

## Chapter 10

# Instruction-Level Parallelism

Every modern high-performance processor can execute several operations in a single clock cycle. The “billion-dollar question” is how fast can a program be run on a processor with instruction-level parallelism? The answer depends on:

1. The potential parallelism in the program.
2. The available parallelism on the processor.
3. Our ability to extract parallelism from the original sequential program.
4. Our ability to find the best parallel schedule given scheduling constraints.

If all the operations in a program are highly dependent upon one another, then no amount of hardware or parallelization techniques can make the program run fast in parallel. There has been a lot of research on understanding the limits of parallelization. Typical nonnumeric applications have many inherent dependences. For example, these programs have many data-dependent branches that make it hard even to predict which instructions are to be executed, let alone decide which operations can be executed in parallel. Therefore, work in this area has focused on relaxing the scheduling constraints, including the introduction of new architectural features, rather than the scheduling techniques themselves.

Numeric applications, such as scientific computing and signal processing, tend to have more parallelism. These applications deal with large aggregate data structures; operations on distinct elements of the structure are often independent of one another and can be executed in parallel. Additional hardware resources can take advantage of such parallelism and are provided in high-performance, general-purpose machines and digital signal processors. These programs tend to have simple control structures and regular data-access patterns, and static techniques have been developed to extract the available parallelism from these programs. Code scheduling for such applications is interesting

and significant, as they offer a large number of independent operations to be mapped onto a large number of resources.

Both parallelism extraction and scheduling for parallel execution can be performed either statically in software, or dynamically in hardware. In fact, even machines with hardware scheduling can be aided by software scheduling. This chapter starts by explaining the fundamental issues in using instruction-level parallelism, which is the same regardless of whether the parallelism is managed by software or hardware. We then motivate the basic data-dependence analyses needed for the extraction of parallelism. These analyses are useful for many optimizations other than instruction-level parallelism as we shall see in Chapter 11.

Finally, we present the basic ideas in code scheduling. We describe a technique for scheduling basic blocks, a method for handling highly data-dependent control flow found in general-purpose programs, and finally a technique called software pipelining that is used primarily for scheduling numeric programs.

## 10.1 Processor Architectures

When we think of instruction-level parallelism, we usually imagine a processor issuing several operations in a single clock cycle. In fact, it is possible for a machine to issue just one operation per clock<sup>1</sup> and yet achieve instruction-level parallelism using the concept of *pipelining*. In the following, we shall first explain pipelining then discuss multiple-instruction issue.

### 10.1.1 Instruction Pipelines and Branch Delays

Practically every processor, be it a high-performance supercomputer or a standard machine, uses an *instruction pipeline*. With an instruction pipeline, a new instruction can be fetched every clock while preceding instructions are still going through the pipeline. Shown in Fig. 10.1 is a simple 5-stage instruction pipeline: it first fetches the instruction (IF), decodes it (ID), executes the operation (EX), accesses the memory (MEM), and writes back the result (WB). The figure shows how instructions  $i$ ,  $i + 1$ ,  $i + 2$ ,  $i + 3$ , and  $i + 4$  can execute at the same time. Each row corresponds to a clock tick, and each column in the figure specifies the stage each instruction occupies at each clock tick.

If the result from an instruction is available by the time the succeeding instruction needs the data, the processor can issue an instruction every clock. Branch instructions are especially problematic because until they are fetched, decoded and executed, the processor does not know which instruction will execute next. Many processors speculatively fetch and decode the immediately succeeding instructions in case a branch is not taken. When a branch is found to be taken, the instruction pipeline is emptied and the branch target is fetched.

---

<sup>1</sup>We shall refer to a clock “tick” or clock cycle simply as a “clock,” when the intent is clear.

	$i$	$i + 1$	$i + 2$	$i + 3$	$i + 4$
1.	IF				
2.	ID	IF			
3.	EX	ID	IF		
4.	MEM	EX	ID	IF	
5.	WB	MEM	EX	ID	IF
6.		WB	MEM	EX	ID
7.			WB	MEM	EX
8.				WB	MEM
9.					WB

Figure 10.1: Five consecutive instructions in a 5-stage instruction pipeline

Thus, taken branches introduce a delay in the fetch of the branch target and introduce “hiccups” in the instruction pipeline. Advanced processors use hardware to predict the outcomes of branches based on their execution history and to prefetch from the predicted target locations. Branch delays are nonetheless observed if branches are mispredicted.

10.1.2 Pipelined Execution

Some instructions take several clocks to execute. One common example is the memory-load operation. Even when a memory access hits in the cache, it usually takes several clocks for the cache to return the data. We say that the execution of an instruction is *pipelined* if succeeding instructions not dependent on the result are allowed to proceed. Thus, even if a processor can issue only one operation per clock, several operations might be in their execution stages at the same time. If the deepest execution pipeline has  $n$  stages, potentially  $n$  operations can be “in flight” at the same time. Note that not all instructions are fully pipelined. While floating-point adds and multiplies often are fully pipelined, floating-point divides, being more complex and less frequently executed, often are not.

Most general-purpose processors dynamically detect dependences between consecutive instructions and automatically stall the execution of instructions if their operands are not available. Some processors, especially those embedded in hand-held devices, leave the dependence checking to the software in order to keep the hardware simple and power consumption low. In this case, the compiler is responsible for inserting “no-op” instructions in the code if necessary to assure that the results are available when needed.

### 10.1.3 Multiple Instruction Issue

By issuing several operations per clock, processors can keep even more operations in flight. The largest number of operations that can be executed simultaneously can be computed by multiplying the instruction issue width by the average number of stages in the execution pipeline.

Like pipelining, parallelism on multiple-issue machines can be managed either by software or hardware. Machines that rely on software to manage their parallelism are known as *VLIW* (Very-Long-Instruction-Word) machines, while those that manage their parallelism with hardware are known as *superscalar* machines. VLIW machines, as their name implies, have wider than normal instruction words that encode the operations to be issued in a single clock. The compiler decides which operations are to be issued in parallel and encodes the information in the machine code explicitly. Superscalar machines, on the other hand, have a regular instruction set with an ordinary sequential-execution semantics. Superscalar machines automatically detect dependences among instructions and issue them as their operands become available. Some processors include both VLIW and superscalar functionality.

Simple hardware schedulers execute instructions in the order in which they are fetched. If a scheduler comes across a dependent instruction, it and all instructions that follow must wait until the dependences are resolved (i.e., the needed results are available). Such machines obviously can benefit from having a static scheduler that places independent operations next to each other in the order of execution.

More sophisticated schedulers can execute instructions “out of order.” Operations are independently stalled and not allowed to execute until all the values they depend on have been produced. Even these schedulers benefit from static scheduling, because hardware schedulers have only a limited space in which to buffer operations that must be stalled. Static scheduling can place independent operations close together to allow better hardware utilization. More importantly, regardless how sophisticated a dynamic scheduler is, it cannot execute instructions it has not fetched. When the processor has to take an unexpected branch, it can only find parallelism among the newly fetched instructions. The compiler can enhance the performance of the dynamic scheduler by ensuring that these newly fetched instructions can execute in parallel.

## 10.2 Code-Scheduling Constraints

Code scheduling is a form of program optimization that applies to the machine code that is produced by the code generator. Code scheduling is subject to three kinds of constraints:

1. *Control-dependence constraints.* All the operations executed in the original program must be executed in the optimized one.

2. *Data-dependence constraints.* The operations in the optimized program must produce the same results as the corresponding ones in the original program.
3. *Resource constraints.* The schedule must not oversubscribe the resources on the machine.

These scheduling constraints guarantee that the optimized program produces the same results as the original. However, because code scheduling changes the order in which the operations execute, the state of the memory at any one point may not match any of the memory states in a sequential execution. This situation is a problem if a program's execution is interrupted by, for example, a thrown exception or a user-inserted breakpoint. Optimized programs are therefore harder to debug. Note that this problem is not specific to code scheduling but applies to all other optimizations, including partial-redundancy elimination (Section 9.5) and register allocation (Section 8.8).

### 10.2.1 Data Dependence

It is easy to see that if we change the execution order of two operations that do not touch any of the same variables, we cannot possibly affect their results. In fact, even if these two operations read the same variable, we can still permute their execution. Only if an operation writes to a variable read or written by another can changing their execution order alter their results. Such pairs of operations are said to share a *data dependence*, and their relative execution order must be preserved. There are three flavors of data dependence:

1. *True dependence:* read after write. If a write is followed by a read of the same location, the read depends on the value written; such a dependence is known as a true dependence.
2. *Antidependence:* write after read. If a read is followed by a write to the same location, we say that there is an antidependence from the read to the write. The write does not depend on the read per se, but if the write happens before the read, then the read operation will pick up the wrong value. Antidependence is a byproduct of imperative programming, where the same memory locations are used to store different values. It is not a “true” dependence and potentially can be eliminated by storing the values in different locations.
3. *Output dependence:* write after write. Two writes to the same location share an output dependence. If the dependence is violated, the value of the memory location written will have the wrong value after both operations are performed.

Antidependence and output dependences are referred to as *storage-related dependences*. These are not “true” dependences and can be eliminated by using

different locations to store different values. Note that data dependences apply to both memory accesses and register accesses.

## 10.2.2 Finding Dependences Among Memory Accesses

To check if two memory accesses share a data dependence, we only need to tell if they can refer to the same location; we do not need to know which location is being accessed. For example, we can tell that addresses given by a pointer  $p$  and an offset from the same pointer  $p + 4$  cannot refer to the same location, even though we may not know what  $p$  points to. Data dependence is generally undecidable at compile time. The compiler must assume that operations may refer to the same location unless it can prove otherwise.

**Example 10.1 :** Given the code sequence

```
1)  a    =  1;
2)  *p    =  2;
3)  x     =  a;
```

unless the compiler knows that  $p$  cannot possibly point to  $a$ , it must conclude that the three operations need to execute serially. There is an output dependence flowing from statement (1) to statement (2), and there are two true dependences flowing from statements (1) and (2) to statement (3).  $\square$

Data-dependence analysis is highly sensitive to the programming language used in the program. For type-unsafe languages like C and C++, where a pointer can be cast to point to any kind of object, sophisticated analysis is necessary to prove independence between any pair of pointer-based memory accesses. Even local or global scalar variables can be accessed indirectly unless we can prove that their addresses have not been stored anywhere by any instruction in the program. In type-safe languages like Java, objects of different types are necessarily distinct from each other. Similarly, local primitive variables on the stack cannot be aliased with accesses through other names.

A correct discovery of data dependences requires a number of different forms of analysis. We shall focus on the major questions that must be resolved if the compiler is to detect all the dependences that exist in a program, and how to use this information in code scheduling. Later chapters show how these analyses are performed.

## Array Data-Dependence Analysis

Array data dependence is the problem of disambiguating between the values of indexes in array-element accesses. For example, the loop

```
for (i = 0; i < n; i++)
    A[2*i] = A[2*i+1];
```

copies odd elements in the array  $A$  to the even elements just preceding them. Because all the read and written locations in the loop are distinct from each other, there are no dependences between the accesses, and all the iterations in the loop can execute in parallel. Array data-dependence analysis, often referred to simply as *data-dependence analysis*, is very important for the optimization of numerical applications. This topic will be discussed in detail in Section 11.6.

### Pointer-Alias Analysis

We say that two pointers are *aliased* if they can refer to the same object. Pointer-alias analysis is difficult because there are many potentially aliased pointers in a program, and they can each point to an unbounded number of dynamic objects over time. To get any precision, pointer-alias analysis must be applied across all the functions in a program. This topic is discussed starting in Section 12.4.

### Interprocedural Analysis

For languages that pass parameters by reference, interprocedural analysis is needed to determine if the same variable is passed as two or more different arguments. Such aliases can create dependences between seemingly distinct parameters. Similarly, global variables can be used as parameters and thus create dependences between parameter accesses and global variable accesses. Interprocedural analysis, discussed in Chapter 12, is necessary to determine these aliases.

## 10.2.3 Tradeoff Between Register Usage and Parallelism

In this chapter we shall assume that the machine-independent intermediate representation of the source program uses an unbounded number of *pseudoregisters* to represent variables that can be allocated to registers. These variables include scalar variables in the source program that cannot be referred to by any other names, as well as temporary variables that are generated by the compiler to hold the partial results in expressions. Unlike memory locations, registers are uniquely named. Thus precise data-dependence constraints can be generated for register accesses easily.

The unbounded number of pseudoregisters used in the intermediate representation must eventually be mapped to the small number of physical registers available on the target machine. Mapping several pseudoregisters to the same physical register has the unfortunate side effect of creating artificial storage dependences that constrain instruction-level parallelism. Conversely, executing instructions in parallel creates the need for more storage to hold the values being computed simultaneously. Thus, the goal of minimizing the number of registers used conflicts directly with the goal of maximizing instruction-level parallelism. Examples 10.2 and 10.3 below illustrate this classic trade-off between storage and parallelism.

## Hardware Register Renaming

Instruction-level parallelism was first used in computer architectures as a means to speed up ordinary sequential machine code. Compilers at the time were not aware of the instruction-level parallelism in the machine and were designed to optimize the use of registers. They deliberately reordered instructions to minimize the number of registers used, and as a result, also minimized the amount of parallelism available. Example 10.3 illustrates how minimizing register usage in the computation of expression trees also limits its parallelism.

There was so little parallelism left in the sequential code that computer architects invented the concept of *hardware register renaming* to undo the effects of register optimization in compilers. Hardware register renaming dynamically changes the assignment of registers as the program runs. It interprets the machine code, stores values intended for the same register in different internal registers, and updates all their uses to refer to the right registers accordingly.

Since the artificial register-dependence constraints were introduced by the compiler in the first place, they can be eliminated by using a register-allocation algorithm that is cognizant of instruction-level parallelism. Hardware register renaming is still useful in the case when a machine's instruction set can only refer to a small number of registers. This capability allows an implementation of the architecture to map the small number of architectural registers in the code to a much larger number of internal registers dynamically.

**Example 10.2:** The code below copies the values of variables in locations **a** and **c** to variables in locations **b** and **d**, respectively, using pseudoregisters **t1** and **t2**.

```
LD t1, a    // t1 = a
ST b, t1    // b = t1
LD t2, c    // t2 = c
ST d, t2    // d = t2
```

If all the memory locations accessed are known to be distinct from each other, then the copies can proceed in parallel. However, if **t1** and **t2** are assigned the same register so as to minimize the number of registers used, the copies are necessarily serialized.   □

**Example 10.3:** Traditional register-allocation techniques aim to minimize the number of registers used when performing a computation. Consider the expression



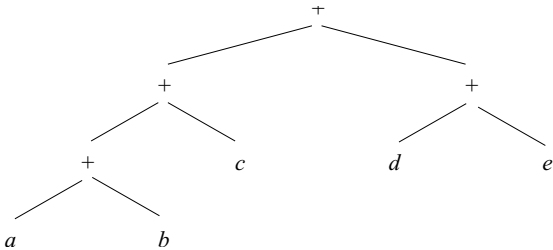


Figure 10.2: Expression tree in Example 10.3

$(a + b) + c + (d + e)$

shown as a syntax tree in Fig. 10.2. It is possible to perform this computation using three registers, as illustrated by the machine code in Fig. 10.3.

```
LD r1, a      // r1 = a
LD r2, b      // r2 = b
ADD r1, r1, r2 // r1 = r1+r2
LD r2, c      // r2 = c
ADD r1, r1, r2 // r1 = r1+r2
LD r2, d      // r2 = d
LD r3, e      // r3 = e
ADD r2, r2, r3 // r2 = r2+r3
ADD r1, r1, r2 // r1 = r1+r2
```

Figure 10.3: Machine code for expression of Fig. 10.2

The reuse of registers, however, serializes the computation. The only operations allowed to execute in parallel are the loads of the values in locations *a* and *b*, and the loads of the values in locations *d* and *e*. It thus takes a total of 7 steps to complete the computation in parallel.

Had we used different registers for every partial sum, the expression could be evaluated in 4 steps, which is the height of the expression tree in Fig. 10.2. The parallel computation is suggested by Fig. 10.4.  $\square$

r1 = a	r2 = b	r3 = c	r4 = d	r5 = e
r6 = r1+r2	r7 = r4+r5			
r8 = r6+r3				
r9 = r8+r7				

Figure 10.4: Parallel evaluation of the expression of Fig. 10.2

### 10.2.4 Phase Ordering Between Register Allocation and Code Scheduling

If registers are allocated before scheduling, the resulting code tends to have many storage dependences that limit code scheduling. On the other hand, if code is scheduled before register allocation, the schedule created may require so many registers that register *spilling* (storing the contents of a register in a memory location, so the register can be used for some other purpose) may negate the advantages of instruction-level parallelism. Should a compiler allocate registers first before it schedules the code? Or should it be the other way round? Or, do we need to address these two problems at the same time?

To answer the questions above, we must consider the characteristics of the programs being compiled. Many nonnumeric applications do not have that much available parallelism. It suffices to dedicate a small number of registers for holding temporary results in expressions. We can first apply a coloring algorithm, as in Section 8.8.4, to allocate registers for all the nontemporary variables, then schedule the code, and finally assign registers to the temporary variables.

This approach does not work for numeric applications where there are many more large expressions. We can use a hierarchical approach where code is optimized inside out, starting with the innermost loops. Instructions are first scheduled assuming that every pseudoregister will be allocated its own physical register. Register allocation is applied after scheduling and spill code is added where necessary, and the code is then rescheduled. This process is repeated for the code in the outer loops. When several inner loops are considered together in a common outer loop, the same variable may have been assigned different registers. We can change the register assignment to avoid having to copy the values from one register to another. In Section 10.5, we shall discuss the interaction between register allocation and scheduling further in the context of a specific scheduling algorithm.

### 10.2.5 Control Dependence

Scheduling operations within a basic block is relatively easy because all the instructions are guaranteed to execute once control flow reaches the beginning of the block. Instructions in a basic block can be reordered arbitrarily, as long as all the data dependences are satisfied. Unfortunately, basic blocks, especially in nonnumeric programs, are typically very small; on average, there are only about five instructions in a basic block. In addition, operations in the same block are often highly related and thus have little parallelism. Exploiting parallelism across basic blocks is therefore crucial.

An optimized program must execute all the operations in the original program. It can execute more instructions than the original, as long as the extra instructions do not change what the program does. Why would executing extra instructions speed up a program's execution? If we know that an instruction

is likely to be executed, and an idle resource is available to perform the operation “for free,” we can execute the instruction *speculatively*. The program runs faster when the speculation turns out to be correct.

An instruction  $i_1$  is said to be *control-dependent* on instruction  $i_2$  if the outcome of  $i_2$  determines whether  $i_1$  is to be executed. The notion of control dependence corresponds to the concept of nesting levels in block-structured programs. Specifically, in the if-else statement

```
if (c) s1; else s2;
```

$s1$  and  $s2$  are control dependent on  $c$ . Similarly, in the while-statement

```
while (c) s;
```

the body  $s$  is control dependent on  $c$ .

**Example 10.4:** In the code fragment

```
if (a > t)
    b = a*a;
d = a+c;
```

the statements  $b = a*a$  and  $d = a+c$  have no data dependence with any other part of the fragment. The statement  $b = a*a$  depends on the comparison  $a > t$ . The statement  $d = a+c$ , however, does not depend on the comparison and can be executed any time. Assuming that the multiplication  $a * a$  does not cause any side effects, it can be performed speculatively, as long as  $b$  is written only after  $a$  is found to be greater than  $t$ .  $\square$

### 10.2.6 Speculative Execution Support

Memory loads are one type of instruction that can benefit greatly from speculative execution. Memory loads are quite common, of course. They have relatively long execution latencies, addresses used in the loads are commonly available in advance, and the result can be stored in a new temporary variable without destroying the value of any other variable. Unfortunately, memory loads can raise exceptions if their addresses are illegal, so speculatively accessing illegal addresses may cause a correct program to halt unexpectedly. Besides, mispredicted memory loads can cause extra cache misses and page faults, which are extremely costly.

**Example 10.5:** In the fragment

```
if (p != null)
    q = *p;
```

dereferencing  $p$  speculatively will cause this correct program to halt in error if  $p$  is null.  $\square$

Many high-performance processors provide special features to support speculative memory accesses. We mention the most important ones next.

## Prefetching

The *prefetch* instruction was invented to bring data from memory to the cache before it is used. A *prefetch* instruction indicates to the processor that the program is likely to use a particular memory word in the near future. If the location specified is invalid or if accessing it causes a page fault, the processor can simply ignore the operation. Otherwise, the processor will bring the data from memory to the cache if it is not already there.

## Poison Bits

Another architectural feature called *poison bits* was invented to allow speculative load of data from memory into the register file. Each register on the machine is augmented with a *poison* bit. If illegal memory is accessed or the accessed page is not in memory, the processor does not raise the exception immediately but instead just sets the poison bit of the destination register. An exception is raised only if the contents of the register with a marked poison bit are used.

## Predicated Execution

Because branches are expensive, and mispredicted branches are even more so (see Section 10.1), *predicated instructions* were invented to reduce the number of branches in a program. A predicated instruction is like a normal instruction but has an extra predicate operand to guard its execution; the instruction is executed only if the predicate is found to be true.

As an example, a conditional move instruction `CMOVZ R2,R3,R1` has the semantics that the contents of register `R3` are moved to register `R2` only if register `R1` is zero. Code such as

```
if (a == 0)
    b = c+d;
```

can be implemented with two machine instructions, assuming that *a*, *b*, *c*, and *d* are allocated to registers `R1`, `R2`, `R4`, `R5`, respectively, as follows:

```
ADD    R3, R4, R5
CMOVZ  R2, R3, R1
```

This conversion replaces a series of instructions sharing a control dependence with instructions sharing only data dependences. These instructions can then be combined with adjacent basic blocks to create a larger basic block. More importantly, with this code, the processor does not have a chance to mispredict, thus guaranteeing that the instruction pipeline will run smoothly.

Predicated execution does come with a cost. Predicated instructions are fetched and decoded, even though they may not be executed in the end. Static schedulers must reserve all the resources needed for their execution and ensure

### Dynamically Scheduled Machines

The instruction set of a statically scheduled machine explicitly defines what can execute in parallel. However, recall from Section 10.1.2 that some machine architectures allow the decision to be made at run time about what can be executed in parallel. With dynamic scheduling, the same machine code can be run on different members of the same family (machines that implement the same instruction set) that have varying amounts of parallel-execution support. In fact, machine-code compatibility is one of the major advantages of dynamically scheduled machines.

Static schedulers, implemented in the compiler by software, can help dynamic schedulers (implemented in the machine's hardware) better utilize machine resources. To build a static scheduler for a dynamically scheduled machine, we can use almost the same scheduling algorithm as for statically scheduled machines except that `no-op` instructions left in the schedule need not be generated explicitly. The matter is discussed further in Section 10.4.7.

that all the potential data dependences are satisfied. Predicated execution should not be used aggressively unless the machine has many more resources than can possibly be used otherwise.

### 10.2.7 A Basic Machine Model

Many machines can be represented using the following simple model. A machine  $M = \langle R, T \rangle$ , consists of:

1. A set of operation types  $T$ , such as loads, stores, arithmetic operations, and so on.
2. A vector  $R = [r_1, r_2, \dots]$  representing hardware resources, where  $r_i$  is the number of units available of the  $i$ th kind of resource. Examples of typical resource types include: memory access units, ALU's, and floating-point functional units.

Each operation has a set of input operands, a set of output operands, and a resource requirement. Associated with each input operand is an input latency indicating when the input value must be available (relative to the start of the operation). Typical input operands have zero latency, meaning that the values are needed immediately, at the clock when the operation is issued. Similarly, associated with each output operand is an output latency, which indicates when the result is available, relative to the start of the operation.

Resource usage for each machine operation type  $t$  is modeled by a two-dimensional *resource-reservation table*,  $RT_t$ . The width of the table is the

number of kinds of resources in the machine, and its length is the duration over which resources are used by the operation. Entry  $RT_t[i, j]$  is the number of units of the  $j$ th resource used by an operation of type  $t$ ,  $i$  clocks after it is issued. For notational simplicity, we assume  $RT_t[i, j] = 0$  if  $i$  refers to a non-existent entry in the table (i.e.,  $i$  is greater than the number of clocks it takes to execute the operation). Of course, for any  $t$ ,  $i$ , and  $j$ ,  $RT_t[i, j]$  must be less than or equal to  $R[j]$ , the number of resources of type  $j$  that the machine has.

Typical machine operations occupy only one unit of resource at the time an operation is issued. Some operations may use more than one functional unit. For example, a multiply-and-add operation may use a multiplier in the first clock and an adder in the second. Some operations, such as a divide, may need to occupy a resource for several clocks. *Fully pipelined* operations are those that can be issued every clock, even though their results are not available until some number of clocks later. We need not model the resources of every stage of a pipeline explicitly; one single unit to represent the first stage will do. Any operation occupying the first stage of a pipeline is guaranteed the right to proceed to subsequent stages in subsequent clocks.

```

1)  a  =  b
2)  c  =  d
3)  b  =  c
4)  d  =  a
5)  c  =  d
6)  a  =  b

```

Figure 10.5: A sequence of assignments exhibiting data dependences

### 10.2.8 Exercises for Section 10.2

**Exercise 10.2.1:** The assignments in Fig. 10.5 have certain dependences. For each of the following pairs of statements, classify the dependence as (i) true dependence, (ii) antidependence, (iii) output dependence, or (iv) no dependence (i.e., the instructions can appear in either order):

- a) Statements (1) and (4).
- b) Statements (3) and (5).
- c) Statements (1) and (6).
- d) Statements (3) and (6).
- e) Statements (4) and (6).

**Exercise 10.2.2:** Evaluate the expression  $((u+v)+(w+x))+(y+z)$  exactly as parenthesized (i.e., do not use the commutative or associative laws to reorder the

additions). Give register-level machine code to provide the maximum possible parallelism.

**Exercise 10.2.3:** Repeat Exercise 10.2.2 for the following expressions:

a)  $(u + (v + (w + x))) + (y + z).$

b)  $(u + (v + w)) + (x + (y + z)).$

If instead of maximizing the parallelism, we minimized the number of registers, how many steps would the computation take? How many steps do we save by using maximal parallelism?

**Exercise 10.2.4:** The expression of Exercise 10.2.2 can be executed by the sequence of instructions shown in Fig. 10.6. If we have as much parallelism as we need, how many steps are needed to execute the instructions?

1)	LD r1, u	// r1 = u
2)	LD r2, v	// r2 = v
3)	ADD r1, r1, r2	// r1 = r1 + r2
4)	LD r2, w	// r2 = w
5)	LD r3, x	// r3 = x
6)	ADD r2, r2, r3	// r2 = r2 + r3
7)	ADD r1, r1, r2	// r1 = r1 + r2
8)	LD r2, y	// r2 = y
9)	LD r3, z	// r3 = z
10)	ADD r2, r2, r3	// r2 = r2 + r3
11)	ADD r1, r1, r2	// r1 = r1 + r2

Figure 10.6: Minimal-register implementation of an arithmetic expression

**! Exercise 10.2.5:** Translate the code fragment discussed in Example 10.4, using the CMOVZ conditional copy instruction of Section 10.2.6. What are the data dependences in your machine code?

## 10.3 Basic-Block Scheduling

We are now ready to start talking about code-scheduling algorithms. We start with the easiest problem: scheduling operations in a basic block consisting of machine instructions. Solving this problem optimally is NP-complete. But in practice, a typical basic block has only a small number of highly constrained operations, so simple scheduling techniques suffice. We shall introduce a simple but highly effective algorithm, called *list scheduling*, for this problem.

### 10.3.1 Data-Dependence Graphs

We represent each basic block of machine instructions by a *data-dependence graph*,  $G = (N, E)$ , having a set of nodes  $N$  representing the operations in the machine instructions in the block and a set of directed edges  $E$  representing the data-dependence constraints among the operations. The nodes and edges of  $G$  are constructed as follows:

1. Each operation  $n$  in  $N$  has a resource-reservation table  $RT_n$ , whose value is simply the resource-reservation table associated with the operation type of  $n$ .
2. Each edge  $e$  in  $E$  is labeled with delay  $d_e$  indicating that the destination node must be issued no earlier than  $d_e$  clocks after the source node is issued. Suppose operation  $n_1$  is followed by operation  $n_2$ , and the same location is accessed by both, with latencies  $l_1$  and  $l_2$  respectively. That is, the location's value is produced  $l_1$  clocks after the first instruction begins, and the value is needed by the second instruction  $l_2$  clocks after that instruction begins (note  $l_1 = 1$  and  $l_2 = 0$  is typical). Then, there is an edge  $n_1 \rightarrow n_2$  in  $E$  labeled with delay  $l_1 - l_2$ .

**Example 10.6:** Consider a simple machine that can execute two operations every clock. The first must be either a branch operation or an ALU operation of the form:

```
OP dst, src1, src2
```

The second must be a load or store operation of the form:

```
LD dst, addr
ST addr, src
```

The load operation (LD) is fully pipelined and takes two clocks. However, a load can be followed immediately by a store ST that writes to the memory location read. All other operations complete in one clock.

Shown in Fig. 10.7 is the dependence graph of an example of a basic block and its resources requirement. We might imagine that R1 is a stack pointer, used to access data on the stack with offsets such as 0 or 12. The first instruction loads register R2, and the value loaded is not available until two clocks later. This observation explains the label 2 on the edges from the first instruction to the second and fifth instructions, each of which needs the value of R2. Similarly, there is a delay of 2 on the edge from the third instruction to the fourth; the value loaded into R3 is needed by the fourth instruction, and not available until two clocks after the third begins.

Since we do not know how the values of R1 and R7 relate, we have to consider the possibility that an address like 8(R1) is the same as the address 0(R7).



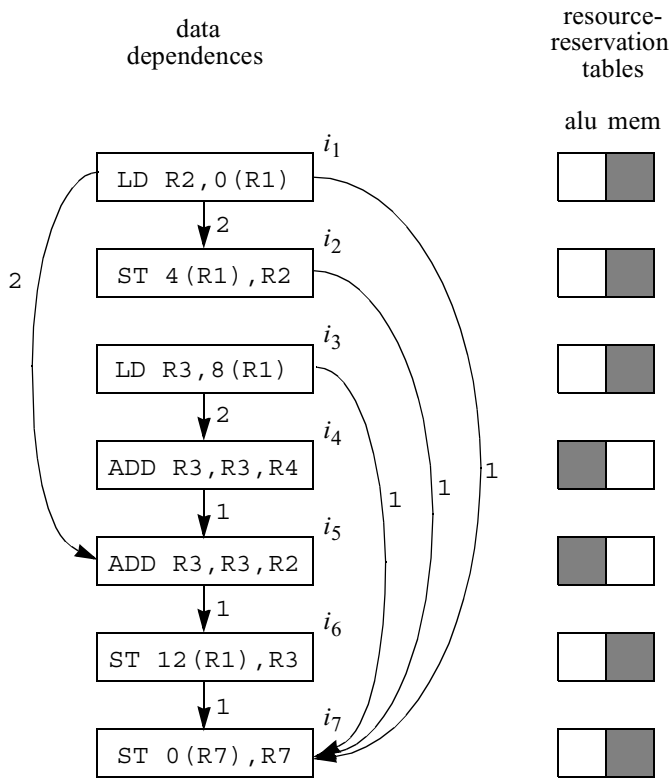


Figure 10.7: Data-dependence graph for Example 10.6

That is, the last instruction may be storing into the same address that the third instruction loads from. The machine model we are using allows us to store into a location one clock after we load from that location, even though the value to be loaded will not appear in a register until one clock later. This observation explains the label 1 on the edge from the third instruction to the last. The same reasoning explains the edge and label from the first instruction to the last. The other edges with label 1 are explained by a dependence or possible dependence conditioned on the value of R7. □

### 10.3.2 List Scheduling of Basic Blocks

The simplest approach to scheduling basic blocks involves visiting each node of the data-dependence graph in “prioritized topological order.” Since there can be no cycles in a data-dependence graph, there is always at least one topological order for the nodes. However, among the possible topological orders, some may be preferable to others. We discuss in Section 10.3.3 some of the strategies for

### Pictorial Resource-Reservation Tables

It is frequently useful to visualize a resource-reservation table for an operation by a grid of solid and open squares. Each column corresponds to one of the resources of the machine, and each row corresponds to one of the clocks during which the operation executes. Assuming that the operation never needs more than one unit of any one resource, we may represent 1's by solid squares, and 0's by open squares. In addition, if the operation is fully pipelined, then we only need to indicate the resources used at the first row, and the resource-reservation table becomes a single row.

This representation is used, for instance, in Example 10.6. In Fig. 10.7 we see resource-reservation tables as rows. The two addition operations require the “alu” resource, while the loads and stores require the “mem” resource.

picking a topological order, but for the moment, we just assume that there is some algorithm for picking a preferred order.

The list-scheduling algorithm we shall describe next visits the nodes in the chosen prioritized topological order. The nodes may or may not wind up being scheduled in the same order as they are visited. But the instructions are placed in the schedule as early as possible, so there is a tendency for instructions to be scheduled in approximately the order visited.

In more detail, the algorithm computes the earliest time slot in which each node can be executed, according to its data-dependence constraints with the previously scheduled nodes. Next, the resources needed by the node are checked against a resource-reservation table that collects all the resources committed so far. The node is scheduled in the earliest time slot that has sufficient resources.

**Algorithm 10.7:** List scheduling a basic block.

**INPUT:** A machine-resource vector  $R = [r_1, r_2, \dots]$ , where  $r_i$  is the number of units available of the  $i$ th kind of resource, and a data-dependence graph  $G = (N, E)$ . Each operation  $n$  in  $N$  is labeled with its resource-reservation table  $RT_n$ ; each edge  $e = n_1 \rightarrow n_2$  in  $E$  is labeled with  $d_e$  indicating that  $n_2$  must execute no earlier than  $d_e$  clocks after  $n_1$ .

**OUTPUT:** A schedule  $S$  that maps the operations in  $N$  into time slots in which the operations can be initiated satisfying all the data and resources constraints.

**METHOD:** Execute the program in Fig. 10.8. A discussion of what the “prioritized topological order” might be follows in Section 10.3.3.  $\square$

```

RT = an empty reservation table;
for (each  $n$  in  $N$  in prioritized topological order) {
     $s = \max_{e=p \rightarrow n \text{ in } E} (S(p) + d_e)$ ;
    /* Find the earliest time this instruction could begin,
       given when its predecessors started. */
    while (there exists  $i$  such that  $RT[s + i] + RT_n[i] > R$ )
         $s = s + 1$ ;
    /* Delay the instruction further until the needed
       resources are available. */
     $S(n) = s$ ;
    for (all  $i$ )
         $RT[s + i] = RT[s + i] + RT_n[i]$ 
}

```

Figure 10.8: A list scheduling algorithm

### 10.3.3 Prioritized Topological Orders

List scheduling does not backtrack; it schedules each node once and only once. It uses a heuristic priority function to choose among the nodes that are ready to be scheduled next. Here are some observations about possible prioritized orderings of the nodes:

- Without resource constraints, the shortest schedule is given by the *critical path*, the longest path through the data-dependence graph. A metric useful as a priority function is the *height* of the node, which is the length of a longest path in the graph originating from the node.
- On the other hand, if all operations are independent, then the length of the schedule is constrained by the resources available. The critical resource is the one with the largest ratio of uses to the number of units of that resource available. Operations using more critical resources may be given higher priority.
- Finally, we can use the source ordering to break ties between operations; the operation that shows up earlier in the source program should be scheduled first.

**Example 10.8:** For the data-dependence graph in Fig. 10.7, the critical path, including the time to execute the last instruction, is 6 clocks. That is, the critical path is the last five nodes, from the load of R3 to the store of R7. The total of the delays on the edges along this path is 5, to which we add 1 for the clock needed for the last instruction.

Using the height as the priority function, Algorithm 10.7 finds an optimal schedule as shown in Fig. 10.9. Notice that we schedule the load of R3 first, since it has the greatest height. The add of R3 and R4 has the resources to be

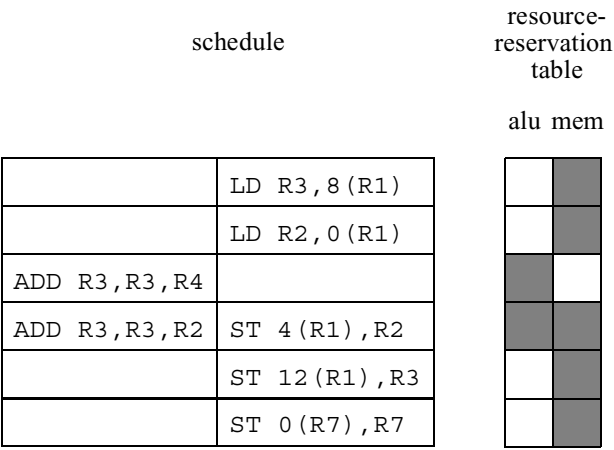


Figure 10.9: Result of applying list scheduling to the example in Fig. 10.7

scheduled at the second clock, but the delay of 2 for a load forces us to wait until the third clock to schedule this add. That is, we cannot be sure that R3 will have its needed value until the beginning of clock 3. □

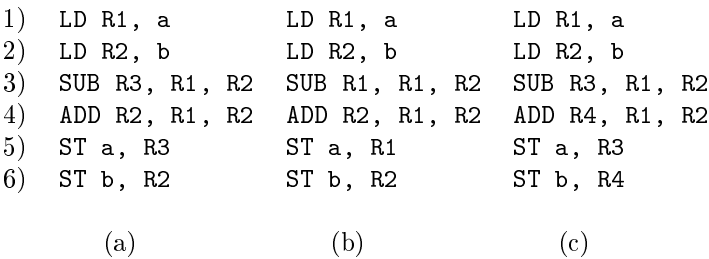


Figure 10.10: Machine code for Exercise 10.3.1

10.3.4 Exercises for Section 10.3

**Exercise 10.3.1:** For each of the code fragments of Fig. 10.10, draw the data-dependence graph.

**Exercise 10.3.2:** Assume a machine with one ALU resource (for the ADD and SUB operations) and one MEM resource (for the LD and ST operations). Assume that all operations require one clock, except for the LD, which requires two. However, as in Example 10.6, a ST on the same memory location can commence one clock after a LD on that location commences. Find a shortest schedule for each of the fragments in Fig. 10.10.

**Exercise 10.3.3:** Repeat Exercise 10.3.2 assuming:

- i.* The machine has one ALU resource and two MEM resources.
- ii.* The machine has two ALU resources and one MEM resource.
- iii.* The machine has two ALU resources and two MEM resources.

```
1)  LD R1, a
2)  ST b, R1
3)  LD R2, c
4)  ST c, R1
5)  LD R1, d
6)  ST d, R2
7)  ST a, R1
```

Figure 10.11: Machine code for Exercise 10.3.4

**Exercise 10.3.4:** Assuming the machine model of Example 10.6 (as in Exercise 10.3.2):

- a) Draw the data dependence graph for the code of Fig. 10.11.
- b) What are all the critical paths in your graph from part (a)?
- ! c) Assuming unlimited MEM resources, what are all the possible schedules for the seven instructions?

## 10.4 Global Code Scheduling

For a machine with a moderate amount of instruction-level parallelism, schedules created by compacting individual basic blocks tend to leave many resources idle. In order to make better use of machine resources, it is necessary to consider code-generation strategies that move instructions from one basic block to another. Strategies that consider more than one basic block at a time are referred to as *global scheduling* algorithms. To do global scheduling correctly, we must consider not only data dependences but also control dependences. We must ensure that

- 1. All instructions in the original program are executed in the optimized program, and
- 2. While the optimized program may execute extra instructions speculatively, these instructions must not have any unwanted side effects.

### 10.4.1 Primitive Code Motion

Let us first study the issues involved in moving operations around by way of a simple example.

**Example 10.9:** Suppose we have a machine that can execute any two operations in a single clock. Every operation executes with a delay of one clock, except for the load operation, which has a latency of two clocks. For simplicity, we assume that all memory accesses in the example are valid and will hit in the cache. Figure 10.12(a) shows a simple flow graph with three basic blocks. The code is expanded into machine operations in Figure 10.12(b). All the instructions in each basic block must execute serially because of data dependencies; in fact, a no-op instruction has to be inserted in every basic block.

Assume that the addresses of variables  $a$ ,  $b$ ,  $c$ ,  $d$ , and  $e$  are distinct and that those addresses are stored in registers R1 through R5, respectively. The computations from different basic blocks therefore share no data dependencies. We observe that all the operations in block  $B_3$  are executed regardless of whether the branch is taken, and can therefore be executed in parallel with operations from block  $B_1$ . We cannot move operations from  $B_1$  down to  $B_3$ , because they are needed to determine the outcome of the branch.

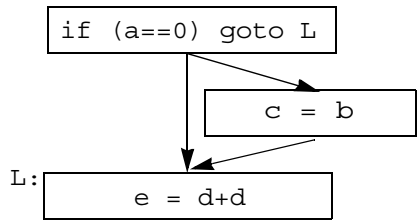
Operations in block  $B_2$  are control-dependent on the test in block  $B_1$ . We can perform the load from  $B_2$  speculatively in block  $B_1$  for free and shave two clocks from the execution time whenever the branch is taken.

Stores should not be performed speculatively because they overwrite the old value in a memory location. It is possible, however, to delay a store operation. We cannot simply place the store operation from block  $B_2$  in block  $B_3$ , because it should only be executed if the flow of control passes through block  $B_2$ . However, we can place the store operation in a duplicated copy of  $B_3$ . Figure 10.12(c) shows such an optimized schedule. The optimized code executes in 4 clocks, which is the same as the time it takes to execute  $B_3$  alone.

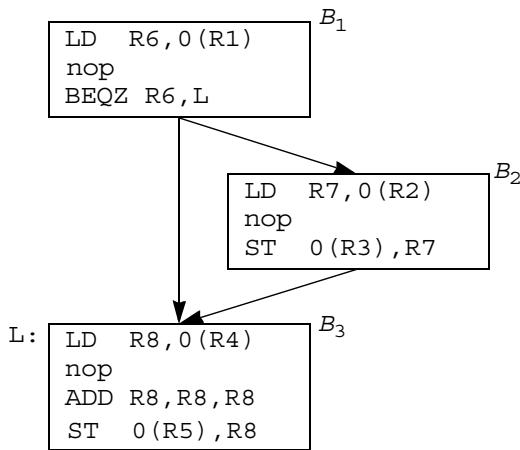
□

Example 10.9 shows that it is possible to move operations up and down an execution path. Every pair of basic blocks in this example has a different “dominance relation,” and thus the considerations of when and how instructions can be moved between each pair are different. As discussed in Section 9.6.1, a block  $B$  is said to dominate block  $B'$  if every path from the entry of the control-flow graph to  $B'$  goes through  $B$ . Similarly, a block  $B$  *postdominates* block  $B'$  if every path from  $B'$  to the exit of the graph goes through  $B$ . When  $B$  dominates  $B'$  and  $B'$  postdominates  $B$ , we say that  $B$  and  $B'$  are *control equivalent*, meaning that one is executed when and only when the other is. For the example in Fig. 10.12, assuming  $B_1$  is the entry and  $B_3$  the exit,

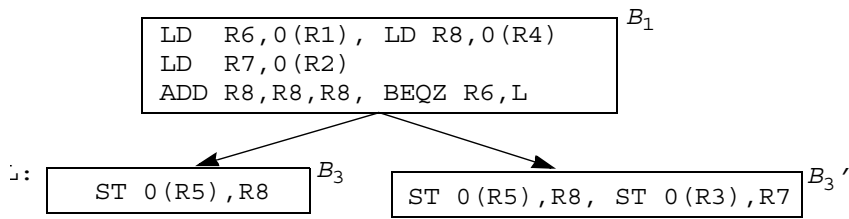
1.  $B_1$  and  $B_3$  are control equivalent:  $B_1$  dominates  $B_3$  and  $B_3$  postdominates  $B_1$ ,
2.  $B_1$  dominates  $B_2$  but  $B_2$  does not postdominate  $B_1$ , and



(a) Source program



(b) Locally scheduled machine code



(c) Globally scheduled machine code

Figure 10.12: Flow graphs before and after global scheduling in Example 10.9

3.  $B_2$  does not dominate  $B_3$  but  $B_3$  postdominates  $B_2$ .

It is also possible for a pair of blocks along a path to share neither a dominance nor postdominance relation.

### 10.4.2 Upward Code Motion

We now examine carefully what it means to move an operation up a path. Suppose we wish to move an operation from block *src* up a control-flow path to block *dst*. We assume that such a move does not violate any data dependences and that it makes paths through *dst* and *src* run faster. If *dst* dominates *src*, and *src* postdominates *dst*, then the operation moved is executed once and only once, when it should.

#### If *src* does not postdominate *dst*

Then there exists a path that passes through *dst* that does not reach *src*. An extra operation would have been executed in this case. This code motion is illegal unless the operation moved has no unwanted side effects. If the moved operation executes “for free” (i.e., it uses only resources that otherwise would be idle), then this move has no cost. It is beneficial only if the control flow reaches *src*.

#### If *dst* does not dominate *src*

Then there exists a path that reaches *src* without first going through *dst*. We need to insert copies of the moved operation along such paths. We know how to achieve exactly that from our discussion of partial redundancy elimination in Section 9.5. We place copies of the operation along basic blocks that form a cut set separating the entry block from *src*. At each place where the operation is inserted, the following constraints must be satisfied:

1. The operands of the operation must hold the same values as in the original,
2. The result does not overwrite a value that is still needed, and
3. It itself is not subsequently overwritten before reaching *src*.

These copies render the original instruction in *src* fully redundant, and it thus can be eliminated.

We refer to the extra copies of the operation as *compensation code*. As discussed in Section 9.5, basic blocks can be inserted along critical edges to create places for holding such copies. The compensation code can potentially make some paths run slower. Thus, this code motion improves program execution only if the optimized paths are executed more frequently than the nonoptimized ones.



### 10.4.3 Downward Code Motion

Suppose we are interested in moving an operation from block *src* down a control-flow path to block *dst*. We can reason about such code motion in the same way as above.

#### If *src* does not dominate *dst*

Then there exists a path that reaches *dst* without first visiting *src*. Again, an extra operation will be executed in this case. Unfortunately, downward code motion is often applied to writes, which have the side effects of overwriting old values. We can get around this problem by replicating the basic blocks along the paths from *src* to *dst*, and placing the operation only in the new copy of *dst*. Another approach, if available, is to use predicated instructions. We guard the operation moved with the predicate that guards the *src* block. Note that the predicated instruction must be scheduled only in a block dominated by the computation of the predicate, because the predicate would not be available otherwise.

#### If *dst* does not postdominate *src*

As in the discussion above, compensation code needs to be inserted so that the operation moved is executed on all paths not visiting *dst*. This transformation is again analogous to partial redundancy elimination, except that the copies are placed below the *src* block in a cut set that separates *src* from the exit.

### Summary of Upward and Downward Code Motion

From this discussion, we see that there is a range of possible global code motions which vary in terms of benefit, cost, and implementation complexity. Figure 10.13 shows a summary of these various code motions; the lines correspond to the following four cases:

	up: <i>src</i> postdom <i>dst</i>	<i>dst</i> dom <i>src</i>	speculation	compensation
	down: <i>src</i> dom <i>dst</i>	<i>dst</i> postdom <i>src</i>	code dup.	code
1	yes	yes	no	no
2	no	yes	yes	no
3	yes	no	no	yes
4	no	no	yes	yes

Figure 10.13: Summary of code motions

1. Moving instructions between control-equivalent blocks is simplest and most cost effective. No extra operations are ever executed and no compensation code is needed.

2. Extra operations may be executed if the source does not postdominate (dominate) the destination in upward (downward) code motion. This code motion is beneficial if the extra operations can be executed for free, and the path passing through the source block is executed.
3. Compensation code is needed if the destination does not dominate (post-dominate) the source in upward (downward) code motion. The paths with the compensation code may be slowed down, so it is important that the optimized paths are more frequently executed.
4. The last case combines the disadvantages of the second and third case: extra operations may be executed and compensation code is needed.

#### 10.4.4 Updating Data Dependences

As illustrated by Example 10.10 below, code motion can change the data-dependence relations between operations. Thus data dependences must be updated after each code movement.

**Example 10.10:** For the flow graph shown in Fig. 10.14, either assignment to  $x$  can be moved up to the top block, since all the dependences in the original program are preserved with this transformation. However, once we have moved one assignment up, we cannot move the other. More specifically, we see that variable  $x$  is not live on exit in the top block before the code motion, but it is live after the motion. If a variable is live at a program point, then we cannot move speculative definitions to the variable above that program point.  $\square$

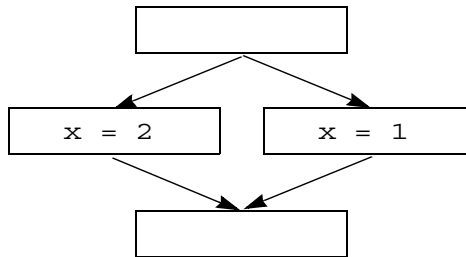


Figure 10.14: Example illustrating the change in data dependences due to code motion.

#### 10.4.5 Global Scheduling Algorithms

We saw in the last section that code motion can benefit some paths while hurting the performance of others. The good news is that instructions are not all created equal. In fact, it is well established that over 90% of a program's execution time is spent on less than 10% of the code. Thus, we should aim to

make the frequently executed paths run faster while possibly making the less frequent paths run slower.

There are a number of techniques a compiler can use to estimate execution frequencies. It is reasonable to assume that instructions in the innermost loops are executed more often than code in outer loops, and that branches that go backward are more likely to be taken than not taken. Also, branch statements found to guard program exits or exception-handling routines are unlikely to be taken. The best frequency estimates, however, come from dynamic profiling. In this technique, programs are instrumented to record the outcomes of conditional branches as they run. The programs are then run on representative inputs to determine how they are likely to behave in general. The results obtained from this technique have been found to be quite accurate. Such information can be fed back to the compiler to use in its optimizations.

### Region-Based Scheduling

We now describe a straightforward global scheduler that supports the two easiest forms of code motion:

1. Moving operations up to control-equivalent basic blocks, and
2. Moving operations speculatively up one branch to a dominating predecessor.

Recall from Section 9.7.1 that a region is a subset of a control-flow graph that can be reached only through one entry block. We may represent any procedure as a hierarchy of regions. The entire procedure constitutes the top-level region, nested in it are subregions representing the natural loops in the function. We assume that the control-flow graph is reducible.

**Algorithm 10.11:** Region-based scheduling.

**INPUT:** A control-flow graph and a machine-resource description.

**OUTPUT:** A schedule  $S$  mapping each instruction to a basic block and a time slot.

**METHOD:** Execute the program in Fig. 10.15. Some shorthand terminology should be apparent: *ControlEquiv*( $B$ ) is the set of blocks that are control-equivalent to block  $B$ , and *DominatedSucc* applied to a set of blocks is the set of blocks that are successors of at least one block in the set and are dominated by all.

Code scheduling in Algorithm 10.11 proceeds from the innermost regions to the outermost. When scheduling a region, each nested subregion is treated as a black box; instructions are not allowed to move in or out of a subregion. They can, however, move around a subregion, provided their data and control dependences are satisfied.

```

for (each region  $R$  in topological order, so that inner regions
      are processed before outer regions) {
  compute data dependences;
  for (each basic block  $B$  of  $R$  in prioritized topological order) {
     $CandBlocks = ControlEquiv(B) \cup$ 
       $DominatedSucc(ControlEquiv(B));$ 
     $CandInsts =$  ready instructions in  $CandBlocks$ ;
    for ( $t = 0, 1, \dots$  until all instructions from  $B$  are scheduled) {
      for (each instruction  $n$  in  $CandInsts$  in priority order)
        if ( $n$  has no resource conflicts at time  $t$ ) {
           $S(n) = \langle B, t \rangle$ ;
          update resource commitments;
          update data dependences;
        }
      update  $CandInsts$ ;
    }
  }
}

```

Figure 10.15: A region-based global scheduling algorithm

All control and dependence edges flowing back to the header of the region are ignored, so the resulting control-flow and data-dependence graphs are acyclic. The basic blocks in each region are visited in topological order. This ordering guarantees that a basic block is not scheduled until all the instructions it depends on have been scheduled. Instructions to be scheduled in a basic block  $B$  are drawn from all the blocks that are control-equivalent to  $B$  (including  $B$ ), as well as their immediate successors that are dominated by  $B$ .

A list-scheduling algorithm is used to create the schedule for each basic block. The algorithm keeps a list of candidate instructions,  $CandInsts$ , which contains all the instructions in the candidate blocks whose predecessors all have been scheduled. It creates the schedule clock-by-clock. For each clock, it checks each instruction from the  $CandInsts$  in priority order and schedules it in that clock if resources permit. Algorithm 10.11 then updates  $CandInsts$  and repeats the process, until all instructions from  $B$  are scheduled.

The priority order of instructions in  $CandInsts$  uses a priority function similar to that discussed in Section 10.3. We make one important modification, however. We give instructions from blocks that are control equivalent to  $B$  higher priority than those from the successor blocks. The reason is that instructions in the latter category are only speculatively executed in block  $B$ .

□

### Loop Unrolling

In region-based scheduling, the boundary of a loop iteration is a barrier to code motion. Operations from one iteration cannot overlap with those from another. One simple but highly effective technique to mitigate this problem is to unroll the loop a small number of times before code scheduling. A for-loop such as

```
for (i = 0; i < N; i++) {
    S(i);
}
```

can be written as in Fig. 10.16(a). Similarly, a repeat-loop such as

```
repeat
    S;
until C;
```

can be written as in Fig. 10.16(b). Unrolling creates more instructions in the loop body, permitting global scheduling algorithms to find more parallelism.

```
for (i = 0; i+4 < N; i+=4) {
    S(i);
    S(i+1);
    S(i+2);
    S(i+3);
}
for ( ; i < N; i++) {
    S(i);
}
```

(a) Unrolling a for-loop.

```
repeat {
    S;
    if (C) break;
    S;
    if (C) break;
    S;
    if (C) break;
    S;
} until C;
```

(b) Unrolling a repeat-loop.

Figure 10.16: Unrolled loops

## Neighborhood Compaction

Algorithm 10.11 only supports the first two forms of code motion described in Section 10.4.1. Code motions that require the introduction of compensation code can sometimes be useful. One way to support such code motions is to follow the region-based scheduling with a simple pass. In this pass, we can examine each pair of basic blocks that are executed one after the other, and check if any operation can be moved up or down between them to improve the execution time of those blocks. If such a pair is found, we check if the instruction to be moved needs to be duplicated along other paths. The code motion is made if it results in an expected net gain.

This simple extension can be quite effective in improving the performance of loops. For instance, it can move an operation at the beginning of one iteration to the end of the preceding iteration, while also moving the operation from the first iteration out of the loop. This optimization is particularly attractive for tight loops, which are loops that execute only a few instructions per iteration. However, the impact of this technique is limited by the fact that each code-motion decision is made locally and independently.

## 10.4.6 Advanced Code Motion Techniques

If our target machine is statically scheduled and has plenty of instruction-level parallelism, we may need a more aggressive algorithm. Here is a high-level description of further extensions:

1. To facilitate the extensions below, we can add new basic blocks along control-flow edges originating from blocks with more than one predecessor. These basic blocks will be eliminated at the end of code scheduling if they are empty. A useful heuristic is to move instructions out of a basic block that is nearly empty, so that the block can be eliminated completely.
2. In Algorithm 10.11, the code to be executed in each basic block is scheduled once and for all as each block is visited. This simple approach suffices because the algorithm can only move operations up to dominating blocks. To allow motions that require the addition of compensation code, we take a slightly different approach. When we visit block  $B$ , we only schedule instructions from  $B$  and all its control-equivalent blocks. We first try to place these instructions in predecessor blocks, which have already been visited and for which a partial schedule already exists. We try to find a destination block that would lead to an improvement on a frequently executed path and then place copies of the instruction on other paths to guarantee correctness. If the instructions cannot be moved up, they are scheduled in the current basic block as before.
3. Implementing downward code motion is harder in an algorithm that visits basic blocks in topological order, since the target blocks have yet to be

scheduled. However, there are relatively fewer opportunities for such code motion anyway. We move all operations that

- (a) can be moved, and
- (b) cannot be executed for free in their native block.

This simple strategy works well if the target machine is rich with many unused hardware resources.

### 10.4.7 Interaction with Dynamic Schedulers

A dynamic scheduler has the advantage that it can create new schedules according to the run-time conditions, without having to encode all these possible schedules ahead of time. If a target machine has a dynamic scheduler, the static scheduler's primary function is to ensure that instructions with high latency are fetched early so that the dynamic scheduler can issue them as early as possible.

Cache misses are a class of unpredictable events that can make a big difference to the performance of a program. If data-prefetch instructions are available, the static scheduler can help the dynamic scheduler significantly by placing these prefetch instructions early enough that the data will be in the cache by the time they are needed. If prefetch instructions are not available, it is useful for a compiler to estimate which operations are likely to miss and try to issue them early.

If dynamic scheduling is not available on the target machine, the static scheduler must be conservative and separate every data-dependent pair of operations by the minimum delay. If dynamic scheduling is available, however, the compiler only needs to place the data-dependent operations in the correct order to ensure program correctness. For best performance, the compiler should assign long delays to dependences that are likely to occur and short ones to those that are not likely.

Branch misprediction is an important cause of loss in performance. Because of the long misprediction penalty, instructions on rarely executed paths can still have a significant effect on the total execution time. Higher priority should be given to such instructions to reduce the cost of misprediction.

### 10.4.8 Exercises for Section 10.4

**Exercise 10.4.1:** Show how to unroll the generic while-loop

```
while (C)
    S;
```

**! Exercise 10.4.2:** Consider the code fragment:

```
if (x == 0) a = b;
else a = c;
d = a;
```

Assume a machine that uses the delay model of Example 10.6 (loads take two clocks, all other instructions take one clock). Also assume that the machine can execute any two instructions at once. Find a shortest possible execution of this fragment. Do not forget to consider which register is best used for each of the copy steps. Also, remember to exploit the information given by register descriptors as was described in Section 8.6, to avoid unnecessary loads and stores.

## 10.5 Software Pipelining

As discussed in the introduction of this chapter, numerical applications tend to have much parallelism. In particular, they often have loops whose iterations are completely independent of one another. These loops, known as *do-all* loops, are particularly attractive from a parallelization perspective because their iterations can be executed in parallel to achieve a speed-up linear in the number of iterations in the loop. Do-all loops with many iterations have enough parallelism to saturate all the resources on a processor. It is up to the scheduler to take full advantage of the available parallelism. This section describes an algorithm, known as *software pipelining*, that schedules an entire loop at a time, taking full advantage of the parallelism across iterations.

### 10.5.1 Introduction

We shall use the do-all loop in Example 10.12 throughout this section to explain software pipelining. We first show that scheduling across iterations is of great importance, because there is relatively little parallelism among operations in a single iteration. Next, we show that loop unrolling improves performance by overlapping the computation of unrolled iterations. However, the boundary of the unrolled loop still poses as a barrier to code motion, and unrolling still leaves a lot of performance “on the table.” The technique of software pipelining, on the other hand, overlaps a number of consecutive iterations continually until it runs out of iterations. This technique allows software pipelining to produce highly efficient and compact code.

**Example 10.12:** Here is a typical do-all loop:

```
for (i = 0; i < n; i++)  
    D[i] = A[i]*B[i] + c;
```

Iterations in the above loop write to different memory locations, which are themselves distinct from any of the locations read. Therefore, there are no memory dependences between the iterations, and all iterations can proceed in parallel.

We adopt the following model as our target machine throughout this section. In this model



- The machine can issue in a single clock: one load, one store, one arithmetic operation, and one branch operation.
- The machine has a loop-back operation of the form

```
BL R, L
```

which decrements register  $R$  and, unless the result is 0, branches to location  $L$ .

- Memory operations have an auto-increment addressing mode, denoted by  $++$  after the register. The register is automatically incremented to point to the next consecutive address after each access.
- The arithmetic operations are fully pipelined; they can be initiated every clock but their results are not available until 2 clocks later. All other instructions have a single-clock latency.

If iterations are scheduled one at a time, the best schedule we can get on our machine model is shown in Fig. 10.17. Some assumptions about the layout of the data also also indicated in that figure: registers  $R1$ ,  $R2$ , and  $R3$  hold the addresses of the beginnings of arrays  $A$ ,  $B$ , and  $D$ , register  $R4$  holds the constant  $c$ , and register  $R10$  holds the value  $n - 1$ , which has been computed outside the loop. The computation is mostly serial, taking a total of 7 clocks; only the loop-back instruction is overlapped with the last operation in the iteration.  $\square$

```
// R1, R2, R3 = &A, &B, &D
// R4          = c
// R10         = n-1

L:  LD  R5, 0(R1++)
    LD  R6, 0(R2++)
    MUL R7, R5, R6
    nop
    ADD R8, R7, R4
    nop
    ST  0(R3++), R8      BL R10, L
```

Figure 10.17: Locally scheduled code for Example 10.12

In general, we get better hardware utilization by unrolling several iterations of a loop. However, doing so also increases the code size, which in turn can have a negative impact on overall performance. Thus, we have to compromise, picking a number of times to unroll a loop that gets most of the performance improvement, yet doesn't expand the code too much. The next example illustrates the tradeoff.

**Example 10.13:** While hardly any parallelism can be found in each iteration of the loop in Example 10.12, there is plenty of parallelism across the iterations. Loop unrolling places several iterations of the loop in one large basic block, and a simple list-scheduling algorithm can be used to schedule the operations to execute in parallel. If we unroll the loop in our example four times and apply Algorithm 10.7 to the code, we can get the schedule shown in Fig. 10.18. (For simplicity, we ignore the details of register allocation for now). The loop executes in 13 clocks, or one iteration every 3.25 clocks.

A loop unrolled  $k$  times takes at least  $2k + 5$  clocks, achieving a throughput of one iteration every  $2 + 5/k$  clocks. Thus, the more iterations we unroll, the faster the loop runs. As  $k \rightarrow \infty$ , a fully unrolled loop can execute on average an iteration every two clocks. However, the more iterations we unroll, the larger the code gets. We certainly cannot afford to unroll all the iterations in a loop. Unrolling the loop 4 times produces code with 13 instructions, or 163% of the optimum; unrolling the loop 8 times produces code with 21 instructions, or 131% of the optimum. Conversely, if we wish to operate at, say, only 110% of the optimum, we need to unroll the loop 25 times, which would result in code with 55 instructions.  $\square$

## 10.5.2 Software Pipelining of Loops

Software pipelining provides a convenient way of getting optimal resource usage and compact code at the same time. Let us illustrate the idea with our running example.

**Example 10.14:** In Fig. 10.19 is the code from Example 10.12 unrolled five times. (Again we leave out the consideration of register usage.) Shown in row  $i$  are all the operations issued at clock  $i$ ; shown in column  $j$  are all the operations from iteration  $j$ . Note that every iteration has the same schedule relative to its beginning, and also note that every iteration is initiated two clocks after the preceding one. It is easy to see that this schedule satisfies all the resource and data-dependence constraints.

We observe that the operations executed at clocks 7 and 8 are the same as those executed at clocks 9 and 10. Clocks 7 and 8 execute operations from the first four iterations in the original program. Clocks 9 and 10 also execute operations from four iterations, this time from iterations 2 to 5. In fact, we can keep executing this same pair of multi-operation instructions to get the effect of retiring the oldest iteration and adding a new one, until we run out of iterations.

Such dynamic behavior can be encoded succinctly with the code shown in Fig. 10.20, if we assume that the loop has at least 4 iterations. Each row in the figure corresponds to one machine instruction. Lines 7 and 8 form a 2-clock loop, which is executed  $n - 3$  times, where  $n$  is the number of iterations in the original loop.  $\square$

```
L:  LD
    LD
        LD
    MUL LD
        MUL LD
    ADD LD
        ADD LD
    ST   MUL LD
        ST   MUL
            ADD
                ADD
                    ST
                        ST   BL (L)
```

Figure 10.18: Unrolled code for Example 10.12

Clock	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
1	LD				
2	LD				
3	MUL	LD			
4		LD			
5		MUL	LD		
6	ADD		LD		
7			MUL	LD	
8	ST	ADD		LD	
9				MUL	LD
10		ST	ADD		LD
11					MUL
12			ST	ADD	
13					
14				ST	ADD
15					
16					ST

Figure 10.19: Five unrolled iterations of the code in Example 10.12

1)		LD				
2)		LD				
3)		MUL	LD			
4)			LD			
5)			MUL	LD		
6)		ADD		LD		
7)	L:			MUL	LD	
8)		ST	ADD		LD	BL (L)
9)					MUL	
10)			ST	ADD		
11)						
12)				ST	ADD	
13)						
14)					ST	

Figure 10.20: Software-pipelined code for Example 10.12

The technique described above is called *software pipelining*, because it is the software analog of a technique used for scheduling hardware pipelines. We can think of the schedule executed by each iteration in this example as an 8-stage pipeline. A new iteration can be started on the pipeline every 2 clocks. At the beginning, there is only one iteration in the pipeline. As the first iteration proceeds to stage three, the second iteration starts to execute in the first pipeline stage.

By clock 7, the pipeline is fully filled with the first four iterations. In the steady state, four consecutive iterations are executing at the same time. A new iteration is started as the oldest iteration in the pipeline retires. When we run out of iterations, the pipeline drains, and all the iterations in the pipeline run to completion. The sequence of instructions used to fill the pipeline, lines 1 through 6 in our example, is called the *prolog*; lines 7 and 8 are the *steady state*; and the sequence of instructions used to drain the pipeline, lines 9 through 14, is called the *epilog*.

For this example, we know that the loop cannot be run at a rate faster than 2 clocks per iteration, since the machine can only issue one read every clock, and there are two reads in each iteration. The software-pipelined loop above executes in  $2n + 6$  clocks, where  $n$  is the number of iterations in the original loop. As  $n \rightarrow \infty$ , the throughput of the loop approaches the rate of one iteration every two clocks. Thus, software scheduling, unlike unrolling, can potentially encode the optimal schedule with a very compact code sequence.

Note that the schedule adopted for each individual iteration is not the shortest possible. Comparison with the locally optimized schedule shown in Fig. 10.17 shows that a delay is introduced before the ADD operation. The delay is placed strategically so that the schedule can be initiated every two clocks without resource conflicts. Had we stuck with the locally compacted schedule,

the initiation interval would have to be lengthened to 4 clocks to avoid resource conflicts, and the throughput rate would be halved. This example illustrates an important principle in pipeline scheduling: the schedule must be chosen carefully in order to optimize the throughput. A locally compacted schedule, while minimizing the time to complete an iteration, may result in suboptimal throughput when pipelined.

### 10.5.3 Register Allocation and Code Generation

Let us begin by discussing register allocation for the software-pipelined loop in Example 10.14.

**Example 10.15:** In Example 10.14, the result of the multiply operation in the first iteration is produced at clock 3 and used at clock 6. Between these clock cycles, a new result is generated by the multiply operation in the second iteration at clock 5; this value is used at clock 8. The results from these two iterations must be held in different registers to prevent them from interfering with each other. Since interference occurs only between adjacent pairs of iterations, it can be avoided with the use of two registers, one for the odd iterations and one for the even iterations. Since the code for odd iterations is different from that for the even iterations, the size of the steady-state loop is doubled. This code can be used to execute any loop that has an odd number of iterations greater than or equal to 5.

```

if (N >= 5)
    N2 = 3 + 2 * floor((N-3)/2);
else
    N2 = 0;
for (i = 0; i < N2; i++)
    D[i] = A[i]* B[i] + c;
for (i = N2; i < N; i++)
    D[i] = A[i]* B[i] + c;

```

Figure 10.21: Source-level unrolling of the loop from Example 10.12

To handle loops that have fewer than 5 iterations and loops with an even number of iterations, we generate the code whose source-level equivalent is shown in Fig. 10.21. The first loop is pipelined, as seen in the machine-level equivalent of Fig. 10.22. The second loop of Fig. 10.21 need not be optimized, since it can iterate at most four times.  $\square$

### 10.5.4 Do-Across Loops

Software pipelining can also be applied to loops whose iterations share data dependences. Such loops are known as *do-across loops*.

```

1.      LD R5,0(R1++)
2.      LD R6,0(R2++)
3.      LD R5,0(R1++)    MUL R7,R5,R6
4.      LD R6,0(R2++)
5.      LD R5,0(R1++)    MUL R9,R5,R6
6.      LD R6,0(R2++)    ADD R8,R7,R4
7.  L:   LD R5,0(R1++)    MUL R7,R5,R6
8.      LD R6,0(R2++)    ADD R8,R9,R4    ST 0(R3++),R8
9.      LD R5,0(R1++)    MUL R9,R5,R6
10.     LD R6,0(R2++)    ADD R8,R7,R4    ST 0(R3++),R8    BL R10,L
11.                                     MUL R7,R5,R6
12.                                     ADD R8,R9,R4    ST 0(R3++),R8
13.
14.                                     ADD R8,R7,R4    ST 0(R3++),R8
15.
16.                                     ST 0(R3++),R8

```

Figure 10.22: Code after software pipelining and register allocation in Example 10.15

**Example 10.16:** The code

```

for (i = 0; i < n; i++) {
    sum = sum + A[i];
    B[i] = A[i] * b;
}

```

has a data dependence between consecutive iterations, because the previous value of `sum` is added to `A[i]` to create a new value of `sum`. It is possible to execute the summation in  $O(\log n)$  time if the machine can deliver sufficient parallelism, but for the sake of this discussion, we simply assume that all the sequential dependences must be obeyed, and that the additions must be performed in the original sequential order. Because our assumed machine model takes two clocks to complete an `ADD`, the loop cannot execute faster than one iteration every two clocks. Giving the machine more adders or multipliers will not make this loop run any faster. The throughput of do-across loops like this one is limited by the chain of dependences across iterations.

The best locally compacted schedule for each iteration is shown in Fig. 10.23(a), and the software-pipelined code is in Fig. 10.23(b). This software-pipelined loop starts an iteration every two clocks, and thus operates at the optimal rate.  $\square$

```

// R1 = &A; R2 = &B
// R3 = sum
// R4 = b
// R10 = n-1

L:  LD R5, 0(R1++)
    MUL R6, R5, R4
    ADD R3, R3, R4
    ST R6, 0(R2++)          BL R10, L

```

(a) The best locally compacted schedule.

```

// R1 = &A; R2 = &B
// R3 = sum
// R4 = b
// R10 = n-2

LD R5, 0(R1++)
MUL R6, R5, R4
L:  ADD R3, R3, R4          LD R5, 0(R1++)
    ST R6, 0(R2++)          MUL R6, R5, R4  BL R10, L
                                ADD R3, R3, R4
                                ST R6, 0(R2++)

```

(b) The software-pipelined version.

Figure 10.23: Software-pipelining of a do-across loop

### 10.5.5 Goals and Constraints of Software Pipelining

The primary goal of software pipelining is to maximize the throughput of a long-running loop. A secondary goal is to keep the size of the code generated reasonably small. In other words, the software-pipelined loop should have a small steady state of the pipeline. We can achieve a small steady state by requiring that the relative schedule of each iteration be the same, and that the iterations be initiated at a constant interval. Since the throughput of the loop is simply the inverse of the initiation interval, the objective of software pipelining is to minimize this interval.

A software-pipeline schedule for a data-dependence graph  $G = (N, E)$  can be specified by

1. An initiation interval  $T$  and
2. A relative schedule  $S$  that specifies, for each operation, when that operation is executed relative to the start of the iteration to which it belongs.

Thus, an operation  $n$  in the  $i$ th iteration, counting from 0, is executed at clock  $i \times T + S(n)$ . Like all the other scheduling problems, software pipelining has two kinds of constraints: resources and data dependences. We discuss each kind in detail below.

**Modular Resource Reservation**

Let a machine's resources be represented by  $R = [r_1, r_2, \dots]$ , where  $r_i$  is the number of units of the  $i$ th kind of resource available. If an iteration of a loop requires  $n_i$  units of resource  $i$ , then the average initiation interval of a pipelined loop is at least  $\max_i(n_i/r_i)$  clock cycles. Software pipelining requires that the initiation intervals between any pair of iterations have a constant value. Thus, the initiation interval must have at least  $\max_i \lceil n_i/r_i \rceil$  clocks. If  $\max_i(n_i/r_i)$  is less than 1, it is useful to unroll the source code a small number of times.

**Example 10.17:** Let us return to our software-pipelined loop shown in Fig. 10.20. Recall that the target machine can issue one load, one arithmetic operation, one store, and one loop-back branch per clock. Since the loop has two loads, two arithmetic operations, and one store operation, the minimum initiation interval based on resource constraints is 2 clocks.

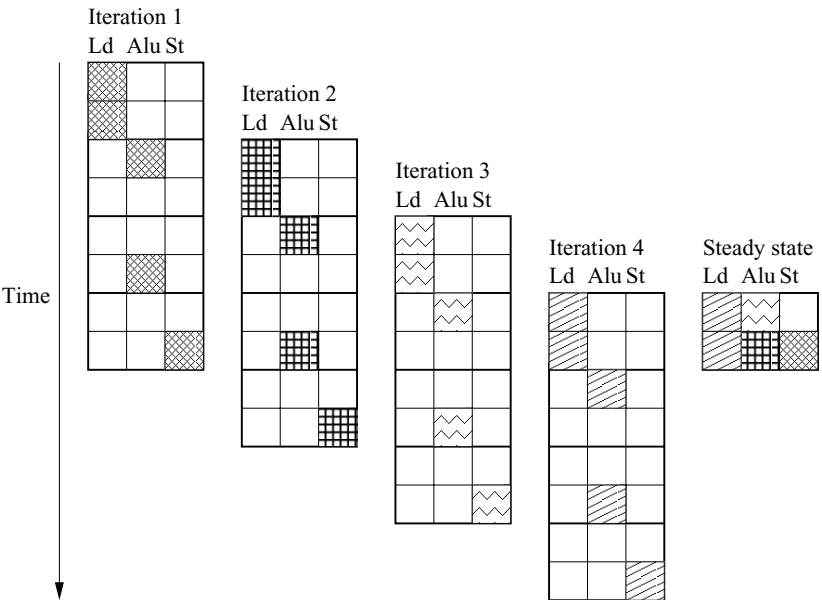


Figure 10.24: Resource requirements of four consecutive iterations from the code in Example 10.13

Figure 10.24 shows the resource requirements of four consecutive iterations across time. More resources are used as more iterations get initiated, culmi-



nating in maximum resource commitment in the steady state. Let  $RT$  be the resource-reservation table representing the commitment of one iteration, and let  $RT_S$  represent the commitment of the steady state.  $RT_S$  combines the commitment from four consecutive iterations started  $T$  clocks apart. The commitment of row 0 in the table  $RT_S$  corresponds to the sum of the resources committed in  $RT[0]$ ,  $RT[2]$ ,  $RT[4]$ , and  $RT[6]$ . Similarly, the commitment of row 1 in the table corresponds to the sum of the resources committed in  $RT[1]$ ,  $RT[3]$ ,  $RT[5]$ , and  $RT[7]$ . That is, the resources committed in the  $i$ th row in the steady state are given by

$$RT_S[i] = \sum_{\{t \mid (t \bmod 2) = i\}} RT[t].$$

We refer to the resource-reservation table representing the steady state as the *modular resource-reservation table* of the pipelined loop.

To check if the software-pipeline schedule has any resource conflicts, we can simply check the commitment of the modular resource-reservation table. Surely, if the commitment in the steady state can be satisfied, so can the commitments in the prolog and epilog, the portions of code before and after the steady-state loop.  $\square$

In general, given an initiation interval  $T$  and a resource-reservation table of an iteration  $RT$ , the pipelined schedule has no resource conflicts on a machine with resource vector  $R$  if and only if  $RT_S[i] \leq R$  for all  $i = 0, 1, \dots, T-1$ .

### Data-Dependence Constraints

Data dependences in software pipelining are different from those we have encountered so far because they can form cycles. An operation may depend on the result of the same operation from a previous iteration. It is no longer adequate to label a dependence edge by just the delay; we also need to distinguish between instances of the same operation in different iterations. We label a dependence edge  $n_1 \rightarrow n_2$  with label  $\langle \delta, d \rangle$  if operation  $n_2$  in iteration  $i$  must be delayed by at least  $d$  clocks after the execution of operation  $n_1$  in iteration  $i - \delta$ . Let  $S$ , a function from the nodes of the data-dependence graph to integers, be the software pipeline schedule, and let  $T$  be the initiation interval target. Then

$$(\delta \times T) + S(n_2) - S(n_1) \geq d.$$

The iteration difference,  $\delta$ , must be nonnegative. Moreover, given a cycle of data-dependence edges, at least one of the edges has a positive iteration difference.

**Example 10.18:** Consider the following loop, and suppose we do not know the values of  $p$  and  $q$ :

```
for (i = 0; i < n; i++)
    *(p++) = *(q++) + c;
```

We must assume that any pair of  $*(p++)$  and  $*(q++)$  accesses may refer to the same memory location. Thus, all the reads and writes must execute in the original sequential order. Assuming that the target machine has the same characteristics as that described in Example 10.12, the data-dependence edges for this code are as shown in Fig. 10.25. Note, however, that we ignore the loop-control instructions that would have to be present, either computing and testing  $i$ , or doing the test based on the value of R1 or R2.  $\square$

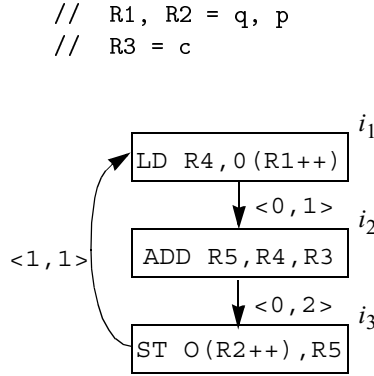


Figure 10.25: Data-dependence graph for Example 10.18

The iteration difference between related operations can be greater than one, as shown in the following example:

```
for (i = 2; i < n; i++)
    A[i] = B[i] + A[i-2];
```

Here the value written in iteration  $i$  is used two iterations later. The dependence edge between the store of  $A[i]$  and the load of  $A[i-2]$  thus has a difference of 2 iterations.

The presence of data-dependence cycles in a loop imposes yet another limit on its execution throughput. For example, the data-dependence cycle in Fig. 10.25 imposes a delay of 4 clock ticks between load operations from consecutive iterations. That is, loops cannot execute at a rate faster than one iteration every 4 clocks.

The initiation interval of a pipelined loop is no smaller than

$$\max_{c \text{ a cycle in } G} \left\lceil \frac{\sum_{e \text{ in } c} d_e}{\sum_{e \text{ in } c} \delta_e} \right\rceil$$

clocks.

In summary, the initiation interval of each software-pipelined loop is bounded by the resource usage in each iteration. Namely, the initiation interval must be no smaller than the ratio of units needed of each resource and the units

available on the machine. In addition, if the loops have data-dependence cycles, then the initiation interval is further constrained by the sum of the delays in the cycle divided by the sum of the iteration differences. The largest of these quantities defines a lower bound on the initiation interval.

### 10.5.6 A Software-Pipelining Algorithm

The goal of software pipelining is to find a schedule with the smallest possible initiation interval. The problem is NP-complete, and can be formulated as an integer-linear-programming problem. We have shown that if we know what the minimum initiation interval is, the scheduling algorithm can avoid resource conflicts by using the modular resource-reservation table in placing each operation. But we do not know what the minimum initiation interval is until we can find a schedule. How do we resolve this circularity?

We know that the initiation interval must be greater than the bound computed from a loop's resource requirement and dependence cycles as discussed above. If we can find a schedule meeting this bound, we have found the optimal schedule. If we fail to find such a schedule, we can try again with larger initiation intervals until a schedule is found. Note that if heuristics, rather than exhaustive search, are used, this process may not find the optimal schedule.

Whether we can find a schedule near the lower bound depends on properties of the data-dependence graph and the architecture of the target machine. We can easily find the optimal schedule if the dependence graph is acyclic and if every machine instruction needs only one unit of one resource. It is also easy to find a schedule close to the lower bound if there are more hardware resources than can be used by graphs with dependence cycles. For such cases, it is advisable to start with the lower bound as the initial initiation-interval target, then keep increasing the target by just one clock with each scheduling attempt. Another possibility is to find the initiation interval using a binary search. We can use as an upper bound on the initiation interval the length of the schedule for one iteration produced by list scheduling.

### 10.5.7 Scheduling Acyclic Data-Dependence Graphs

For simplicity, we assume for now that the loop to be software pipelined contains only one basic block. This assumption will be relaxed in Section 10.5.11.

**Algorithm 10.19:** Software pipelining an acyclic dependence graph.

**INPUT:** A machine-resource vector  $R = [r_1, r_2, \dots]$ , where  $r_i$  is the number of units available of the  $i$ th kind of resource, and a data-dependence graph  $G = (N, E)$ . Each operation  $n$  in  $N$  is labeled with its resource-reservation table  $RT_n$ ; each edge  $e = n_1 \rightarrow n_2$  in  $E$  is labeled with  $\langle \delta_e, d_e \rangle$  indicating that  $n_2$  must execute no earlier than  $d_e$  clocks after node  $n_1$  from the  $\delta_e$ th preceding iteration.

**OUTPUT:** A software-pipelined schedule  $S$  and an initiation interval  $T$ .

**METHOD:** Execute the program in Fig. 10.26.  $\square$

```

main() {
     $T_0 = \max_j \left\lceil \frac{\sum_{n,i} RT_n(i, j)}{r_j} \right\rceil$ ;
    for ( $T = T_0, T_0 + 1, \dots$ , until all nodes in  $N$  are scheduled) {
         $RT$  = an empty reservation table with  $T$  rows;
        for (each  $n$  in  $N$  in prioritized topological order) {
             $s_0 = \max_{e=p \rightarrow n \text{ in } E} (S(p) + d_e)$ ;
            for ( $s = s_0, s_0 + 1, \dots, s_0 + T - 1$ )
                if ( $NodeScheduled(RT, T, n, s)$ ) break;
            if ( $n$  cannot be scheduled in  $RT$ ) break;
        }
    }
}

NodeScheduled( $RT, T, n, s$ ) {
     $RT' = RT$ ;
    for (each row  $i$  in  $RT_n$ )
         $RT'[(s + i) \bmod T] = RT'[(s + i) \bmod T] + RT_n[i]$ ;
    if (for all  $i$ ,  $RT'(i) \leq R$ ) {
         $RT = RT'$ ;
         $S(n) = s$ ;
        return true;
    }
    else return false;
}

```

Figure 10.26: Software-pipelining algorithm for acyclic graphs

Algorithm 10.19 software pipelines acyclic data-dependence graphs. The algorithm first finds a bound on the initiation interval,  $T_0$ , based on the resource requirements of the operations in the graph. It then attempts to find a software-pipelined schedule starting with  $T_0$  as the target initiation interval. The algorithm repeats with increasingly larger initiation intervals if it fails to find a schedule.

The algorithm uses a list-scheduling approach in each attempt. It uses a modular resource-reservation  $RT$  to keep track of the resource commitment in the steady state. Operations are scheduled in topological order so that the data dependences can always be satisfied by delaying operations. To schedule an operation, it first finds a lower bound  $s_0$  according to the data-dependence constraints. It then invokes *NodeScheduled* to check for possible resource conflicts in the steady state. If there is a resource conflict, the algorithm tries to schedule the operation in the next clock. If the operation is found to conflict for

$T$  consecutive clocks, because of the modular nature of resource-conflict detection, further attempts are guaranteed to be futile. At that point, the algorithm considers the attempt a failure, and another initiation interval is tried.

The heuristics of scheduling operations as soon as possible tends to minimize the length of the schedule for an iteration. Scheduling an instruction as early as possible, however, can lengthen the lifetimes of some variables. For example, loads of data tend to be scheduled early, sometimes long before they are used. One simple heuristic is to schedule the dependence graph backwards because there are usually more loads than stores.

### 10.5.8 Scheduling Cyclic Dependence Graphs

Dependence cycles complicate software pipelining significantly. When scheduling operations in an acyclic graph in topological order, data dependences with scheduled operations can impose only a lower bound on the placement of each operation. As a result, it is always possible to satisfy the data-dependence constraints by delaying operations. The concept of “topological order” does not apply to cyclic graphs. In fact, given a pair of operations sharing a cycle, placing one operation will impose both a lower and upper bound on the placement of the second.

Let  $n_1$  and  $n_2$  be two operations in a dependence cycle,  $S$  be a software-pipeline schedule, and  $T$  be the initiation interval for the schedule. A dependence edge  $n_1 \rightarrow n_2$  with label  $\langle \delta_1, d_1 \rangle$  imposes the following constraint on  $S(n_1)$  and  $S(n_2)$ :

$$(\delta_1 \times T) + S(n_2) - S(n_1) \geq d_1.$$

Similarly, a dependence edge  $n_1 \rightarrow n_2$  with label  $\langle \delta_2, d_2 \rangle$  imposes constraint

$$(\delta_2 \times T) + S(n_1) - S(n_2) \geq d_2.$$

Thus,

$$S(n_1) + d_1 - (\delta_1 \times T) \leq S(n_2) \leq S(n_1) - d_2 + (\delta_2 \times T).$$

A *strongly connected component* (SCC) in a graph is a set of nodes where every node in the component can be reached by every other node in the component. Scheduling one node in an SCC will bound the time of every other node in the component both from above and from below. Transitively, if there exists a path  $p$  leading from  $n_1$  to  $n_2$ , then

$$S(n_2) - S(n_1) \geq \sum_{e \text{ in } p} (d_e - (\delta_e \times T)) \quad (10.1)$$

Observe that

- Around any cycle, the sum of the  $\delta$ 's must be positive. If it were 0 or negative, then it would say that an operation in the cycle either had to precede itself or be executed at the same clock for all iterations.
- The schedule of operations within an iteration is the same for all iterations; that requirement is essentially the meaning of a “software pipeline.” As a result, the sum of the delays (second components of edge labels in a data-dependence graph) around a cycle is a lower bound on the initiation interval  $T$ .

From these two points, if path  $p$  is a cycle, then for any feasible initiation interval  $T$ , the value of the right side of Equation (10.1) is negative or zero. As a result, the strongest constraints on the placement of nodes is obtained from the *simple* paths — those paths that contain no cycles.

Thus, for each feasible  $T$ , computing the transitive effect of data dependences on each pair of nodes is equivalent to finding the length of the longest simple path from the first node to the second. Moreover, since cycles cannot increase the length of a path, we can use a simple dynamic-programming algorithm to find the longest paths without the “simple-path” requirement, and be sure that the resulting lengths will also be the lengths of the longest simple paths (see Exercise 10.5.7).

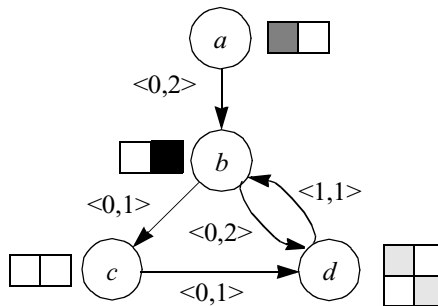


Figure 10.27: Dependence graph and resource requirement in Example 10.20

**Example 10.20:** Figure 10.27 shows a data-dependence graph with four nodes  $a, b, c, d$ . Attached to each node is its resource-reservation table; attached to each edge is its iteration difference and delay. Assume for this example that the target machine has one unit of each kind of resource. Since there are three uses of the first resource and two of the second, the initiation interval must be no less than 3 clocks. There are two SCC's in this graph: the first is a trivial component consisting of the node  $a$  alone, and the second consists of nodes  $b, c$ , and  $d$ . The longest cycle,  $b, c, d, b$ , has a total delay of 3 clocks connecting nodes that are 1 iteration apart. Thus, the lower bound on the initiation interval provided by data-dependence cycle constraints is also 3 clocks.

Placing any of  $b$ ,  $c$ , or  $d$  in a schedule constrains all the other nodes in the component. Let  $T$  be the initiation interval. Figure 10.28 shows the transitive dependences. Part (a) shows the delay and the iteration difference  $\delta$ , for each edge. The delay is represented directly, but  $\delta$  is represented by “adding” to the delay the value  $-\delta T$ .

Figure 10.28(b) shows the length of the longest simple path between two nodes, when such a path exists; its entries are the sums of the expressions given by Fig. 10.28(a), for each edge along the path. Then, in (c) and (d), we see the expressions of (b) with the two relevant values of  $T$ , that is, 3 and 4, substituted for  $T$ . The difference between the schedule of two nodes  $S(n_2) - S(n_1)$  must be no less than the value given in entry  $(n_1, n_2)$  in each of the tables (c) or (d), depending on the value of  $T$  chosen.

For instance, consider the entry in Fig. 10.28 for the longest (simple) path from  $c$  to  $b$ , which is  $2 - T$ . The longest simple path from  $c$  to  $b$  is  $c \rightarrow d \rightarrow b$ . The total delay is 2 along this path, and the sum of the  $\delta$ 's is 1, representing the fact that the iteration number must increase by 1. Since  $T$  is the time by which each iteration follows the previous, the clock at which  $b$  must be scheduled is at least  $2 - T$  clocks *after* the clock at which  $c$  is scheduled. Since  $T$  is at least 3, we are really saying that  $b$  may be scheduled  $T - 2$  clocks *before*  $c$ , or later than that clock, but not earlier.

Notice that considering nonsimple paths from  $c$  to  $b$  does not produce a stronger constraint. We can add to the path  $c \rightarrow d \rightarrow b$  any number of iterations of the cycle involving  $d$  and  $b$ . If we add  $k$  such cycles, we get a path length of  $2 - T + k(3 - T)$ , since the total delay along the path is 3, and the sum of the  $\delta$ 's is 1. Since  $T \geq 3$ , this length can never exceed  $2 - T$ ; i.e., the strongest lower bound on the clock of  $b$  relative to the clock of  $c$  is  $2 - T$ , the bound we get by considering the longest simple path.

For example, from entries  $(b, c)$  and  $(c, b)$ , we see that

$$\begin{aligned} S(c) - S(b) &\geq 1 \\ S(b) - S(c) &\geq 2 - T. \end{aligned}$$

That is,

$$S(b) + 1 \leq S(c) \leq S(b) - 2 + T.$$

If  $T = 3$ ,

$$S(b) + 1 \leq S(c) \leq S(b) + 1.$$

Put equivalently,  $c$  must be scheduled one clock after  $b$ . If  $T = 4$ , however,

$$S(b) + 1 \leq S(c) \leq S(b) + 2.$$

That is,  $c$  is scheduled one or two clocks after  $b$ .

Given the all-points longest path information, we can easily compute the range where it is legal to place a node due to data dependences. We see that there is no slack in the case when  $T = 3$ , and the slack increases as  $T$  increases.

□

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>		2		
<i>b</i>			1	2
<i>c</i>				1
<i>d</i>		1- <i>T</i>		

(a) Original edges.

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>		2	3	4
<i>b</i>			1	2
<i>c</i>		2- <i>T</i>		1
<i>d</i>		1- <i>T</i>	2- <i>T</i>	

(b) Longest simple paths.

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>		2	3	4
<i>b</i>			1	2
<i>c</i>		-1		1
<i>d</i>		-2	-1	

(c) Longest simple paths ( $T=3$ ).

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>		2	3	4
<i>b</i>			1	2
<i>c</i>		-2		1
<i>d</i>		-3	-2	

(d) Longest simple paths ( $T=4$ ).

Figure 10.28: Transitive dependences in Example 10.20

**Algorithm 10.21:** Software pipelining.

**INPUT:** A machine-resource vector  $R = [r_1, r_2, \dots]$ , where  $r_i$  is the number of units available of the  $i$ th kind of resource, and a data-dependence graph  $G = (N, E)$ . Each operation  $n$  in  $N$  is labeled with its resource-reservation table  $RT_n$ ; each edge  $e = n_1 \rightarrow n_2$  in  $E$  is labeled with  $\langle \delta_e, d_e \rangle$  indicating that  $n_2$  must execute no earlier than  $d_e$  clocks after node  $n_1$  from the  $\delta_e$ th preceding iteration.

**OUTPUT:** A software-pipelined schedule  $S$  and an initiation interval  $T$ .

**METHOD:** Execute the program in Fig. 10.29.  $\square$

Algorithm 10.21 has a high-level structure similar to that of Algorithm 10.19, which only handles acyclic graphs. The minimum initiation interval in this case is bounded not just by resource requirements, but also by the data-dependence cycles in the graph. The graph is scheduled one strongly connected component at a time. By treating each strongly connected component as a unit, edges between strongly connected components necessarily form an acyclic graph. While the top-level loop in Algorithm 10.19 schedules nodes in the graph in topological order, the top-level loop in Algorithm 10.21 schedules strongly connected components in topological order. As before, if the algorithm fails to schedule all the components, then a larger initiation interval is tried. Note that Algorithm 10.21 behaves exactly like Algorithm 10.19 if given an acyclic data-dependence graph.

Algorithm 10.21 computes two more sets of edges:  $E'$  is the set of all edges whose iteration difference is 0,  $E^*$  is the all-points longest-path edges. That is,



```

main() {
   $E' = \{e | e \text{ in } E, \delta_e = 0\};$ 
   $T_0 = \max \left( \max_j \left\lceil \frac{\sum_{n,i} RT_n(i,j)}{r_j} \right\rceil, \max_{c \text{ a cycle in } G} \left\lceil \frac{\sum_{e \text{ in } c} d_e}{\sum_{e \text{ in } c} \delta_e} \right\rceil \right);$ 
  for ( $T = T_0, T_0 + 1, \dots$  or until all SCC's in  $G$  are scheduled) {
     $RT$  = an empty reservation table with  $T$  rows;
     $E^* = \text{AllPairsLongestPath}(G, T);$ 
    for (each SCC  $C$  in  $G$  in prioritized topological order) {
      for (all  $n$  in  $C$ )
         $s_0(n) = \max_{e=p \rightarrow n \text{ in } E^*, p \text{ scheduled}} (S(p) + d_e);$ 
       $first$  = some  $n$  such that  $s_0(n)$  is a minimum;
       $s_0 = s_0(first);$ 
      for ( $s = s_0; s < s_0 + T; s = s + 1$ )
        if ( $\text{SccScheduled}(RT, T, C, first, s)$ ) break;
      if ( $C$  cannot be scheduled in  $RT$ ) break;
    }
  }
}

SccScheduled( $RT, T, c, first, s$ ) {
   $RT' = RT;$ 
  if (not  $\text{NodeScheduled}(RT', T, first, s)$ ) return false;
  for (each remaining  $n$  in  $c$  in prioritized
    topological order of edges in  $E'$ ) {
     $s_l = \max_{e=n' \rightarrow n \text{ in } E^*, n' \text{ in } c, n' \text{ scheduled}} (S(n') + d_e - (\delta_e \times T));$ 
     $s_u = \min_{e=n \rightarrow n' \text{ in } E^*, n' \text{ in } c, n' \text{ scheduled}} (S(n') - d_e + (\delta_e \times T));$ 
    for ( $s = s_l; s \leq \min(s_u, s_l + T - 1); s = s + 1$ )
      if ( $\text{NodeScheduled}(RT', T, n, s)$ ) break;
    if ( $n$  cannot be scheduled in  $RT'$ ) return false;
  }
   $RT = RT';$ 
  return true;
}

```

Figure 10.29: A software-pipelining algorithm for cyclic dependence graphs

for each pair of nodes  $(p, n)$ , there is an edge  $e$  in  $E^*$  whose associated distance  $d_e$  is the length of the longest simple path from  $p$  to  $n$ , provided that there is at least one path from  $p$  to  $n$ .  $E^*$  is computed for each value of  $T$ , the initiation-interval target. It is also possible to perform this computation just once with a symbolic value of  $T$  and then substitute for  $T$  in each iteration, as we did in Example 10.20.

Algorithm 10.21 uses backtracking. If it fails to schedule a SCC, it tries to reschedule the entire SCC a clock later. These scheduling attempts continue for up to  $T$  clocks. Backtracking is important because, as shown in Example 10.20, the placement of the first node in an SCC can fully dictate the schedule of all other nodes. If the schedule happens not to fit with the schedule created thus far, the attempt fails.

To schedule a SCC, the algorithm determines the earliest time each node in the component can be scheduled satisfying the transitive data dependences in  $E^*$ . It then picks the one with the earliest start time as the *first* node to schedule. The algorithm then invokes *SccScheduled* to try to schedule the component at the earliest start time. The algorithm makes at most  $T$  attempts with successively greater start times. If it fails, then the algorithm tries another initiation interval.

The *SccScheduled* algorithm resembles Algorithm 10.19, but has three major differences.

1. The goal of *SccScheduled* is to schedule the strongly connected component at the given time slot  $s$ . If the *first* node of the strongly connected component cannot be scheduled at  $s$ , *SccScheduled* returns false. The *main* function can invoke *SccScheduled* again with a later time slot if that is desired.
2. The nodes in the strongly connected component are scheduled in topological order, based on the edges in  $E'$ . Because the iteration differences on all the edges in  $E'$  are 0, these edges do not cross any iteration boundaries and cannot form cycles. (Edges that cross iteration boundaries are known as *loop carried*). Only loop-carried dependences place upper bounds on where operations can be scheduled. So, this scheduling order, along with the strategy of scheduling each operation as early as possible, maximizes the ranges in which subsequent nodes can be scheduled.
3. For strongly connected components, dependences impose both a lower and upper bound on the range in which a node can be scheduled. *SccScheduled* computes these ranges and uses them to further limit the scheduling attempts.

**Example 10.22:** Let us apply Algorithm 10.21 to the cyclic data-dependence graph in Example 10.20. The algorithm first computes that the bound on the initiation interval for this example is 3 clocks. We note that it is not possible to meet this lower bound. When the initiation interval  $T$  is 3, the transitive

dependencies in Fig. 10.28 dictate that  $S(d) - S(b) = 2$ . Scheduling nodes  $b$  and  $d$  two clocks apart will produce a conflict in a modular resource-reservation table of length 3.







Attempt	Initiation Interval	Node	Range	Schedule	Modular Resource Reservation
1	$T = 3$	$a$	$(0, \infty)$	0	
		$b$	$(2, \infty)$	2	
		$c$	$(3, 3)$	--	
2	$T = 3$	$a$	$(0, \infty)$	0	
		$b$	$(2, \infty)$	3	
		$c$	$(4, 4)$	4	
		$d$	$(5, 5)$	--	
3	$T = 3$	$a$	$(0, \infty)$	0	
		$b$	$(2, \infty)$	4	
		$c$	$(5, 5)$	5	
		$d$	$(6, 6)$	--	
4	$T = 4$	$a$	$(0, \infty)$	0	
		$b$	$(2, \infty)$	2	
		$c$	$(3, 4)$	3	
		$d$	$(4, 5)$	--	
5	$T = 4$	$a$	$(0, \infty)$	0	
		$b$	$(2, \infty)$	3	
		$c$	$(4, 5)$	5	
		$d$	$(5, 5)$	--	
6	$T = 4$	$a$	$(0, \infty)$	0	
		$b$	$(2, \infty)$	4	
		$c$	$(5, 6)$	5	
		$d$	$(6, 7)$	6	

Figure 10.30: Behavior of Algorithm 10.21 on Example 10.20

Figure 10.30 shows how Algorithm 10.21 behaves with this example. It first tries to find a schedule with a 3-clock initiation interval. The attempt starts by scheduling nodes  $a$  and  $b$  as early as possible. However, once node  $b$  is placed in clock 2, node  $c$  can only be placed at clock 3, which conflicts with the resource usage of node  $a$ . That is,  $a$  and  $c$  both need the first resource at clocks that have a remainder of 0 modulo 3.

The algorithm backtracks and tries to schedule the strongly connected component  $\{b, c, d\}$  a clock later. This time node  $b$  is scheduled at clock 3, and node  $c$  is scheduled successfully at clock 4. Node  $d$ , however, cannot be scheduled in

clock 5. That is, both  $b$  and  $d$  need the second resource at clocks that have a remainder of 0 modulo 3. Note that it is just a coincidence that the two conflicts discovered so far are at clocks with a remainder of 0 modulo 3; the conflict might have occurred at clocks with remainder 1 or 2 in another example.

The algorithm repeats by delaying the start of the SCC  $\{b, c, d\}$  by one more clock. But, as discussed earlier, this SCC can never be scheduled with an initiation interval of 3 clocks, so the attempt is bound to fail. At this point, the algorithm gives up and tries to find a schedule with an initiation interval of 4 clocks. The algorithm eventually finds the optimal schedule on its sixth attempt.  $\square$

### 10.5.9 Improvements to the Pipelining Algorithms

Algorithm 10.21 is a rather simple algorithm, although it has been found to work well on actual machine targets. The important elements in this algorithm are

1. The use of a modular resource-reservation table to check for resource conflicts in the steady state.
2. The need to compute the transitive dependence relations to find the legal range in which a node can be scheduled in the presence of dependence cycles.
3. Backtracking is useful, and nodes on *critical cycles* (cycles that place the highest lower bound on the initiation interval  $T$ ) must be rescheduled together because there is no slack between them.

There are many ways to improve Algorithm 10.21. For instance, the algorithm takes a while to realize that a 3-clock initiation interval is infeasible for the simple Example 10.22. We can schedule the strongly connected components independently first to determine if the initiation interval is feasible for each component.

We can also modify the order in which the nodes are scheduled. The order used in Algorithm 10.21 has a few disadvantages. First, because nontrivial SCC's are harder to schedule, it is desirable to schedule them first. Second, some of the registers may have unnecessarily long lifetimes. It is desirable to pull the definitions closer to the uses. One possibility is to start with scheduling strongly connected components with critical cycles first, then extend the schedule on both ends.

### 10.5.10 Modular Variable Expansion

A scalar variable is said to be *privatizable* in a loop if its live range falls within an iteration of the loop. In other words, a privatizable variable must not be live upon either entry or exit of any iteration. These variables are so named because

### Are There Alternatives to Heuristics?

We can formulate the problem of simultaneously finding an optimal software pipeline schedule and register assignment as an integer-linear-programming problem. While many integer linear programs can be solved quickly, some of them can take an exorbitant amount of time. To use an integer-linear-programming solver in a compiler, we must be able to abort the procedure if it does not complete within some preset limit.

Such an approach has been tried on a target machine (the SGI R8000) empirically, and it was found that the solver could find the optimal solution for a large percentage of the programs in the experiment within a reasonable amount of time. It turned out that the schedules produced using a heuristic approach were also close to optimal. The results suggest that, at least for that machine, it does not make sense to use the integer-linear-programming approach, especially from a software engineering perspective. Because the integer-linear solver may not finish, it is still necessary to implement some kind of a heuristic scheduler in the compiler. Once such a heuristic scheduler is in place, there is little incentive to implement a scheduler based on integer programming techniques as well.

different processors executing different iterations in a loop can have their own private copies and thus not interfere with one another.

*Variable expansion* refers to the transformation of converting a privatizable scalar variable into an array and having the  $i$ th iteration of the loop read and write the  $i$ th element. This transformation eliminates the antidependence constraints between reads in one iteration and writes in the subsequent iterations, as well as output dependences between writes from different iterations. If all loop-carried dependences can be eliminated, all the iterations in the loop can be executed in parallel.

Eliminating loop-carried dependences, and thus eliminating cycles in the data-dependence graph, can greatly improve the effectiveness of software pipelining. As illustrated by Example 10.15, we need not expand a privatizable variable fully by the number of iterations in the loop. Only a small number of iterations can be executing at a time, and privatizable variables may simultaneously be live in an even smaller number of iterations. The same storage can thus be reused to hold variables with nonoverlapping lifetimes. More specifically, if the lifetime of a register is  $l$  clocks, and the initiation interval is  $T$ , then only  $q = \lceil \frac{l}{T} \rceil$  values can be live at any one point. We can allocate  $q$  registers to the variable, with the variable in the  $i$ th iteration using the  $(i \bmod q)$ th register. We refer to this transformation as *modular variable expansion*.

**Algorithm 10.23:** Software pipelining with modular variable expansion.

**INPUT:** A data-dependence graph and a machine-resource description.

**OUTPUT:** Two loops, one software pipelined and one unpipelined.

**METHOD:**

1. Remove the loop-carried antidependences and output dependences associated with privatizable variables from the data-dependence graph.
2. Software-pipeline the resulting dependence graph using Algorithm 10.21. Let  $T$  be the initiation interval for which a schedule is found, and  $L$  be the length of the schedule for one iteration.
3. From the resulting schedule, compute  $q_v$ , the minimum number of registers needed by each privatizable variable  $v$ . Let  $Q = \max_v q_v$ .
4. Generate two loops: a software-pipelined loop and an unpipelined loop. The software-pipelined loop has

$$\left\lceil \frac{L}{T} \right\rceil + Q - 1$$

copies of the iterations, placed  $T$  clocks apart. It has a prolog with

$$\left( \left\lceil \frac{L}{T} \right\rceil - 1 \right) T$$

instructions, a steady state with  $QT$  instructions, and an epilog of  $L - T$  instructions. Insert a loop-back instruction that branches from the bottom of the steady state to the top of the steady state.

The number of registers assigned to privatizable variable  $v$  is

$$q'_v = \begin{cases} q_v & \text{if } Q \bmod q_v = 0 \\ Q & \text{otherwise} \end{cases}$$

The variable  $v$  in iteration  $i$  uses the  $(i \bmod q'_v)$ th register assigned.

Let  $n$  be the variable representing the number of iterations in the source loop. The software-pipelined loop is executed if

$$n \geq \left\lceil \frac{L}{T} \right\rceil + Q - 1.$$

The number of times the loop-back branch is taken is

$$n_1 = \left\lfloor \frac{n - \left\lceil \frac{L}{T} \right\rceil + 1}{Q} \right\rfloor.$$

Thus, the number of source iterations executed by the software-pipelined loop is

$$n_2 = \begin{cases} \left\lceil \frac{L}{T} \right\rceil - 1 + Qn_1 & \text{if } n \geq \left\lceil \frac{L}{T} \right\rceil + Q - 1 \\ 0 & \text{otherwise} \end{cases}$$

The number of iterations executed by the unpipelined loop is  $n_3 = n - n_2$ .

□

**Example 10.24:** For the software-pipelined loop in Fig. 10.22,  $L = 8$ ,  $T = 2$ , and  $Q = 2$ . The software-pipelined loop has 7 copies of the iterations, with the prolog, steady state, and epilog having 6, 4, and 6 instructions, respectively. Let  $n$  be the number of iterations in the source loop. The software-pipelined loop is executed if  $n \geq 5$ , in which case the loop-back branch is taken

$$\left\lfloor \frac{n-3}{2} \right\rfloor$$

times, and the software-pipelined loop is responsible for

$$3 + 2 \times \left\lfloor \frac{n-3}{2} \right\rfloor$$

of the iterations in the source loop. □

Modular expansion increases the size of the steady state by a factor of  $Q$ . Despite this increase, the code generated by Algorithm 10.23 is still fairly compact. In the worst case, the software-pipelined loop would take three times as many instructions as that of the schedule for one iteration. Roughly, together with the extra loop generated to handle the left-over iterations, the total code size is about four times the original. This technique is usually applied to tight inner loops, so this increase is reasonable.

Algorithm 10.23 minimizes code expansion at the expense of using more registers. We can reduce register usage by generating more code. We can use the minimum  $q_v$  registers for each variable  $v$  if we use a steady state with

$$T \times \text{LCM}_v q_v$$

instructions. Here,  $\text{LCM}_v$  represents the operation of taking the *least common multiple* of all the  $q_v$ 's, as  $v$  ranges over all the privatizable variables (i.e., the smallest integer that is an integer multiple of all the  $q_v$ 's). Unfortunately, the least common multiple can be quite large even for a few small  $q_v$ 's.

### 10.5.11 Conditional Statements

If predicated instructions are available, we can convert control-dependent instructions into predicated ones. Predicated instructions can be software-pipelined like any other operations. However, if there is a large amount of data-dependent control flow within the loop body, scheduling techniques described in Section 10.4 may be more appropriate.

If a machine does not have predicated instructions, we can use the concept of *hierarchical reduction*, described below, to handle a small amount of data-dependent control flow. Like Algorithm 10.11, in hierarchical reduction the control constructs in the loop are scheduled inside-out, starting with the most

deeply nested structures. As each construct is scheduled, the entire construct is reduced to a single node representing all the scheduling constraints of its components with respect to the other parts of the program. This node can then be scheduled as if it were a simple node within the surrounding control construct. The scheduling process is complete when the entire program is reduced to a single node.

In the case of a conditional statement with “then” and “else” branches, we schedule each of the branches independently. Then:

1. The constraints of the entire conditional statement are conservatively taken to be the union of the constraints from both branches.
2. Its resource usage is the maximum of the resources used in each branch.
3. Its precedence constraints are the union of those in each branch, obtained by pretending that both branches are executed.

This node can then be scheduled like any other node. Two sets of code, corresponding to the two branches, are generated. Any code scheduled in parallel with the conditional statement is duplicated in both branches. If multiple conditional statements are overlapped, separate code must be generated for each combination of branches executed in parallel.

### 10.5.12 Hardware Support for Software Pipelining

Specialized hardware support has been proposed for minimizing the size of software-pipelined code. The *rotating register file* in the Itanium architecture is one such example. A rotating register file has a *base register*, which is added to the register number specified in the code to derive the actual register accessed. We can get different iterations in a loop to use different registers simply by changing the contents of the base register at the boundary of each iteration. The Itanium architecture also has extensive predicated instruction support. Not only can predication be used to convert control dependence to data dependence but it also can be used to avoid generating the prologs and epilogs. The body of a software-pipelined loop contains a superset of the instructions issued in the prolog and epilog. We can simply generate the code for the steady state and use predication appropriately to suppress the extra operations to get the effects of having a prolog and an epilog.

While Itanium’s hardware support improves the density of software-pipelined code, we must also realize that the support is not cheap. Since software pipelining is a technique intended for tight innermost loops, pipelined loops tend to be small anyway. Specialized support for software pipelining is warranted principally for machines that are intended to execute many software-pipelined loops and in situations where it is very important to minimize code size.



```

1)  L:  LD   R1, a(R9)
2)      ST   b(R9), R1
3)      LD   R2, c(R9)
4)      ADD  R3, R1, R2
5)      ST   c(R9), R3
6)      SUB  R4, R1, R2
7)      ST   b(R9), R4
8)      BL  R9, L

```

Figure 10.31: Machine code for Exercise 10.5.2

### 10.5.13 Exercises for Section 10.5

**Exercise 10.5.1:** In Example 10.20 we showed how to establish the bounds on the relative clocks at which  $b$  and  $c$  are scheduled. Compute the bounds for each of five other pairs of nodes (*i*) for general  $T$  (*ii*) for  $T = 3$  (*iii*) for  $T = 4$ .

**Exercise 10.5.2:** In Fig. 10.31 is the body of a loop. Addresses such as  $a(R9)$  are intended to be memory locations, where  $a$  is a constant, and  $R9$  is the register that counts iterations through the loop. You may assume that each iteration of the loop accesses different locations, because  $R9$  has a different value. Using the machine model of Example 10.12, schedule the loop of Fig. 10.31 in the following ways:

- a) Keeping each iteration as tight as possible (i.e., only introduce one **nop** after each arithmetic operation), unroll the loop twice. Schedule the second iteration to commence at the earliest possible moment without violating the constraint that the machine can only do one load, one store, one arithmetic operation, and one branch at any clock.
- b) Repeat part (a), but unroll the loop three times. Again, start each iteration as soon as you can, subject to the machine constraints.
- ! c) Construct fully pipelined code subject to the machine constraints. In this part, you can introduce extra **nop**'s if needed, but you must start a new iteration every two clock ticks.

**Exercise 10.5.3:** A certain loop requires 5 loads, 7 stores, and 8 arithmetic operations. What is the minimum initiation interval for a software pipelining of this loop on a machine that executes each operation in one clock tick, and has resources enough to do, in one clock tick:

- a) 3 loads, 4 stores, and 5 arithmetic operations.
- b) 3 loads, 3 stores, and 3 arithmetic operations.

**! Exercise 10.5.4:** Using the machine model of Example 10.12, find the minimum initiation interval and a uniform schedule for the iterations, for the following loop:

```

for (i = 1; i < n; i++) {
    A[i] = B[i-1] + 1;
    B[i] = A[i-1] + 2;
}

```

Remember that the counting of iterations is handled by auto-increment of registers, and no operations are needed solely for the counting associated with the for-loop.

**! Exercise 10.5.5:** Prove that Algorithm 10.19, in the special case where every operation requires only one unit of one resource, can always find a software-pipeline schedule meeting the lower bound.

**! Exercise 10.5.6:** Suppose we have a cyclic data-dependence graph with nodes  $a$ ,  $b$ ,  $c$ , and  $d$ . There are edges from  $a$  to  $b$  and from  $c$  to  $d$  with label  $\langle 0, 1 \rangle$  and there are edges from  $b$  to  $c$  and from  $d$  to  $a$  with label  $\langle 1, 1 \rangle$ . There are no other edges.

- a) Draw the cyclic dependence graph.
- b) Compute the table of longest simple paths among the nodes.
- c) Show the lengths of the longest simple paths if the initiation interval  $T$  is 2.
- d) Repeat (c) if  $T = 3$ .
- e) For  $T = 3$ , what are the constraints on the relative times that each of the instructions represented by  $a$ ,  $b$ ,  $c$ , and  $d$  may be scheduled?

**! Exercise 10.5.7:** Give an  $O(n^3)$  algorithm to find the length of the longest simple path in an  $n$ -node graph, on the assumption that no cycle has a positive length. *Hint:* Adapt Floyd's algorithm for shortest paths (see, e.g., A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, 1992).

**!! Exercise 10.5.8:** Suppose we have a machine with three instruction types, which we'll call  $A$ ,  $B$ , and  $C$ . All instructions require one clock tick, and the machine can execute one instruction of each type at each clock. Suppose a loop consists of six instructions, two of each type. Then it is possible to execute the loop in a software pipeline with an initiation interval of two. However, some sequences of the six instructions require insertion of one delay, and some require insertion of two delays. Of the 90 possible sequences of two  $A$ 's, two  $B$ 's and two  $C$ 's, how many require no delay? How many require one delay?

*Hint:* There is symmetry among the three instruction types so two sequences that can be transformed into one another by permuting the names  $A$ ,  $B$ , and  $C$  must require the same number of delays. For example,  $ABBCAC$  must be the same as  $BCCABA$ .

## 10.6 Summary of Chapter 10

- ◆ *Architectural Issues:* Optimized code scheduling takes advantage of features of modern computer architectures. Such machines often allow pipelined execution, where several instructions are in different stages of execution at the same time. Some machines also allow several instructions to begin execution at the same time.
- ◆ *Data Dependences:* When scheduling instructions, we must be aware of the effect instructions have on each memory location and register. True data dependences occur when one instruction must read a location after another has written it. Antidependences occur when there is a write after a read, and output dependences occur when there are two writes to the same location.
- ◆ *Eliminating Dependences:* By using additional locations to store data, antidependences and output dependences can be eliminated. Only true dependences cannot be eliminated and must surely be respected when the code is scheduled.
- ◆ *Data-Dependence Graphs for Basic Blocks:* These graphs represent the timing constraints among the statements of a basic block. Nodes correspond to the statements. An edge from  $n$  to  $m$  labeled  $d$  says that the instruction  $m$  must start at least  $d$  clock cycles after instruction  $n$  starts.
- ◆ *Prioritized Topological Orders:* The data-dependence graph for a basic block is always acyclic, and there usually are many topological orders consistent with the graph. One of several heuristics can be used to select a preferred topological order for a given graph, e.g., choose nodes with the longest critical path first.
- ◆ *List Scheduling:* Given a prioritized topological order for a data-dependence graph, we may consider the nodes in that order. Schedule each node at the earliest clock cycle that is consistent with the timing constraints implied by the graph edges, the schedules of all previously scheduled nodes, and the resource constraints of the machine.
- ◆ *Interblock Code Motion:* Under some circumstances it is possible to move statements from the block in which they appear to a predecessor or successor block. The advantage is that there may be opportunities to execute instructions in parallel at the new location that do not exist at the original location. If there is not a dominance relation between the old and

new locations, it may be necessary to insert compensation code along certain paths, in order to make sure that exactly the same sequence of instructions is executed, regardless of the flow of control.

- ◆ *Do-All Loops*: A do-all loop has no dependences across iterations, so any iterations may be executed in parallel.
- ◆ *Software Pipelining of Do-All Loops*: Software pipelining is a technique for exploiting the ability of a machine to execute several instructions at once. We schedule iterations of the loop to begin at small intervals, perhaps placing no-op instructions in the iterations to avoid conflicts between iterations for the machine's resources. The result is that the loop can be executed quickly, with a preamble, a coda, and (usually) a tiny inner loop.
- ◆ *Do-Across Loops*: Most loops have data dependences from each iteration to later iterations. These are called do-across loops.
- ◆ *Data-Dependence Graphs for Do-Across Loops*: To represent the dependences among instructions of a do-across loop requires that the edges be labeled by a pair of values: the required delay (as for graphs representing basic blocks) and the number of iterations that elapse between the two instructions that have a dependence.
- ◆ *List Scheduling of Loops*: To schedule a loop, we must choose the one schedule for all the iterations, and also choose the initiation interval at which successive iterations commence. The algorithm involves deriving the constraints on the relative schedules of the various instructions in the loop by finding the length of the longest acyclic paths between the two nodes. These lengths have the initiation interval as a parameter, and thus put a lower bound on the initiation interval.

## 10.7 References for Chapter 10

For a more in-depth discussion on processor architecture and design, we recommend Hennessy and Patterson [5].

The concept of data dependence was first discussed in Kuck, Muraoka, and Chen [6] and Lamport [8] in the context of compiling code for multiprocessors and vector machines.

Instruction scheduling was first used in scheduling horizontal microcode ([2, 3, 11, and 12]). Fisher's work on microcode compaction led him to propose the concept of a VLIW machine, where compilers directly can control the parallel execution of operations [3]. Gross and Hennessy [4] used instruction scheduling to handle the delayed branches in the first MIPS RISC instruction set. This chapter's algorithm is based on Bernstein and Rodeh's [1] more general treatment of scheduling of operations for machines with instruction-level parallelism.

The basic idea behind software pipelining was first developed by Patel and Davidson [9] for scheduling hardware pipelines. Software pipelining was first used by Rau and Glaeser [10] to compile for a machine with specialized hardware designed to support software pipelining. The algorithm described here is based on Lam [7], which assumes no specialized hardware support.

1. Bernstein, D. and M. Rodeh, "Global instruction scheduling for superscalar machines," *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 241–255.
2. Dasgupta, S., "The organization of microprogram stores," *Computing Surveys* **11:1** (1979), pp. 39–65.
3. Fisher, J. A., "Trace scheduling: a technique for global microcode compaction," *IEEE Trans. on Computers* **C-30:7** (1981), pp. 478–490.
4. Gross, T. R. and Hennessy, J. L., "Optimizing delayed branches," *Proc. 15th Annual Workshop on Microprogramming* (1982), pp. 114–120.
5. Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufman, San Francisco, 2003.
6. Kuck, D., Y. Muraoka, and S. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup," *IEEE Transactions on Computers* **C-21:12** (1972), pp. 1293–1310.
7. Lam, M. S., "Software pipelining: an effective scheduling technique for VLIW machines," *Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328.
8. Lamport, L., "The parallel execution of DO loops," *Comm. ACM* **17:2** (1974), pp. 83–93.
9. Patel, J. H. and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays," *Proc. Third Annual Symposium on Computer Architecture* (1976), pp. 159–164.
10. Rau, B. R. and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," *Proc. 14th Annual Workshop on Microprogramming* (1981), pp. 183–198.
11. Tokoro, M., E. Tamura, and T. Takizuka, "Optimization of microprograms," *IEEE Trans. on Computers* **C-30:7** (1981), pp. 491–504.
12. Wood, G., "Global optimization of microprograms through modular control constructs," *Proc. 12th Annual Workshop in Microprogramming* (1979), pp. 1–6.

*This page intentionally left blank*