* To build the dependence graph, the scheduler walks the block from bottom to top. For each operation, it constructs a node to represent the newly created value. It adds edges from that node to each node that uses the value. Each edge is annotated with the latency of the current operation. (If the scheduler does not perform renaming, $\mathcal{D}$ must represent antidependences as well.)
3. *Assign priorities to each operation* The scheduler uses these priorities as a guide when it picks from the set of available operations at each step. Many priority schemes have been used in list schedulers. The scheduler may compute several different scores for each node, using one as the primary ordering and the others to break ties between equally ranked nodes. One classic priority scheme uses the length of the longest

```
Cycle ← 1
Ready ← leaves of $\mathcal{D}$
Active ← $\emptyset$

while (Ready $\cup$ Active $\neq \emptyset$)

   for each op $\in$ Active
       if S(op) + delay(op) < Cycle then
          remove op from Active
          for each successor s of op in $\mathcal{D}$
              if s is ready
                 then add s to Ready

   if Ready $\neq \emptyset$ then
       remove an op from Ready
       S(op) ← Cycle
       add op to Active

   Cycle ← Cycle + 1
```

■ **FIGURE 12.3** List-Scheduling Algorithm.

latency-weighted path from the node to a root of $\mathcal{D}$. Other priority schemes are described in Section 12.3.4.

4. *Iteratively select an operation and schedule it* To schedule operations, the algorithm starts in the block's first cycle and chooses as many operations as possible to issue in that cycle. It then increments its cycle counter, updates its notion of which operations are ready to execute, and schedules the next cycle. It repeats this process until each operation has been scheduled. Clever use of data structures makes this process efficient.

Renaming and building $\mathcal{D}$ are straightforward. Typical priority computations traverse the dependence graph $\mathcal{D}$ and compute some metric on it. The heart of the algorithm, and the key to understanding it, lies in the final step—the scheduling algorithm. Figure 12.3 shows the basic framework for this step, assuming that the target has a single functional unit.

The scheduling algorithm performs an abstract simulation of the block's execution. It ignores the details of values and operations to focus on the timing constraints imposed by edges in $\mathcal{D}$. To track time, it maintains a simulation clock, in the variable `Cycle`. It initializes `Cycle` to 1 and increments it as it proceeds through the block.

The algorithm uses two lists to track operations. The `Ready` list holds all the operations that can execute in the current cycle. If an operation is in `Ready`, all of its operands have been computed. Initially, `Ready` contains all the leaves of $\mathcal{D}$, since they do not depend on other operations in the block. The `Active` list holds all operations that were issued in an earlier cycle but have not yet finished. Each time the scheduler increments `Cycle`, it removes from `Active` any operation `op` that finishes before `Cycle`. It then checks each successor of `op` in $\mathcal{D}$ to determine if it can move onto the `Ready` list—that is, if all its operands are available.

The list-scheduling algorithm follows a simple discipline. At each time step, it accounts for any operations completed in the previous cycle, it schedules an operation for the current cycle, and it increments `Cycle`. The process terminates when the simulated clock indicates that every operation has completed. If all the times specified by *delay* are accurate and all operands of the leaves of $\mathcal{D}$ are available in the first cycle, this simulated running time should match the actual execution time. A simple postpass can rearrange the operations and insert `nop`s as needed.

The algorithm must respect one final constraint. Any block-ending or jump must be scheduled so that the program counter does not change before the block ends. So, if i is the block-ending branch, it cannot be scheduled earlier than cycle $L(S) + 1 - delay(i)$. Thus, a single-cycle branch must be scheduled in the last cycle of the block, and a two-cycle branch must be scheduled no earlier than the second to last cycle in the block.

The quality of the schedule produced by this algorithm depends primarily on the mechanism used to pick an operation from the `Ready` queue. Consider the simplest scenario, where the `Ready` list contains at most one item in each iteration. In this restricted case, the algorithm must generate an optimal schedule. Only one operation can execute in the first cycle. (There must be at least one leaf in $\mathcal{D}$, and our restriction ensures that there is exactly one.) At each subsequent cycle, the algorithm has no choices to make—either `Ready` contains an operation and the algorithm schedules it, or `Ready` is empty and the algorithm schedules nothing to issue in that cycle. The difficulty arises when, in some cycle, the `Ready` queue contains multiple operations.

When the algorithm must choose among several ready operations, that choice is critical. The algorithm should take the operation with the highest priority score. In the case of a tie, it should use one or more other criteria to break the tie (see Section 12.3.4). The metric suggested earlier, the longest latency-weighted distance to a root in $\mathcal{D}$, corresponds to always choosing the node on the critical path for the current cycle in the schedule being constructed. To the extent that the impact of a scheduling priority is predictable, this scheme should provide balanced pursuit of the longest paths.

```
for each load operation, $l$, in the block
    delay($l$) ← 1

for each operation $i$ in $\mathcal{D}$
    let $\mathcal{D}_{\mathrm{i}}$ be the nodes and edges in $\mathcal{D}$ independent of $i$
    for each connected component $C$ of $\mathcal{D}_{\mathrm{i}}$ do
        find the maximal number of loads, $N$, on any path through $C$
        for each load operation $l$ in $C$
            delay($l$) ← delay($l$) + delay($i$) / $N$
```

■ **FIGURE 12.4** Computing Delays for Load Operations.

12.3.2 Scheduling Operations with Variable Delays

Memory operations often have uncertain and variable delays. A load operation on a machine with multiple levels of cache memory might have an actual delay ranging from zero cycles to hundreds or thousands of cycles. If the scheduler assumes the worst-case delay, it risks idling the processor for long periods. If it assumes the best-case delay, it will stall the processor on a cache miss. In practice, the compiler can obtain good results by calculating an individual latency for each load based on the amount of instruction-level parallelism available to cover the load's latency. This approach, called *balanced scheduling*, schedules the load with regard to the code that surrounds it rather than the hardware on which it will execute. It distributes the locally available parallelism across loads in the block. This strategy mitigates the effect of a cache miss by scheduling as much extra delay as possible for each load. It will not slow down execution in the absence of cache misses.

Figure 12.4 shows the computation of delays for individual loads in a block. The algorithm initializes the delay for each load to one. Next, it considers each operation i in the dependence graph $\mathcal{D}$ for the block. It finds the computations in $\mathcal{D}$ that are independent of i, called $\mathcal{D}_{\mathrm{i}}$. Conceptually, this task is a reachability problem on $\mathcal{D}$. We can find $\mathcal{D}_{\mathrm{i}}$ by removing from $\mathcal{D}$ every node that is a transitive predecessor of i or a transitive successor of i, along with any edges associated with those nodes.

The algorithm then finds the connected components of $\mathcal{D}_{\mathrm{i}}$. For each component C, it finds the maximum number N of loads on any single path through C. N is the number of loads in C that can share operation i's delay, so the algorithm adds $delay(i)/N$ to the delay of each load in C. For a given load l, the operation sums the fractional share of each independent operation i's delay that can be used to cover the latency of *l*. Using this value as *delay*(*l*) produces a schedule that shares the slack time of independent operations evenly across all loads in the block.

12.3.3 Extending the Algorithm

The list-scheduling algorithm, as presented, makes several assumptions that may not hold true in practice. The algorithm assumes that only one operation can issue per cycle; most processors can issue multiple operations per cycle. To handle this situation, we must expand the `while` loop so that it looks for an operation for each functional unit in each cycle. The initial extension is straightforward—the compiler writer can add a loop that iterates over the functional units.

The complexity arises when some operations can execute on multiple functional units and others cannot. The compiler writer may need to choose an order for the functional units that schedules the more-constrained units first and the less-constrained units later. On a processor with a partitioned register set, the scheduler may need to place an operation in the partition where its operands reside or schedule it into a cycle where the inter-partition transfer apparatus is idle.

At block boundaries, the scheduler needs to account for the fact that some operands computed in predecessor blocks may not be available in the first cycle. If the compiler invokes the list scheduler on the blocks in reverse postorder on the CFG, then the compiler can ensure that the scheduler knows how many cycles into the block it must wait on operands entering the block along forward edges in the CFG. (This solution does not help with a loop-closing branch; see Section 12.5 for a discussion of loop scheduling.)

12.3.4 Tie Breaking in the List-Scheduling Algorithm

The complexity of instruction scheduling causes compiler writers to use relatively inexpensive heuristic techniques—variants of the list-scheduling algorithm—rather than solving the problem to optimality. In practice, list scheduling produces good results; it often builds optimal or near-optimal schedules. However, as with many greedy algorithms, its behavior is not robust—small changes in the input may make large differences in the solution.

The methodology used to break ties has a strong impact on the quality of schedules produced by list scheduling. When two or more items have the same rank, the scheduler should break the tie based on another priority ranking. A good scheduler might have two or three tie-breaking priority ranks for each operation; it applies them in some consistent order. In addition to the latency-weighted path length described earlier, the scheduler might use the following:

- A node's rank is the number of immediate successors it has in $\mathcal{D}$. This metric encourages the scheduler to pursue many distinct paths through the graph—closer to a breadth-first approach. It tends to keep more operations on the `Ready` queue.
- A node's rank is the total number of descendants it has in $\mathcal{D}$. This metric amplifies the effect of the previous ranking. Nodes that compute critical values for many other nodes are scheduled early.
- A node's rank is equal to its *delay*. This metric schedules long-latency operations as soon as possible. It pushes them early in the block when more operations remain that might be used to cover their latency.
- A node's rank is equal to the number of operands for which this operation is the last use. As a tie breaker, this metric moves last uses closer to their definitions, which may decrease demand for registers.

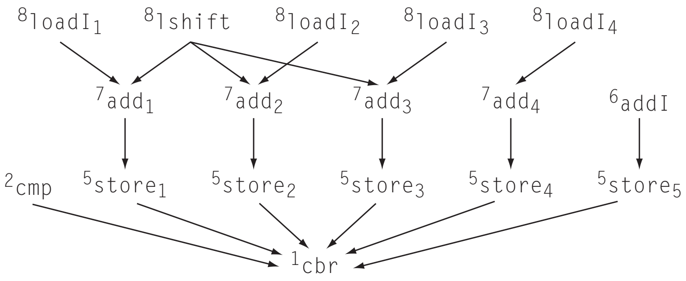
Unfortunately, none of these priority schemes dominates the others in terms of overall schedule quality. Each excels on some examples and does poorly on others. Thus, there is little agreement about which rankings to use or in which order to apply them.

12.3.5 Forward versus Backward List Scheduling

The list-scheduling algorithm, as presented in Figure 12.3, works over the dependence graph from its leaves to its roots and creates the schedule from the first cycle in the block to the last. An alternate formulation of the algorithm works over the dependence graph in the opposite direction, scheduling from roots to leaves. The first operation scheduled is the last to issue and the last operation scheduled is the first to issue. This version of the algorithm is called *backward list scheduling*, and the original version is called *forward list scheduling*.

List scheduling is not an expensive part of compilation. Thus, some compilers run the scheduler several times with different combinations of heuristics and keep the best schedule. (The scheduler can reuse most of the preparatory work—renaming, building the dependence graph, and computing some of the priorities.) In such a scheme, the compiler should consider using both forward and backward scheduling.

In practice, neither forward scheduling nor backward scheduling always wins. The difference between forward and backward list scheduling lies



Opcode	loadI	lshift	add	addI	cmp	store
Latency	1	1	2	1	1	4

■ **FIGURE 12.5** Dependence Graph for a Block from go.

in the order in which the scheduler considers operations. If the schedule depends critically on the careful ordering of some small set of operations, the two directions may produce noticeably different results. If the critical operations occur near the leaves, forward scheduling seems more likely to consider them together, while backward scheduling must work its way through the remainder of the block to reach them. Symmetrically, if the critical operations occur near the roots, backward scheduling may examine them together, while forward scheduling sees them in an order dictated by decisions made starting at the other end of the block.

To make this latter point more concrete, consider the example shown in Figure 12.5. It shows the dependence graph for a basic block found in the SPEC 95 benchmark program go. The compiler added dependences from the store operations to the block-ending branch to ensure that the memory operations complete before the next block begins execution. (Violating this assumption could produce an incorrect value from a subsequent load operation.) Superscripts on nodes in the dependence graph give the latency from the node to the end of the block; subscripts differentiate among similar operations. The example assumes operation latencies that appear in the table below the dependence graph.

This example demonstrates the difference between forward and backward list scheduling. It came to our attention in a study of list scheduling; the compiler was targeting an ILOC machine with two integer functional units and one unit to perform memory operations. The five store operations take most of the time in the block. The schedule that minimizes execution time must begin executing stores as early as possible.

Forward list scheduling, using latency to the end of the block for priority, executes the operations in priority order, except for the comparison. It schedules the five operations with rank eight, then the four rank seven operations and the rank six operation. It begins on the operations with rank five, and slides the cmp in alongside the stores, since the cmp is a leaf. If ties are broken arbitrarily by taking left-to-right order, this produces the schedule shown in Figure 12.6a. Notice that the memory operations begin in cycle 5, producing a schedule that issues the branch in cycle 13.

Using the same priorities with backward list scheduling, the compiler first places the branch in the last slot of the block. The cmp precedes it by one cycle, determined by *delay*(cmp). The next operation scheduled is store$_1$ (by the left-to-right tie-breaking rule). It is assigned the issue slot on the memory unit that is four cycles earlier, determined by *delay*(store). The scheduler fills in successively earlier slots on the memory unit with the other store operations, in order. It begins filling in the integer operations, as they become ready. The first is add$_1$, two cycles before store$_1$. When the algorithm terminates, it has produced the schedule shown in Figure 12.6b.

The backward schedule takes one fewer cycle than the forward schedule. It places the addI earlier in the block, allowing store$_5$ to issue in cycle 4—one cycle earlier than the first memory operation in the forward schedule. By considering the problem in a different order, using the same underlying priorities and tie breakers, the backward algorithm finds a different result.

	Integer	Integer	Memory
1	loadI$_1$	lshift	—
2	loadI$_2$	loadI$_3$	—
3	loadI$_4$	add$_1$	—
4	add$_2$	add$_3$	—
5	add$_4$	addI	store$_1$
6	cmp	—	store$_2$
7	—	—	store$_3$
8	—	—	store$_4$
9	—	—	store$_5$
10	—	—	—
11	—	—	—
12	—	—	—
13	cbr	—	—

(a) Forward Schedule

	Integer	Integer	Memory
1	loadI$_4$	—	—
2	addI	lshift	—
3	add$_4$	loadI$_3$	—
4	add$_3$	loadI$_2$	store$_5$
5	add$_2$	loadI$_1$	store$_4$
6	add$_1$	—	store$_3$
7	—	—	store$_2$
8	—	—	store$_1$
9	—	—	—
10	—	—	—
11	cmp	—	—
12	cbr	—	—

(b) Backward Schedule

■ **FIGURE 12.6** Schedules for the Block from go.

WHAT ABOUT OUT-OF-ORDER EXECUTION?

Some processors include hardware support for executing instructions out of order (OOO). We refer to such processors as *dynamically scheduled* machines. This feature is not new; for example, it appeared on the IBM 360/91. To support OOO execution, a dynamically scheduled processor looks ahead in the instruction stream for operations that can execute before they would in a statically scheduled processor. To do this, the dynamically scheduled processor builds and maintains a portion of the dependence graph at runtime. It uses this piece of the dependence graph to discover when each instruction can execute and issues each instruction at the first legal opportunity.

When can an out-of-order processor improve on the static schedule? If runtime circumstances are better than the assumptions made by the scheduler, then the OOO hardware might issue an operation earlier than its position in the static schedule. This can happen at a block boundary, if an operand is available before its worst-case time. It can happen with a variable-latency operation. Because it knows actual runtime addresses, an OOO processor can also disambiguate some load-store dependences that the scheduler cannot.

OOO execution does not eliminate the need for instruction scheduling. Because the lookahead window is finite, bad schedules can defy improvement. For example, a lookahead window of 50 instructions will not let the processor execute a string of 100 integer instructions followed by 100 floating-point instructions in interleaved ⟨integer, floating-point⟩ pairs. It may, however, interleave shorter strings, say of length 30. OOO execution helps the compiler by improving good, but nonoptimal, schedules.

A related processor feature is dynamic register renaming. This scheme provides the processor with more physical registers than the ISA allows the compiler to name. The processor can break antidependences that occur within its lookahead window by using additional physical registers that are hidden from the compiler to implement two references connected by an antidependence.

Why does this happen? The forward scheduler must place all the rank-eight operations in the schedule before any rank-seven operations. Even though the `addI` operation is a leaf, its lower rank causes the forward scheduler to defer it. By the time the scheduler runs out of rank-eight operations, other rank-seven operations are available. In contrast, the backward scheduler places the `addI` before three of the rank-eight operations—a result that the forward scheduler could not consider.

12.3.6 Improving the Efficiency of List Scheduling

To pick an operation from the `Ready` list, as described so far, requires a linear scan over `Ready`. This makes the cost of creating and maintaining `Ready` approach $\mathbf{O}(n^2)$. Replacing the list with a priority queue can reduce the cost of these manipulations to $\mathbf{O}(n \log_2 n)$, for a minor increase in the difficulty of implementation.

A similar approach can reduce the cost of manipulating the `Active` list. When the scheduler adds an operation to `Active`, it can assign it a priority equal to the cycle in which the operation completes. A priority queue that seeks the smallest priority will push all the operations completed in the current cycle to the front, for a small increase in cost over a simple list implementation.

Further improvement is possible in the implementation of `Active`. The scheduler can maintain a set of separate lists, one for each cycle in which an operation can finish. The number of lists required to cover all the operation latencies is $MaxLatency = \max_{n \in \mathcal{D}} \mathtt{delay}(n)$. When the compiler schedules operation n in `Cycle`, it adds n to $\mathtt{WorkList}[(\mathtt{Cycle} + \mathtt{delay}(n)) \bmod MaxLatency]$. When it goes to update the `Ready` queue, all of the operations with successors to consider are found in $\mathtt{WorkList}[\mathtt{Cycle\ mod}\ MaxLatency]$. This scheme uses a small amount of extra space; the sum of operations in the `WorkList`s is the same as in the `Active` list. The individual `WorkLists` will have a small amount of overhead space. It uses a little more time on each insertion into a `WorkList`, to calculate which `WorkList` it should use. In return, it avoids the quadratic cost of searching `Active` and replaces it with a linear walk through a smaller `WorkList`.

SECTION REVIEW

List scheduling has been the dominant paradigm that compilers have used to schedule operations for many years. It computes, for each operation, the cycle in which it should issue. The algorithm is reasonably efficient; its complexity relates directly to the underlying dependence graph. This greedy heuristic approach, in its forward and backward forms, produces excellent results for single blocks.

Algorithms that perform scheduling over larger regions in the CFG use list scheduling to order operations. Its strengths and weaknesses carry over to those other domains. Thus, any improvements made to local list scheduling have the potential to improve the regional scheduling algorithms, as well.

Review Questions

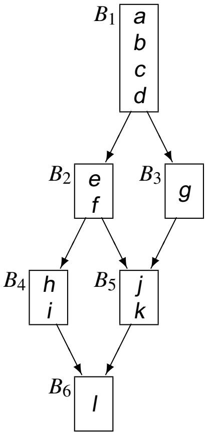
1. You are asked to implement a list scheduler for a compiler that will produce code for your laptop. What metric do you use as your primary ranking for the ready list and how do you break ties? Provide a rationale for your choices.
2. Different priority metrics cause the scheduler to consider the operations in different orders. Could you apply randomization to achieve similar effects?

12.4 REGIONAL SCHEDULING

As with value numbering, moving from single basic blocks to larger scopes can improve the quality of code that the compiler generates. For instruction scheduling, many different approaches have been proposed for regions larger than a block but smaller than a whole procedure. Almost all of those approaches use the basic list-scheduling algorithm as the engine for reordering instructions. They surround that algorithm with an infrastructure that lets it consider longer (e.g. multi-block) sequences of code. In this section, we will examine three ideas for improving schedule quality by changing the context in which the compiler applies list scheduling.

12.4.1 Scheduling Extended Basic Blocks

Recall from Section 8.3 that an extended basic block (EBB) consists of a set of blocks $B_1, B_2, \ldots, B_n$ in which B_1 has multiple predecessors and every other block B_i has exactly one predecessor, some B_j in the EBB. The compiler can identify EBBs in a simple pass over the CFG. Consider the simple code fragment shown in the margin. It has one large EBB, $\{B_1, B_2, B_3, B_4\}$, and two trivial EBBs, $\{B_5\}$ and $\{B_6\}$. The large EBB has two paths, $\langle B_1, B_2, B_4\rangle$, and $\langle B_1, B_3\rangle$, The paths share B_1 as a common prefix.



Example CFG

To obtain a larger context for list scheduling, the compiler can treat paths in an EBB, such $\langle B_1, B_2, B_4\rangle$, as if they are single blocks, provided it accounts for the shared path prefixes, such as B_1, which occurs in both $\langle B_1, B_2, B_4\rangle$ and $\langle B_1, B_3\rangle$, and the premature exits, such as $B_1 \rightarrow B_3$ and $B_2 \rightarrow B_5$. (We saw this same concept in the superlocal value numbering algorithm in Section 8.5.1.) This approach lets the compiler apply its highly effective scheduling engine—list scheduling—to longer sequences of operations. The effect is to increase the fraction of code that is scheduled together, which should improve execution times.

To see how shared prefixes and premature exits complicate list scheduling, consider the possibilities for code motion in the path $\langle B_1, B_2, B_4\rangle$ in the example from the margin. Such code motion may require that the scheduler insert *compensation code* to maintain correctness.

Compensation code
code inserted into a block B_i to counteract the effects of cross-block code motion along a path that does not include B_i

- The compiler can move an operation forward—that is, later on the path. For example, it might move operation *c* from B_1 into B_2. While that decision might speed execution along the path $\langle B_1, B_2, B_4\rangle$, it changes the computation performed along the path $\langle B_1, B_3\rangle$. Moving *c* forward out of B_1 means that the path $\langle B_1, B_3\rangle$ no longer executes *c*. Unless *c* is dead along all paths leading from B_3, the scheduler must correct this situation.
  To fix this problem, the scheduler must insert a copy of *c* into B_3. If it was legal to move *c* past *d* on $\langle B_1, B_2, B_4\rangle$, it must be legal to move *c* past *d* on $\langle B_1, B_3\rangle$ as well, since the dependences that could prevent that motion are wholly contained in B_1. The new copy of *c* does not lengthen execution along the path $\langle B_1, B_3\rangle$ but it does increase the overall size of the code fragment.
- The compiler can move an operation backward—that is, earlier on the path. For example, it might move *f* from B_2 to B_1. While that decision might speed execution along the path $\langle B_1, B_2, B_4\rangle$, it inserts a computation of *f* into the path $\langle B_1, B_3\rangle$. That action has two consequences. First, it lengthens the execution of $\langle B_1, B_3\rangle$. Second, it may produce incorrect code along $\langle B_1, B_3\rangle$.
  If *f* has a side effect that changes the values produced along any path leading from B_3, then the scheduler must rewrite the code to undo that effect in B_3. In some cases, renaming can cure the problem; in other cases, it must insert one or more compensating operations into B_3. These operations further slow execution along the path $\langle B_1, B_3\rangle$.

If *f* kills some value used in B_3, renaming the result of *f* can avoid the problem. If the value is live after B_4, the scheduler may need to copy it back to its original name after B_4.

The issue of compensation code also makes clear the order in which the scheduler should consider paths in an EBB. Since the first path scheduled receives little or no compensation code, the scheduler should choose paths in order of their likely execution frequency. It can either use profile data or estimates, in the same way that the global code-placement algorithm in Section 8.6.2 does.

The scheduler can take steps to mitigate the impact of compensation code. It can use live information to avoid some of the compensation code suggested by forward motion. If the result of the moved operation is not live on entry to the off-path block, no compensation code is needed in that block. It can avoid all of the compensation code needed by backward motion by simply prohibiting backward motion across block boundaries. While this restriction limits the scheduler's ability to improve the code, it avoids lengthening other paths and still allows the scheduler some opportunity for improvement.

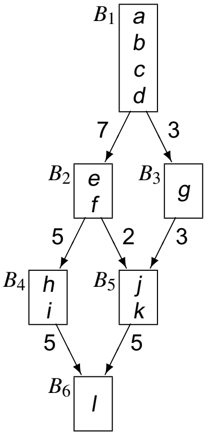
The mechanics of EBB scheduling are straightforward. To schedule an EBB path, the scheduler performs renaming, if necessary, over the region. Next, it builds a single dependence graph for the entire path, ignoring any premature exits. It computes the priority metrics needed to select among ready operations and to break ties. Finally, it applies list scheduling, as for a single block. Each time it assigns an operation to a specific instruction in a specific cycle of the schedule, it inserts any compensation code necessitated by that choice.

In this scheme, the compiler schedules each block once. In our example, the scheduler might first process the path $\langle B_1, B_2, B_4\rangle$. The next path is $\langle B_1, B_3\rangle$. Since B_1's schedule is already fixed, it will use knowledge of B_1's schedule as an initial condition when it processes B_3, but it will not change the schedule for B_1. Finally, it schedules the trivial EBBs, B_5 and B_6.

12.4.2 Trace Scheduling

Trace
an acyclic path through the CFG, selected using profile information

Trace scheduling extends the basic concept of scheduling paths beyond the range of a path through an EBB. Instead of focusing on EBBs, trace scheduling constructs maximal-length acyclic paths through the CFG and applies the list-scheduling algorithm to those paths, or *traces*. Because trace scheduling has the same issues with compensation code as EBB scheduling, the compiler should choose traces in a way that ensures that hot paths—the most frequently executed paths—are scheduled before colder paths.



To build traces for scheduling, the compiler needs access to profile information for the edges in the CFG. The diagram in the margin shows our example with execution counts on each edge. To build a trace, the scheduler can use a simple greedy approach. It begins a trace by selecting the most frequently executed edge in the CFG. In our example, it would select the edge $\langle B_1, B_2\rangle$ to create an initial trace of $\langle B_1, B_2\rangle$. It then examines the edges entering the first node of the trace or leaving the last node of the trace and chooses the edge with the highest execution count. In the example, it chooses $\langle B_2, B_4\rangle$ over $\langle B_2, B_5\rangle$ to make the trace $\langle B_1, B_2, B_4\rangle$. Since B_4 has just one successor, B_6, it chooses $\langle B_4, B_6\rangle$ as its next edge and produces the trace $\langle B_1, B_2, B_4, B_6\rangle$.

Trace construction stops when the algorithm either runs out of possible edges, as in our example, or encounters a loop-closing branch. The latter condition prevents the scheduler from constructing a trace that moves operations out of a loop. The assumption is that earlier optimization will have performed loop-invariant code motion (e.g. lazy code motion in Section 10.3.1) and that the scheduler should not put itself in the position to insert compensation code on the loop-closing branch.

Given a trace, the scheduler applies the list-scheduling algorithm to the entire trace, in the same way that EBB scheduling applies it to a path through an EBB. With an arbitrary trace, one additional opportunity for compensation code occurs; the trace may have interim entry points—blocks in mid-trace that have multiple predecessors.

- Forward code motion of an operation *i* across an interim entry point may add *i* to the off-trace path. If *i* redefines a value that is also live across the interim entry, some combination of renaming or recomputation may be necessary. The alternative is to either prohibit forward motion across the interim entry or to use cloning to avoid this situation (see Section 12.4.3).
- Backward code motion of an operation *i* across an interim entry point may necessitate adding *i* to the off-trace path. This situation is straightforward, since *i* already occurred on the off-trace path (albeit later in execution). Because the scheduler must correct for any naming issues introduced by the on-trace backward motion, the off-trace compensation code can simply define the same name.

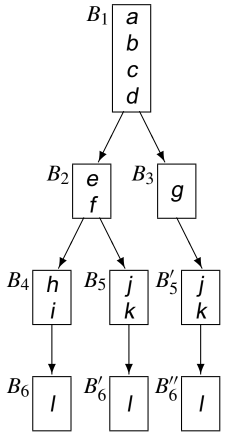
To schedule the entire procedure, the trace scheduler constructs a trace and schedules it. It then removes the blocks in the trace from consideration, and selects the next most frequently executed trace. This trace is scheduled, with the requirement that it respect any constraints imposed by previously scheduled code. The process continues, picking a trace, scheduling it, and removing it from consideration, until all the blocks have been scheduled.

EBB scheduling can be considered a degenerate case of trace scheduling in which interim entries to the trace are prohibited.

12.4.3 Cloning for Context

In our continuing example, join points in the CFG limit the opportunities for either EBB scheduling or trace scheduling. To improve the results, the compiler can clone blocks to create longer join-free paths. Superblock cloning has exactly this effect (see Section 10.6.1). For EBB scheduling, it increases the size of the EBB and the length of some of the paths through the EBB. For trace scheduling, it avoids the complications caused by interim entry points in the trace. In either case, cloning also eliminates some of the branches and jumps in the EBB.

The figure in the margin shows the CFG that might result from cloning in our continuing example. Block B_5 has been cloned to create separate instances for the path from B_2 and the path from B_3. Similarly, B_6 has been cloned twice to create a unique instance for each path that enters it. Taken together, these actions eliminate all join points in the CFG.



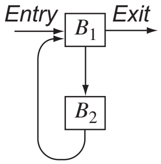
Example After Cloning

After cloning, the entire graph forms one single EBB. If the compiler decides that $\langle B_1, B_2, B_4, B_6\rangle$ is the hot path, it will schedule $\langle B_1, B_2, B_4, B_6\rangle$ first. At that point, it has two other paths to schedule. It can schedule $\langle B_5, B'_6\rangle$ using the scheduled $\langle B_1, B_2\rangle$ as a prefix. It can schedule $\langle B_3, B'_5, B''_6\rangle$ using the scheduled B_1 as a prefix. In the cloned CFG, neither of these latter choices interferes with the other.

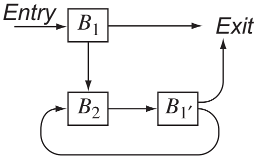
Contrast this result with the simple EBB scheduler. It scheduled B_3 with respect to B_1 and scheduled both B_5 and B_6 without prior context. Because B_5 and B_6 have multiple predecessors and inconsistent context, the EBB scheduler cannot do better than local scheduling. Cloning these blocks to give the scheduler extra context costs one copy of statements *j* and *k* and two copies of statement *l*.

In practice, the compiler can simplify the CFG by combining pairs of blocks such as B_4 and B_6 that are linked by an edge where the source has no other successors and the sink has no other predecessors. Combining such blocks eliminates the end-of-block jump in the first block of the pair.

A second situation where cloning merits consideration arises in tail-recursive programs. Recall from Sections 7.8.2 and 10.4.1 that a program is tail recursive if its last action is a recursive self-invocation. When the compiler detects a tail call, it can convert the call to a jump back to the procedure's entry. From the scheduler's point of view, cloning may improve the situation.



Tail Recursion After
Tail Call Optimization



After Cloning

The first diagram shown in the margin shows the abstracted CFG graph for a tail-recursive routine, after the tail call has been optimized. Block B_1 is entered along two paths, the path from the procedure entry and the path from B_2. This forces the scheduler to use worst-case assumptions about what precedes B_1. By cloning B_1 as shown in the lower diagram, the compiler can make control enter $B_{1'}$ along only one edge, which may improve the results of regional scheduling. To further simplify the situation, the compiler might coalesce $B_{1'}$ onto the end of B_2, creating a single-block loop body. The resulting loop can be scheduled with either a local scheduler or a loop scheduler, as appropriate.

SECTION REVIEW

Regional scheduling techniques use a variety of methods to construct longer segments of straightline code for list scheduling. The quality of the code produced by these methods is, to some extent, determined by the quality of the underlying scheduler. The infrastructure of regional scheduling simply provides more context and more operations to the list scheduler, in an attempt to provide that scheduler with more freedom and more opportunities.

All three techniques examined in this section must deal with compensation code. While compensation code introduces complications into the algorithms and may introduce delays along some paths, experience suggests that the benefits of regional scheduling outweigh the complications.

Review Questions

1. In EBB scheduling, the compiler must schedule some blocks with respect to their already-scheduled prefixes. A naive implementation might reanalyze the prescheduled blocks and rebuild their dependence graphs. What data structures could your compiler use to avoid this extra work?
2. Both trace scheduling and cloning for context try to improve on the results of EBB scheduling. Compare and contrast these approaches. How would you expect the results to differ?

12.5 ADVANCED TOPICS

Compiler optimization has, since the first FORTRAN compiler, focused on improving code in loops. The reason is simple: code inside loops executes more frequently than code outside of loops. This observation has led to the development of specialized scheduling techniques that attempt to decrease the total running time of a loop. The most widely used technique is called *software pipelining* because it builds a schedule that mimics the behavior of a hardware pipeline.

12.5.1 The Strategy of Software Pipelining

Specialized loop-scheduling techniques can create schedules that improve on the results of local scheduling, EBB scheduling, and trace scheduling for one simple reason: they can account for the flow of values around the entire loop, including the loop-closing branch. Specialized loop-scheduling techniques make sense only when the default scheduler is unable to produce compact and efficient code for the loop. If the loop body, after scheduling, contains no stalls, interlocks, or nops, then a loop scheduler is unlikely to improve its performance. Similarly, if the loop body is long enough that the end-of-loop effects are a tiny fraction of its running time, a specialized loop scheduler is unlikely to show significant improvement.

Still, many small, computationally intensive loops benefit from loop scheduling. Typically, these loops have too few operations relative to the length of their critical paths to keep the underlying hardware busy. A software pipelined loop overlaps the execution of successive iterations of the loop; in a given cycle, the loop might issue operations from two or three different iterations. These pipelined loops consist of a fixed length *kernel*, along with a prologue and an epilogue to handle the initialization and finalization of the loop. The combined effect is analogous to that of a hardware pipeline, which has distinct operations in process concurrently.

Loop kernel
The central portion of a software pipelined loop; the *kernel* executes most of the loop's iterations in an interleaved fashion.

For a pipelined loop to execute correctly, the code must first execute a prologue section that fills up the pipeline. If the kernel executes operations from three iterations of the original loop, then each kernel iteration processes roughly one-third of each active iteration of the original loop. To start execution, the prologue must perform enough work to prepare for the last third of iteration 1, the second third of iteration 2, and the first third of iteration 3. After the loop kernel completes, a corresponding epilogue is needed to complete the final iterations—emptying the pipeline. In the example, it would need to execute the final two-thirds of the penultimate iteration and the final third of the last iteration. The prologue and epilogue sections increase code size. While the specific increase is a function of the loop and the number of iterations that the kernel executes concurrently, it is not unusual for the prologue and epilogue to double the amount of code required for the loop.

To make these ideas concrete, consider the following loop, written in C:

```
for (i=1; i < 200; i++)
    z[i] = x[i] * y[i];
```

Figure 12.7 shows the code that a compiler might generate for this loop, after optimization. In this case, both operator strength reduction and linear function test replacement have been applied (see Section 10.4), so the address expressions for x, y, and z are updated with addI operations and the end of loop test has been rewritten in terms of the offset in x, eliminating the need to maintain a value for i.

The code in Figure 12.7 has been scheduled for a machine with one functional unit, assuming that loads and stores take three cycles, multiplies takes two cycles, and all other operations take one cycle. The first column shows cycle counts, normalized to the first operation in the loop (at label L_1).

Cycle	Functional Unit 0	Comments
−4	loadI @x ⇒ $r_{@x}$	Set up the loop
−3	loadI @y ⇒ $r_{@y}$	with initial loads
−2	loadI @z ⇒ $r_{@z}$	
−1	addI $r_{@x}$, 792 ⇒ r_{ub}	
1	L_1: loadAO r_{arp}, $r_{@x}$ ⇒ r_x	Get x[i] & y[i]
2	loadAO r_{arp}, $r_{@y}$ ⇒ r_y	
3	addI $r_{@x}$, 4 ⇒ $r_{@x}$	Bump the pointers
4	addI $r_{@y}$, 4 ⇒ $r_{@y}$	in shadow of loads
5	mult r_x, r_y ⇒ r_z	The actual work
6	cmp_LT $r_{@x}$, r_{ub} ⇒ r_{cc}	Shadow of mult
7	storeAO r_z ⇒ r_{arp}, $r_{@z}$	Save the result
8	addI $r_{@z}$, 4 ⇒ $r_{@z}$	Bump z's pointer
9	cbr r_{cc} → L_1, L_2	Loop-closing branch
	L_2: ...	

■ **FIGURE 12.7** Example Loop Scheduled for One Functional Unit.

The preloop code initializes a pointer for each array ($r_{@x}$, $r_{@y}$, and $r_{@z}$). It computes an upper bound for the range of $r_{@x}$ into r_{ub}; the end-of-loop test uses r_{ub}. The loop body loads x and y, performs the multiply, and stores the result into z. The scheduler has filled all of the issue slots in the shadow of long-latency operations with other operations. During the load latencies, the schedule updates $r_{@x}$ and $r_{@y}$. It performs the comparison in the multiply's shadow. It fills the slots after the store with the update of $r_{@z}$ and the branch. This produces a tight schedule for a one-functional-unit machine.

Consider what happens if we run this same code on a superscalar processor with two functional units and the same latencies. Assume that loads and stores must execute on unit 0, that functional units stall when an operation issues before its operands are ready, and that the processor cannot issue operations to a stalled unit. Figure 12.8 shows the execution trace of the loop's first iteration. The mult in cycle 3 stalls because neither r_x nor r_y is ready. It stalls in cycle 4 waiting for r_y, begins executing again in cycle 5, and produces r_z at the end of cycle 6. This forces the storeAO to stall until the start of cycle 7. Assuming that the hardware can tell that $r_{@z}$ contains an address that is distinct from $r_{@x}$ and $r_{@y}$, the processor can issue the first loadAO for the second iteration in cycle 7. If not, then the processor will stall until the store completes.

Using two functional units improved the execution time. It cut the preloop time in half, to two cycles. It reduced the time between the start of successive iterations by one-third, to six cycles. The critical path executes as quickly as

Cycle	Functional Unit 0	Functional Unit 1
−2	loadI @x ⇒ $r_{@x}$	loadI @y ⇒ $r_{@y}$
−1	loadI @z ⇒ $r_{@z}$	addI $r_{@x}$, 792 ⇒ r_{ub}
1	L_1: loadAO r_{arp}, $r_{@x}$ ⇒ r_x	*no operation issued*
2	loadAO r_{arp}, $r_{@y}$ ⇒ r_y	addI $r_{@x}$, 4 ⇒ $r_{@x}$
3	addI $r_{@y}$, 4 ⇒ $r_{@y}$	mult r_x, r_y ⇒ r_z
4	cmp_LT $r_{@x}$, r_{ub} ⇒ r_{cc}	*stall on* r_y
5	storeAO r_z ⇒ r_{arp}, $r_{@z}$	addI $r_{@z}$, 4 ⇒ $r_{@z}$
6	*stall on* r_z	cbr r_{cc} → L_1, L_2
7	. . . *start of next iteration* . . .	

■ **FIGURE 12.8** Execution Trace on a Two-Unit Superscalar Processor.

This figure shows an *execution trace*, not the scheduled code.

Cycle	Functional Unit 0	Functional Unit 1
−2	loadI @x ⇒ $r_{@x}$	loadI @y ⇒ $r_{@y}$
−1	loadI @z ⇒ $r_{@z}$	addI $r_{@x}$, 788 ⇒ r_{ub}
1	L_1: loadAO r_{arp}, $r_{@x}$ ⇒ r_x	addI $r_{@x}$, 4 ⇒ $r_{@x}$
2	loadAO r_{arp}, $r_{@y}$ ⇒ r_y	addI $r_{@y}$, 4 ⇒ $r_{@y}$
3	cmp_LT $r_{@x}$, r_{ub} ⇒ r_{cc}	nop
4	storeAO r_z ⇒ r_{arp}, $r_{@z}$	addI $r_{@z}$, 4 ⇒ $r_{@z}$
5	cbr r_{cc} → L_1, L_2	mult r_x, r_y ⇒ r_z
+1	L_2: nop	nop
+2	storeAO r_z ⇒ r_{arp}, $r_{@z}$	nop
+3	. . .	. . .

■ **FIGURE 12.9** Example Loop after Software Pipelining.

we can expect; the multiply issues before r_y is available and executes as soon as possible. The store proceeds as soon as r_z is available. Some issue slots are wasted (unit 0 in cycle 6 and unit 1 in cycles 1 and 4).

Reordering the linear code can change the execution schedule. For example, moving the update of $r_{@x}$ in front of the load from $r_{@y}$ allows the processor to issue the updates of $r_{@x}$ and $r_{@y}$ in the same cycles as the loads from those registers. This lets some of the operations issue earlier in the schedule, but it does nothing to speed up the critical path. The net result is the same—a six-cycle loop. Pipelining the code can reduce the time needed for each iteration, as shown in Figure 12.9. In this case, it reduces the number of cycles per iteration from six to five. The next subsection presents the algorithm that generated this schedule.

12.5.2 An Algorithm for Software Pipelining

To create a software pipelined loop, the scheduler follows a simple plan. First, it estimates the number of cycles in the kernel, called the *initiation interval*. Second, it tries to schedule the kernel; if that process fails, it increases the kernel size by one and tries again. (This process must halt because scheduling will succeed before the kernel size exceeds the size of the nonpipelined loop.) As the final step, the scheduler generates prologue and epilogue code to match the scheduled kernel.

Estimating Kernel Size

As an initial estimate for kernel size, the loop scheduler can compute lower bounds on the number of cycles that must be in the loop kernel.

- The compiler can estimate the minimum number of cycles in the kernel from a simple observation: every operation in the loop body must issue. It can compute the number of cycles required to issue all the operations as follows:

  $$RC = max_u(\lceil I_u / N_u \rceil)$$

  where u varies over all functional unit types u, I_u is the number of operations of type u in the loop and N_u is the number of functional units of type u. We call *RC* the resource constraint.

Recurrence
a loop-based computation that creates a cycle in the dependence graph

A recurrence must span multiple iterations.

- The compiler can estimate the minimum number of cycles in the kernel from another simple observation: the initiation interval must be long enough to allow each recurrence to complete. It can compute the a lower bound from recurrence lengths as follows:

  $$DC = max_r(\lceil d_r / k_r \rceil)$$

  where r ranges over all recurrences in the loop body, d_r is the cumulative *delay* around recurrence r, and k_r is the number of iterations that r spans. We call *DC* the dependence constraint.

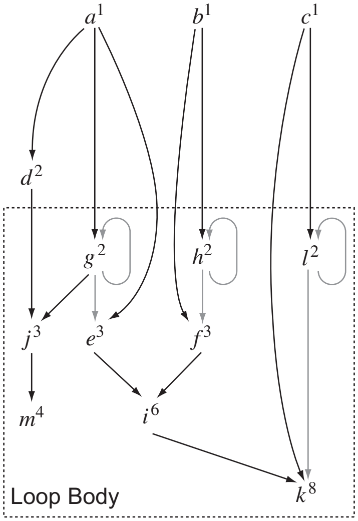
The scheduler can use $ii = max(RC, DC)$ as its first initiation interval. In our example loop, all computations are of the same type. Since the loop body contains nine operations for two functional units, that suggests a resource constraint of $\lceil 9/2 \rceil = 5$. However, the `loadAO` and `storeAO` operations can only execute on unit 0, so we must also compute $\lceil 3/1 \rceil = 3$ as the constraint for unit 0. Since $5 > 3$, *RC* is 5. From the dependence graph in Figure 12.10b, the recurrences are on `r@x`, `r@y`, and `r@z`. All three have *delay* of one and span a single iteration, so *DC* is one. Taking the larger of *RC* and *DC*, the algorithm finds an initial value for *ii* as 5.

```
a:      loadI   @x          ⇒ r@x
b:      loadI   @y          ⇒ r@y
c:      loadI   @z          ⇒ r@z
d:      addI    r@x,792     ⇒ rub

e:  L1: loadAO  rarp,r@x    ⇒ rx
f:      loadAO  rarp,r@y    ⇒ ry
g:      addI    r@x,4       ⇒ r@x
h:      addI    r@y,4       ⇒ r@y
i:      mult    rx,ry       ⇒ rz
j:      cmp_LT  r@x,rub     ⇒ rcc
k:      storeAO rz          ⇒ rarp,r@z
l:      addI    r@z,4       ⇒ r@z
m:      cbr     rcc         → L1,L2

n:  L2: ...
```

(a) Code for Example Loop



(b) Dependence Graph

■ **FIGURE 12.10** Dependence Graph for the Example Loop in Figure 12.7.

Scheduling the Kernel

To schedule the kernel, the compiler uses list scheduling with a fixed-length schedule of *ii* slots. Updates to the scheduling clock, `Cycle` in Figure 12.3, are performed modulo *ii*. Loop scheduling introduces a complication that cannot arise in straightline code (e.g. a block, an EBB, or a trace): cycles in the dependence graph.

Modulo scheduling
List scheduling with a cyclic clock is sometimes called *modulo scheduling*.

The scheduler must recognize that loop-carried dependences, such as (g, e), do not constrain the first iteration of the loop. (Loop-carried dependences are drawn in gray in Figure 12.10b.) In the first iteration, only operations *e* and *f* depend solely on values computed before the loop.

Loop-carried dependence
a dependence that represents a value carried along the CFG edge for the loop-closing branch

The loop-carried dependences also expose antidependences. In the example, an antidependence runs from *e* to *g*; the code cannot update $\texttt{r}_{\texttt{@x}}$ before using it in the load operation. Similar antidependences from *f* to *h* and from *k* to *l*. If we assume that an operation reads its operands at the start of the cycle when it issues and writes its result at the end of the cycle when the operation finishes, then the delay on an antidependence is zero. Thus scheduling the operation at the source of the antidependence satisfies the constraint from the antidependence. We will see this behavior in the example below.

Modulo scheduling the dependence graph for the loop into a five-cycle, two-functional-unit schedule produces the kernel schedule shown in Figure 12.11. In cycle 1, with an initial ready list of (e,f), the scheduler

Cycle	Functional Unit 0					Functional Unit 1			
1	L_1:	loadAO	$r_{arp}, r_{@x}$	⇒	r_x	addI	$r_{@x}$, 4	⇒	$r_{@x}$
2		loadAO	$r_{arp}, r_{@y}$	⇒	r_y	addI	$r_{@y}$, 4	⇒	$r_{@y}$
3		cmp_LT	$r_{@x}, r_{ub}$	⇒	r_{cc}	nop			
4		storeAO	r_z	⇒	$r_{arp}, r_{@z}$	addI	$r_{@z}$, 4	⇒	$r_{@z}$
5		cbr	r_{cc}	→	L_1, L_2	mult	r_x, r_y	⇒	r_z

■ **FIGURE 12.11** Kernel Schedule for the Pipelined Loop.

chooses *e*, using some tie breaker, and schedules *e* on unit 0. Scheduling *e* satisfies the antidependence to *g*. Since the only dependences entering *g* from inside the loop are loop-carried dependences, *g* is now ready and can be scheduled into unit 1 in cycle 1.

Advancing the cycle counter to 2, the ready list contains *f* and *j*. The scheduler selects *f*, breaking the tie in favor of the operation with the longer latency. It schedules *f* onto unit 0. This action satisfies the antidependence from *f* to *h*; the scheduler immediately places *h* on unit 1 in cycle 2.

In cycle 3, the ready list contains just *j*. The scheduler places it on unit 0. In cycle 4, the dependence from *j* to *m* is satisfied; however, the additional constraint that keeps a block-ending branch at the end of the block delays it for a cycle.

In cycle 4, the ready list is empty. When the cycle counter advances to cycle 5, both *m* and *i* are ready. The scheduler places them on units 0 and 1.

When the counter advances beyond cycle 5, it wraps to cycle 1. The ready list is empty, but the active list is not, so the scheduler bumps the cycle counter. In cycle 2, operation *i* has finished and operation *k* is ready. Operation *k* is a store, which must execute on unit 0. Unit 0 is busy in cycles 2 and 3, so the scheduler keeps bumping the cycle counter looking for a slot where it can place operation *k*. Finally, in cycle 4, it finds an issue slot for operation *k*.

Kernel scheduling fails when it does not find an issue slot for some operation. If that happens, the algorithm increments *ii* and tries again.

Scheduling operation *k* in cycle 4 satisfies the antidependence from *k* to *l*. The scheduler immediately schedules *l* onto unit 1 in cycle 4. The scheduler then bumps the counter until both these operations come off the active list. Since neither has any descendents in the dependence graph, both ready and active become empty and the algorithm halts.

Generating Prologue and Epilogue Code

In principle, generating the prologue and epilogue code is simple. The key insight, in both cases, is that the compiler can use the dependence graph as its guide.

To generate the prologue code, the compiler starts from each upward exposed use in the loop and follows the dependence graph in a backward scheduling phase. For each upward exposed use, it must generate the chain of operations that generate the necessary value, properly scheduled to cover their latencies. To generate the epilogue, the compiler starts from each downward exposed use in the loop and follows the dependence graph in a forward scheduling phase.

The example loop has particularly simple prologue and epilogue code, because the initiation interval is large relative to the delays in the loop. Exercise 9 at the end of the chapter shows a version of the same code with a tighter loop body and, hence, a more complex prologue and epilogue.

12.6 SUMMARY AND PERSPECTIVE

To obtain reasonable performance on a modern processor, the compiler must schedule operations carefully. Almost all modern compilers use some form of list scheduling. The algorithm is easily adapted and parameterized by changing priority schemes, tie-breaking rules, and even the direction of scheduling. List scheduling is robust, in the sense that it produces good results across a wide variety of codes. In practice, it often finds a time-optimal schedule.

Variations on list scheduling that operate over larger regions address problems that arise, at least in part, from the increased complexity of modern processors. Techniques that schedule EBBs and loops are, in essence, responses to the increase in both the number of pipelines that the compiler must consider and their individual latencies. As machines have become more complex, schedulers have needed more scheduling context to discover enough instruction-level parallelism to keep the machines busy. Software pipelining provides a way of increasing the number of operations issued per cycle and decreasing total time for executing a loop. Trace scheduling was developed for VLIW architectures, for which the compiler needed to keep many functional units busy.

■ CHAPTER NOTES

Scheduling problems arise in many domains, ranging from construction, through industrial production, through service delivery, to getting payloads onto the space shuttle. A rich literature has grown up about scheduling, including many specialized variants of the problem. Instruction scheduling has been studied as a distinct problem since the 1960s.

Algorithms that guarantee optimal schedules exist for simple situations. For example, on a machine with one functional unit and uniform operation latencies, the Sethi-Ullman labelling algorithm creates an optimal schedule for an expression tree [311]. It can be adapted to produce good code for expression DAGs. Fischer and Proebsting built on the labelling algorithm to derive an algorithm that produces optimal or near optimal results for small memory latencies [289]. Unfortunately, it has trouble when latencies rise or the number of functional units grows.

Much of the literature on instruction scheduling deals with variants on the list-scheduling algorithm described in this chapter. Landskov et al. is often cited as the definitive work on list scheduling [239], but the algorithm goes back, at least, to Heller in 1961 [187]. Other papers that build on list scheduling include Bernstein and Rodeh [39], Gibbons and Muchnick [159], and Hennessy and Gross [188]. Krishnamurthy et al. provide a high-level survey of the literature for pipelined processors [234, 320]. Kerns, Lo, and Eggers developed balanced scheduling as a way to adapt list scheduling to uncertain memory latencies [221, 249]. Schielke's RBF algorithm explored the use of randomization and repetition as an alternative to multilayered priority schemes [308].

Many authors have described regional scheduling algorithms. The first automated regional technique was Fisher's trace-scheduling algorithm [148, 149]. It has been used in several commercial systems [137, 251] and numerous research systems [318]. Hwu et al. proposed *superblock* scheduling as an alternative [201]; inside a loop, it clones blocks to avoid join points, in a fashion similar to that shown in Section 12.4.3. Click proposed a global scheduling algorithm based on the use of a global value graph [85]. Several authors have proposed techniques to make use of specific hardware features [303, 318]. Other approaches that use replication to improve scheduling include Ebcioğlu and Nakatani [136] and Gupta and Soffa [174]. Sweany and Beaty proposed choosing paths based on dominance information [327]; others have looked at various aspects of that approach [105, 199, 326].

Software pipelining has been explored extensively. Rau and Glaeser introduced the idea in 1981 [294]. Lam developed the scheme for software pipelining presented here [236]; the paper includes a hierarchical scheme for handling control flow inside a loop. Aiken and Nicolau developed a similar approach, called *perfect pipelining* [10] at the same time as Lam's work.

The example for backward versus forward scheduling in Figure 12.5 was brought to our attention by Philip Schielke [308]. He took it from the SPEC 95 benchmark program `go`. It captures, concisely, an effect that has caused many compiler writers to include both forward and backward schedulers in their compilers' back ends.

EXERCISES

Section 12.2

1. Develop an algorithm that builds the dependence graph for a basic block. Assume that the block is written in ILOC and that any values defined outside the block are ready before execution of the block begins.

2. If the primary use for a dependence graph is instruction scheduling, then accurate modeling of actual delays on the target machine is critical.

a. How should the dependence graph model the uncertainty caused by ambiguous memory references?

b. In some pipelined processors, write-after-read delays can be shorter than read-after-write delays. For example, the sequence

$$[\ \texttt{add}\ r_{10}, r_{12} \Rightarrow r_2 \mid \texttt{sub}\ r_{13}, r_{11} \Rightarrow r_{10}\ ]$$

would read the value from r_{10} for use in the `add` before writing the result of the `sub` into r_{10}. How can a compiler represent antidependences in a dependence graph for such an architecture?

c. Some processors bypass memory to reduce read-after-write delays. On these machines, a sequence such as

```
storeAI r21        ⇒ rarp, 16
loadAI  rarp, 16   ⇒ r12
```

forwards the value of the store (in r_{21} at the beginning of the sequence) directly to the result of the load (r_{12}). How can the dependence graph reflect this hardware bypass feature?

Section 12.3

3. Extend the local list-scheduling algorithm from Figure 12.3 to handle multiple functional units. Assume that all functional units have identical capabilities.

4. A critical aspect of any scheduling algorithm is the mechanism for setting initial priorities and for breaking ties when several operations with the same priority are ready at the same cycle. Some alternative tiebreakers might be:

a. Take the operations with register-based operands in preference to operations with immediate operands.

b. Take the operation whose operands were most recently defined.

c. Take a randomly chosen operation from the ready list.

d. Take a load before any computation.

For each tiebreaker, suggest a rationalization—a guess as to why someone suggested it. Which tiebreaker would you use first? Which would you use second? Justify (or rationalize) your answers.

5. Some operations, such as a register-to-register copy, can execute on almost any functional unit, albeit with a different opcode. Can the scheduler capitalize on these alternatives? Suggest modifications to the basic list-scheduling framework that allow it to use "synonyms" for a basic operation such as a copy.

6. Most modern microprocessors have *delay slots* on some or all branch operations. With a single delay slot, the operation immediately following the branch executes while the branch processes; thus, the ideal slot for scheduling a branch is in the second-to-last cycle of a basic block. (Most processors have a version of the branch that does not execute the delay slot, so that the compiler can avoid generating a `nop` instruction in an unfilled delay slot.)

a. How would you adapt the list-scheduling algorithm to improve its ability to "fill" delay slots?

b. Sketch a post-scheduling pass that would fill delay slots.

c. Propose a creative use for the branch-delay slots that cannot be filled with useful operations.

Section 12.4

7. The order in which operations occur determines when values are created and when they are used for the last time. Taken together, these effects determine the lifetime of the value.

a. How can the scheduler reduce the demand for registers? Suggest concrete tiebreaking heuristics that would fit into a list scheduler.

b. What is the interaction between these register-oriented tiebreakers and the scheduler's ability to produce short schedules?

8. Software pipelining overlaps loop iterations to create an effect that resembles hardware pipelining.

a. What impact will software pipelining have on the demand for registers?

b. How can the scheduler use predicated execution to reduce the code-space penalty for software pipelining?

Section 12.5

9. The example code in Figure 12.7 generates a five-cycle software pipelined kernel because it contains nine operations. If the compiler chose a different scheme for generating the addresses of `x`, `y`, and `z`, it could further reduce the operation count in the loop body.

Cycle	Functional Unit 0	Comments
–5	addI r_{arp}, @x ⇒ $r_{@x}$	Set up the loop
–4	addI r_{arp}, @y ⇒ $r_{@y}$	with initial loads
–3	addI r_{arp}, @z ⇒ $r_{@z}$	
–2	loadI 0 ⇒ r_{ctr}	
–1	loadI 792 ⇒ r_{ub}	
1	L_1: loadAO r_{ctr}, $r_{@x}$ ⇒ r_x	Get x[i] & y[i]
2	loadAO r_{ctr}, $r_{@y}$ ⇒ r_y	
3	mult r_x, r_y ⇒ r_z	The actual work
4	cmp_LT r_{ctr},r_{ub} ⇒ r_{cc}	Shadow of mult
5	storeAO r_z ⇒ r_{ctr}, $r_{@z}$	Save the result
6	addI r_{ctr}, 4 ⇒ $r_{@z}$	Bump the offset counter
7	cbr r_{cc} → L_1, L_2	Loop-closing branch
	L_2: ...	

This figure shows the *scheduled* code.

This scheme uses one more register, r_{ctr}, than the original version. Thus, depending on context, it might need spill code where the original did not.

a. Compute *RC* and *DC* for this version of the loop.

b. Generate the software pipelined loop body.

c. Generate the prologue and epilogue code for your pipelined loop body.