Custom_eBook

Team 13

Table of contents

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

schedule		resource- reservation table

alu mem

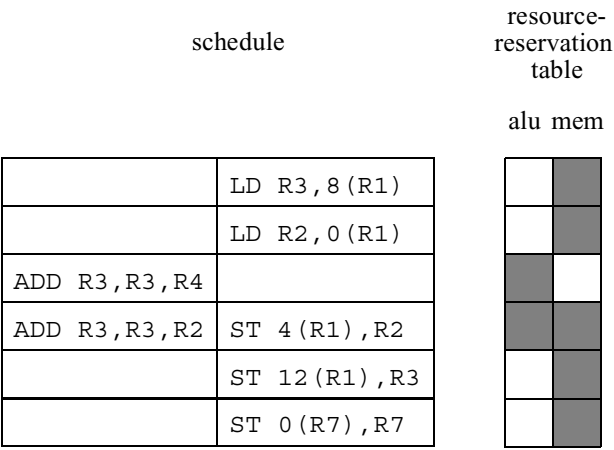|		alu	mem
	LD R3,8(R1)		▓
	LD R2,0(R1)		▓
ADD R3,R3,R4		▓	
ADD R3,R3,R2	ST 4(R1),R2	▓	▓
	ST 12(R1),R3		▓
	ST 0(R7),R7		▓

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.   □
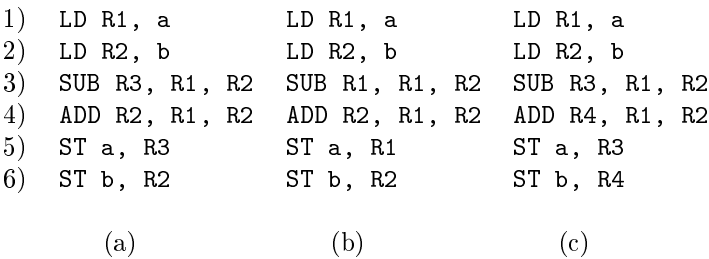
```
1)   LD R1, a          LD R1, a          LD R1, a
2)   LD R2, b          LD R2, b          LD R2, b
3)   SUB R3, R1, R2    SUB R1, R1, R2    SUB R3, R1, R2
4)   ADD R2, R1, R2    ADD R2, R1, R2    ADD R4, R1, R2
5)   ST a, R3          ST a, R1          ST a, R3
6)   ST b, R2          ST b, R2          ST b, R4

        (a)               (b)               (c)
```

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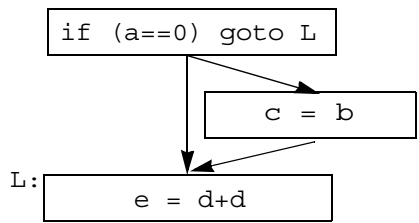
Operations in block B_2 are control-dependent on the test in block B_1. We can perform the load from B_2 speculatively in block B_1 for free and shave two clocks from the execution time whenever the branch is taken.

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. □
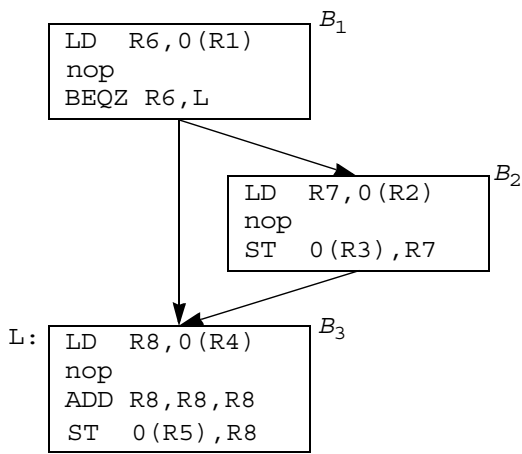
Example 10.9 shows that it is possible to move operations up and down an execution path. Every pair of basic blocks in this example has a different "dominance relation," and thus the considerations of when and how instructions can be moved between each pair are different. As discussed in Section 9.6.1, 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,
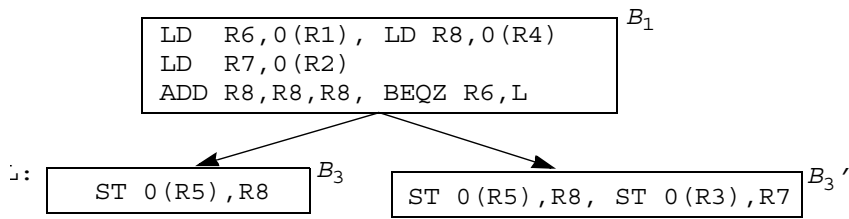
1. B_1 and B_3 are control equivalent: B_1 dominates B_3 and B_3 postdominates B_1,

2. B_1 dominates B_2 but B_2 does not postdominate B_1, and

```
if (a==0) goto L
```

```
c = b
```

L:
```
e = d+d
```

(a) Source program

B_1
```
LD   R6,0(R1)
nop
BEQZ R6,L
```

B_2
```
LD   R7,0(R2)
nop
ST   0(R3),R7
```

L:
B_3
```
LD   R8,0(R4)
nop
ADD R8,R8,R8
ST   0(R5),R8
```

(b) Locally scheduled machine code

B_1
```
LD   R6,0(R1), LD R8,0(R4)
LD   R7,0(R2)
ADD R8,R8,R8,  BEQZ R6,L
```

L:
B_3
```
ST 0(R5),R8
```

B_3'
```
ST 0(R5),R8, ST 0(R3),R7
```

(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: *src* postdom *dst*	*dst* dom *src*	speculation	compensation
	down: *src* dom *dst*	*dst* postdom *src*	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.   □
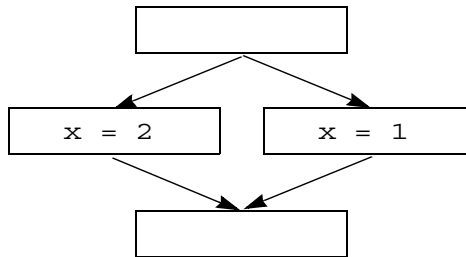

Figure 10.14: Example illustrating the change in data dependences due to code motion.

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) ∪
            DominatedSucc(ControlEquiv(B));
        CandInsts = ready instructions in CandBlocks;
        for (t = 0, 1, . . . 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) = ⟨B, t⟩;
                    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 sched-uled 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. □

```
        //   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 $k \to \infty$, 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.   □

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.   □

```
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 $n \to \infty$, 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 itera-tions, 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. □

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.   □

```
                    // 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 ith 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 $R = [r_1, r_2, \ldots]$, where r_i is the number of units of the ith 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.
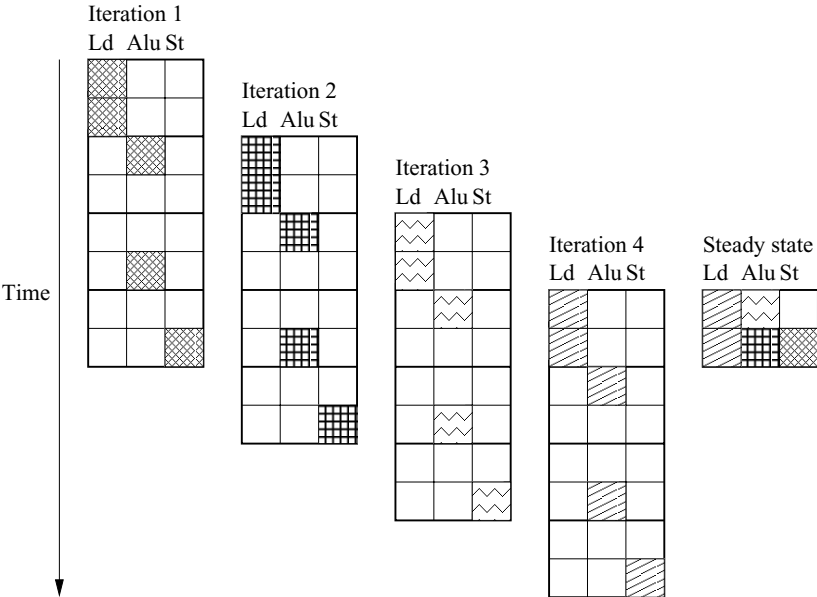

Figure 10.24: Resource requirements of four consecutive iterations from the code in Example 10.13

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 ith 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, δ, 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. □

```
//   R1, R2 = q, p
//   R3 = c
```
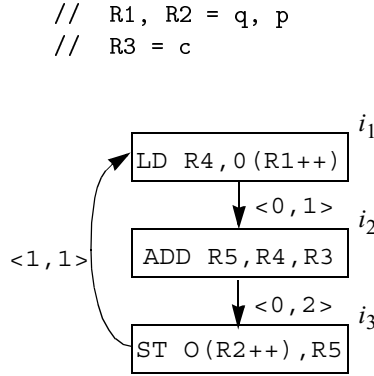

Figure 10.25: Data-dependence graph for Example 10.18

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 ith 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 δ_eth preceding iteration.

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

METHOD: Execute the program in Fig. 10.26.   □

```
main() {
        T_0 = max_j ⌈ (Σ_{n,i} RT_n(i,j)) / r_j ⌉;
        for (T = T_0, T_0 + 1, . . . , 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→n in E} (S(p) + d_e);
                        for (s = s_0, s_0 + 1, . . . , 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) mod T] = RT'[(s + i) mod T] + RT_n[i];
        if (for all i, RT'(i) ≤ 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

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

Observe that

- Around any cycle, the sum of the δ'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).
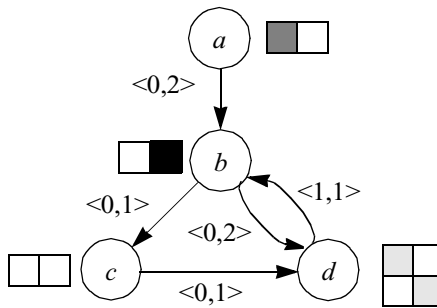
Figure 10.27: Dependence graph and resource requirement in Example 10.20

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 δ, for each edge. The delay is represented directly, but δ 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 δ'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 δ'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

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

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. □

	a	b	c	d
a		2		
b			1	2
c				1
d		1−T		

(a) Original edges.

	a	b	c	d
a		2	3	4
b			1	2
c		2−T		1
d		1−T	2−T	

(b) Longest simple paths.

	a	b	c	d
a		2	3	4
b			1	2
c		−1		1
d		−2	−1	

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

	a	b	c	d
a		2	3	4
b			1	2
c		−2		1
d		−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 $R = [r_1, r_2, \ldots]$, where r_i is the number of units available of the ith 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 δ_eth preceding iteration.

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

METHOD: Execute the program in Fig. 10.29. □

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,

$main()$ {

  $E' = \{e|e \text{ in } E, \delta_e = 0\};$

  $T_0 = \max\left(\max\limits_j\left\lceil\dfrac{\sum_{n,i} RT_n(i,j)}{r_j}\right\rceil, \max\limits_{c \text{ a cycle in } G}\left\lceil\dfrac{\sum_{e \text{ in } c} d_e}{\sum_{e \text{ in } c} \delta_e}\right\rceil\right);$

  for $(T = T_0, T_0 + 1, \ldots$ or until all SCC's in G are scheduled) {

    $RT =$ an empty reservation table with T rows;

    $E^* = AllPairsLongestPath(G, T);$

    for (each SCC C in G in prioritized topological order) {

      for (all n in C)

        $s_0(n) = \max_{e=p\to n \text{ in } E^*, p \text{ scheduled}} (S(p) + d_e);$

      $first =$ 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 $(SccScheduled\ (RT, T, C, first, s))$ **break**;

      if (C cannot be scheduled in RT) **break**;

    }

  }

}

$SccScheduled(RT, T, c, first, s)$ {

  $RT' = RT;$

  if (not $NodeScheduled\ (RT', T, first, s)$) **return false**;

  for (each remaining n in c in prioritized

        topological order of edges in E') {

    $s_l = \max_{e=n'\to n \text{ in } E^*, n' \text{ in } c, n' \text{ scheduled}}\big(S(n') + d_e - (\delta_e \times T)\big);$

    $s_u = \min_{e=n\to n' \text{ in } E^*, n' \text{ in } c, n' \text{ scheduled}}\big(S(n') - d_e + (\delta_e \times T)\big);$

    for $(s = s_l;\ s \leq \min(s_u, s_l + T - 1);\ s = s + 1)$

      if $\big(NodeScheduled(RT', T, n, s)\big)$ **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

dependences 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.   □

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 ith iteration of the loop read and write the ith 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 ith 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 $(i \bmod q_i')$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

$$n_2 = \begin{cases} \left\lceil \frac{L}{T} \right\rceil - 1 + Q n_1 & \text{if } n \geq \left\lceil \frac{L}{T} \right\rceil + Q - 1 \\ 0 & \text{otherwise} \end{cases}$$

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, 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 af-
   ter each arithmetic operation), unroll the loop twice. Schedule the second
   iteration to commence at the earliest possible moment without violat-
   ing 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 itera-
   tion 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 super-scalar 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

Chapter 11

Optimizing for Parallelism and Locality

This chapter shows how a compiler can enhance parallelism and locality in computationally intensive programs involving arrays to speed up target programs running on multiprocessor systems. Many scientific, engineering, and commercial applications have an insatiable need for computational cycles. Examples include weather prediction, protein-folding for designing drugs, fluid-dynamics for designing aeropropulsion systems, and quantum chromodynamics for studying the strong interactions in high-energy physics.

One way to speed up a computation is to use parallelism. Unfortunately, it is not easy to develop software that can take advantage of parallel machines. Dividing the computation into units that can execute on different processors in parallel is already hard enough; yet that by itself does not guarantee a speedup. We must also minimize interprocessor communication, because communication overhead can easily make the parallel code run even slower than the sequential execution!

Minimizing communication can be thought of as a special case of improving a program's *data locality*. In general, we say that a program has good data locality if a processor often accesses the same data it has used recently. Surely if a processor on a parallel machine has good locality, it does not need to communicate with other processors frequently. Thus, parallelism and data locality need to be considered hand-in-hand. Data locality, by itself, is also important for the performance of individual processors. Modern processors have one or more level of caches in the memory hierarchy; a memory access can take tens of machine cycles whereas a cache hit would only take a few cycles. If a program does not have good data locality and misses in the cache often, its performance will suffer.

Another reason why parallelism and locality are treated together in this same chapter is that they share the same theory. If we know how to optimize for data locality, we know where the parallelism is. You will see in this chapter that the

program model we used for data-flow analysis in Chapter 9 is inadequate for parallelization and locality optimization. The reason is that work on data-flow analysis assumes we don't distinguish among the ways a given statement is reached, and in fact these Chapter 9 techniques take advantage of the fact that we don't distinguish among different executions of the same statement, e.g., in a loop. To parallelize a code, we need to reason about the dependences among different dynamic executions of the same statement to determine if they can be executed on different processors simultaneously.

This chapter focuses on techniques for optimizing the class of numerical applications that use arrays as data structures and access them with simple regular patterns. More specifically, we study programs that have *affine* array accesses with respect to surrounding loop indexes. For example, if i and j are the index variables of surrounding loops, then $Z[i][j]$ and $Z[i][i+j]$ are affine accesses. A function of one or more variables, $x_1, x_2, \ldots, x_n$ is *affine* if it can be expressed as a sum of a constant, plus constant multiples of the variables, i.e., $c_0 + c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$, where $c_0, c_1, \ldots, c_n$ are constants. Affine functions are usually known as linear functions, although strictly speaking, linear functions do not have the c_0 term.

Here is a simple example of a loop in this domain:

```
for (i = 0; i < 10; i++) {
    Z[i] = 0;
}
```

Because iterations of the loop write to different locations, different processors can execute different iterations concurrently. On the other hand, if there is another statement Z[j] = 1 being executed, we need to worry about whether i could ever be the same as j, and if so, in which order do we execute those instances of the two statements that share a common value of the array index.

Knowing which iterations can refer to the same memory location is important. This knowledge lets us specify the data dependences that must be honored when scheduling code for both uniprocessors and multiprocessors. Our objective is to find a schedule that honors all the data dependences such that operations that access the same location and cache lines are performed close together if possible, and on the same processor in the case of multiprocessors.

The theory we present in this chapter is grounded in linear algebra and integer programming techniques. We model iterations in an n-deep loop nest as an n-dimensional polyhedron, whose boundaries are specified by the bounds of the loops in the code. Affine functions map each iteration to the array locations it accesses. We can use integer linear programming to determine if there exist two iterations that can refer to the same location.

The set of code transformations we discuss here fall into two categories: *affine partitioning* and *blocking*. Affine partitioning splits up the polyhedra of iterations into components, to be executed either on different machines or one-by-one sequentially. On the other hand, blocking creates a hierarchy of iterations. Suppose we are given a loop that sweeps through an array row-by-

row. We may instead subdivide the array into blocks and visit all elements in a block before moving to the next. The resulting code will consist of outer loops traversing the blocks, and then inner loops to sweep the elements within each block. Linear algebra techniques are used to determine both the best affine partitions and the best blocking schemes.

In the following, we first start with an overview of the concepts in parallel computation and locality optimization in Section 11.1. Then, Section 11.2 is an extended concrete example — matrix multiplication — that shows how *loop transformations* that reorder the computation inside a loop can improve both locality and the effectiveness of parallelization.

Sections 11.3 to Sections 11.6 present the preliminary information necessary for loop transformations. Section 11.3 shows how we model the individual iterations in a loop nest; Section 11.4 shows how we model array index functions that map each loop iteration to the array locations accessed by the iteration; Section 11.5 shows how to determine which iterations in a loop refer to the same array location or the same cache line using standard linear algebra algorithms; and Section 11.6 shows how to find all the data dependences among array references in a program.

The rest of the chapter applies these preliminaries in coming up with the optimizations. Section 11.7 first looks at the simpler problem of finding parallelism that requires no synchronization. To find the best affine partitioning, we simply find the solution to the constraint that operations that share a data dependence must be assigned to the same processor.

Well, not too many programs can be parallelized without requiring any synchronization. Thus, in Sections 11.8 through 11.9.9, we consider the general case of finding parallelism that requires synchronization. We introduce the concept of pipelining, show how to find the affine partitioning that maximizes the degree of pipelining allowed by a program. We show how to optimize for locality in Section 11.10. Finally, we discuss how affine transforms are useful for optimizing for other forms of parallelism.

11.1 Basic Concepts

This section introduces the basic concepts related to parallelization and locality optimization. If operations can be executed in parallel, they also can be reordered for other goals such as locality. Conversely, if data dependences in a program dictate that instructions in a program must execute serially, there is obviously no parallelism, nor is there any opportunity to reorder instructions to improve locality. Thus parallelization analysis also finds the available opportunities for code motion to improve data locality.

To minimize communication in parallel code, we group together all related operations and assign them to the same processor. The resulting code must therefore have data locality. One crude approach to getting good data locality on a uniprocessor is to have the processor execute the code assigned to each

processor in succession.

In this introduction, we start by presenting an overview of parallel computer architectures. We then show the basic concepts in parallelization, the kind of transformations that can make a big difference, as well as the concepts useful for parallelization. We then discuss how similar considerations can be used to optimize locality. Finally, we introduce informally the mathematical concepts used in this chapter.

11.1.1 Multiprocessors

The most popular parallel machine architecture is the symmetric multiprocessor (SMP). High-performance personal computers often have two processors, and many server machines have four, eight, and some even tens of processors. Moreover, as it has become feasible for several high-performance processors to fit on a single chip, multiprocessors have become even more widely used.

Processors on a symmetric multiprocessor share the same address space. To communicate, a processor can simply write to a memory location, which is then read by any other processor. Symmetric multiprocessors are so named because all processors can access all of the memory in the system with a uniform access time. Fig. 11.1 shows the high-level architecture of a multiprocessor. The processors may have their own first-level, second-level, and in some cases, even a third-level cache. The highest-level caches are connected to physical memory through typically a shared bus.
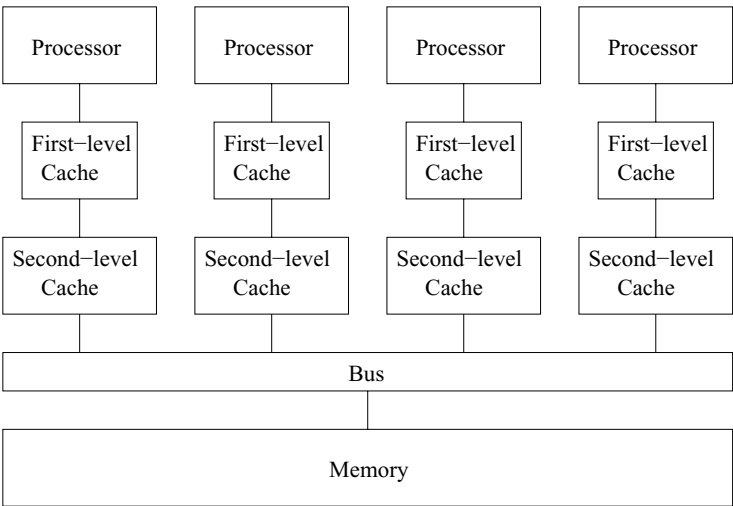

Figure 11.1: The symmetric multi-processor architecture

Symmetric multiprocessors use a *coherent cache protocol* to hide the presence of caches from the programmer. Under such a protocol, several processors are

allowed to keep copies of the same cache line[1] at the same time, provided that they are only reading the data. When a processor wishes to write to a cache line, copies from all other caches are removed. When a processor requests data not found in its cache, the request goes out on the shared bus, and the data will be fetched either from memory or from the cache of another processor.

The time taken for one processor to communicate with another is about twice the cost of a memory access. The data, in units of cache lines, must first be written from the first processor's cache to memory, and then fetched from the memory to the cache of the second processor. You may think that interprocessor communication is relatively cheap, since it is only about twice as slow as a memory access. However, you must remember that memory accesses are very expensive when compared to cache hits—they can be a hundred times slower. This analysis brings home the similarity between efficient parallelization and locality analysis. For a processor to perform well, either on its own or in the context of a multiprocessor, it must find most of the data it operates on in its cache.

In the early 2000's, the design of symmetric multiprocessors no longer scaled beyond tens of processors, because the shared bus, or any other kind of interconnect for that matter, could not operate at speed with the increasing number of processors. To make processor designs scalable, architects introduced yet another level in the memory hierarchy. Instead of having memory that is equally far away for each processor, they distributed the memories so that each processor could access its local memory quickly as shown in Fig. 11.2. Remote memories thus constituted the next level of the memory hierarchy; they are collectively bigger but also take longer to access. Analogous to the principle in memory-hierarchy design that fast stores are necessarily small, machines that support fast interprocessor communication necessarily have a small number of processors.

There are two variants of a parallel machine with distributed memories: NUMA (nonuniform memory access) machines and message-passing machines. NUMA architectures provide a shared address space to the software, allowing processors to communicate by reading and writing shared memory. On message-passing machines, however, processors have disjoint address spaces, and processors communicate by sending messages to each other. Note that even though it is simpler to write code for shared memory machines, the software must have good locality for either type of machine to perform well.

11.1.2 Parallelism in Applications

We use two high-level metrics to estimate how well a parallel application will perform: *parallelism coverage* which is the percentage of the computation that runs in parallel, *granularity of parallelism*, which is the amount of computation that each processor can execute without synchronizing or communicating with others. One particularly attractive target of parallelization is loops: a loop may

---

[1] You may wish to review the discussion of caches and cache lines in Section 7.4.
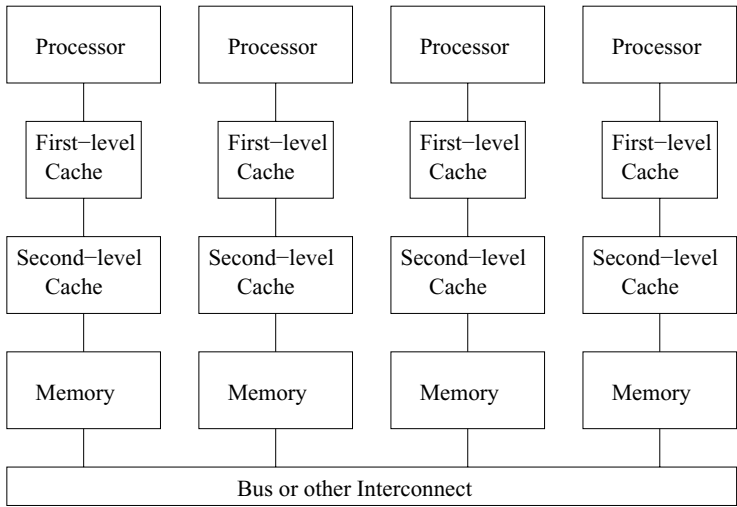
Figure 11.2: Distributed memory machines

have many iterations, and if they are independent of each other, we have found a great source of parallelism.

Amdahl's Law

The significance of parallelism coverage is succinctly captured by Amdahl's Law. *Amdahl's Law* states that, if f is the fraction of the code parallelized, and if the parallelized version runs on a p-processor machine with no communication or parallelization overhead, the speedup is

$$\frac{1}{(1 - f) + (f/p)}.$$

For example, if half of the computation remains sequential, the computation can only double in speed, regardless of how many processors we use. The speedup achievable is a factor of 1.6 if we have 4 processors. Even if the parallelism coverage is 90%, we get at most a factor of 3 speed up on 4 processors, and a factor of 10 on an unlimited number of processors.

Granularity of Parallelism

It is ideal if the entire computation of an application can be partitioned into many independent coarse-grain tasks because we can simply assign the different tasks to different processors. One such example is the SETI (Search for Extra-Terrestrial Intelligence) project, which is an experiment that uses home computers connected over the Internet to analyze different portions of radio telescope data in parallel. Each unit of work, requiring only a small amount

of input and generating a small amount of output, can be performed independently of all others. As a result, such a computation runs well on machines over the Internet, which has relatively high communication latency (delay) and low bandwidth.

Most applications require more communication and interaction between processors, yet still allow coarse-grained parallelism. Consider, for example, the web server responsible for serving a large number of mostly independent requests out of a common database. We can run the application on a multiprocessor, with a thread implementing the database and a number of other threads servicing user requests. Other examples include drug design or airfoil simulation, where the results of many different parameters can be evaluated independently. Sometimes the evaluation of even just one set of parameters in a simulation takes so long that it is desirable to speed it up with parallelization. As the granularity of available parallelism in an application decreases, better interprocessor communication support and more programming effort are needed.

Many long-running scientific and engineering applications, with their simple control structures and large data sets, can be more readily parallelized at a finer grain than the applications mentioned above. Thus, this chapter is devoted primarily to techniques that apply to numerical applications, and in particular, to programs that spend most of their time manipulating data in multidimensional arrays. We shall examine this class of programs next.

11.1.3 Loop-Level Parallelism

Loops are the main target for parallelization, especially in applications using arrays. Long running applications tend to have large arrays, which lead to loops that have many iterations, one for each element in the array. It is not uncommon to find loops whose iterations are independent of one another. We can divide the large number of iterations of such loops among the processors. If the amount of work performed in each iteration is roughly the same, simply dividing the iterations evenly across processors will achieve maximum parallelism. Example 11.1 is an extremely simple example showing how we can take advantage of loop-level parallelism.

Example 11.1 : The loop

```
for (i = 0; i < n; i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}
```

computes the square of differences between elements in vectors X and Y and stores it into Z. The loop is parallelizable because each iteration accesses a different set of data. We can execute the loop on a computer with M processors by giving each processor an unique ID $p = 0, 1, \ldots, M-1$ and having each processor execute the same code:

Task-Level Parallelism

It is possible to find parallelism outside of iterations in a loop. For example, we can assign two different function invocations, or two independent loops, to two processors. This form of parallelism is known as *task parallelism*. The task level is not as attractive a source of parallelism as is the loop level. The reason is that the number of independent tasks is a constant for each program and does not scale with the size of the data, as does the number of iterations of a typical loop. Moreover, the tasks generally are not of equal size, so it is hard to keep all the processors busy all the time.

```
b = ceil(n/M);
for (i = b*p; i < min(n,b*(p+1)); i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}
```

We divide the iterations in the loop evenly among the processors; the pth processor is given the pth swath of iterations to execute. Note that the number of iterations may not be divisible by M, so we assure that the last processor does not execute past the bound of the original loop by introducing a minimum operation. □

The parallel code shown in Example 11.1 is an SPMD (Single Program Multiple Data) program. The same code is executed by all processors, but it is parameterized by an identifier unique to each processor, so different processors can take different actions. Typically one processor, known as the *master*, executes all the serial part of the computation. The master processor, upon reaching a parallelized section of the code, wakes up all the *slave* processors. All the processors execute the parallelized regions of the code. At the end of each parallelized region of code, all the processors participate in a *barrier synchronization*. Any operation executed before a processor enters a synchronization barrier is guaranteed to be completed before any other processors are allowed to leave the barrier and execute operations that come after the barrier.

If we parallelize only little loops like those in Example 11.1, then the resulting code is likely to have low parallelism coverage and relatively fine-grain parallelism. We prefer to parallelize the outermost loops in a program, as that yields the coarsest granularity of parallelism. Consider, for example, the application of a two-dimensional FFT transformation that operates on an $n \times n$ data set. Such a program performs n FFT's on the rows of the data, then another n FFT's on the columns. It is preferable to assign each of the n independent FFT's to one processor each, rather than trying to use several processors to collaborate on one FFT. The code is easier to write, the parallelism coverage

for the algorithm is 100%, and the code has good data locality as it requires no communication at all while computing an FFT.

Many applications do not have large outermost loops that are parallelizable. The execution time of these applications, however, is often dominated by time-consuming *kernels*, which may have hundreds of lines of code consisting of loops with different nesting levels. It is sometimes possible to take the kernel, reorganize its computation and partition it into mostly independent units by focusing on its locality.

11.1.4 Data Locality

There are two somewhat different notions of data locality that need to be considered when parallelizing programs. *Temporal* locality occurs when the same data is used several times within a short time period. *Spatial* locality occurs when different data elements that are located near to each other are used within a short period of time. An important form of spatial locality occurs when all the elements that appear on one cache line are used together. The reason is that as soon as one element from a cache line is needed, all the elements in the same line are brought to the cache and will probably still be there if they are used soon. The effect of this spatial locality is that cache misses are minimized, with a resulting important speedup of the program.

Kernels can often be written in many semantically equivalent ways but with widely varying data localities and performances. Example 11.2 shows an alternative way of expressing the computation in Example 11.1.

Example 11.2 : Like Example 11.1 the following also finds the squares of differences between elements in vectors X and Y.

```
for (i = 0; i < n; i++)
    Z[i] = X[i] - Y[i];
for (i = 0; i < n; i++)
    Z[i] = Z[i] * Z[i];
```

The first loop finds the differences, the second finds the squares. Code like this appears often in real programs, because that is how we can optimize a program for *vector machines*, which are supercomputers which have instructions that perform simple arithmetic operations on vectors at a time. We see that the bodies of the two loops here are *fused* as one in Example 11.1.

Given that the two programs perform the same computation, which performs better? The fused loop in Example 11.1 has better performance because it has better data locality. Each difference is squared immediately, as soon as it is produced; in fact, we can hold the difference in a register, square it, and write the result just once into the memory location $Z[i]$. In contrast, the code in this example writes $Z[i]$ long before it uses that value. If the size of the array is larger than the cache, $Z[i]$ needs to be refetched from memory the second time it is used in this example. Thus, this code can run significantly slower. □

```
for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
        Z[i,j] = 0;
```

(a) Zeroing an array column-by-column.

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        Z[i,j] = 0;
```

(b) Zeroing an array row-by-row.

```
b = ceil(n/M);
for (i = b*p; i < min(n,b*(p+1)); i++)
    for (j = 0; j < n; j++)
        Z[i,j] = 0;
```

(c) Zeroing an array row-by-row in parallel.

Figure 11.3: Sequential and parallel code for zeroing an array

Example 11.3 : Suppose we want to set array Z, stored in row-major order (recall Section 6.4.3), to all zeros. Fig. 11.3(a) and (b) sweeps through the array column-by-column and row-by-row, respectively. We can transpose the loops in Fig. 11.3(a) to arrive at Fig. 11.3(b). In terms of spatial locality, it is preferable to zero out the array row-by-row since all the words in a cache line are zeroed consecutively. In the column-by-column approach, even though each cache line is reused by consecutive iterations of the outer loop, cache lines will be thrown out before reuse if the size of a colum is greater than the size of the cache. For best performance, we parallelize the outer loop of Fig. 11.3(b) in a manner similar to that used in Example 11.1 [see Fig. 11.3(c)]. □

The two examples above illustrate several important characteristics associated with numeric applications operating on arrays:

- Array code often has many parallelizable loops.

- When loops have parallelism, their iterations can be executed in arbitrary order; they can be reordered to improve data locality drastically.

- As we create large units of parallel computation that are independent of each other, executing these serially tends to produce good data locality.

11.1.5 Introduction to Affine Transform Theory

Writing correct and efficient sequential programs is difficult; writing parallel programs that are correct and efficient is even harder. The level of difficulty

increases as the granularity of parallelism exploited decreases. As we see above, programmers must pay attention to data locality to get high performance. Furthermore, the task of taking an existing sequential program and parallelizing it is extremely hard. It is hard to catch all the dependences in the program, especially if it is not a program with which we are familiar. Debugging a parallel program is harder yet, because errors can be nondeterministic.

Ideally, a parallelizing compiler automatically translates ordinary sequential programs into efficient parallel programs and optimizes the locality of these programs. Unfortunately, compilers without high-level knowledge about the application, can only preserve the semantics of the original algorithm, which may not be amenable to parallelization. Furthermore, programmers may have made arbitrary choices that limit the program's parallelism.

Successes in parallelization and locality optimizations have been demonstrated for Fortran numeric applications that operate on arrays with affine accesses. Without pointers and pointer arithmetic, Fortran is easier to analyze. Note that not all applications have affine accesses; most notably, many numeric applications operate on sparse matrices whose elements are accessed indirectly through another array. This chapter focuses on the parallelization and optimizations of kernels, consisting of mostly tens of lines.

As illustrated by Examples 11.2 and 11.3, parallelization and locality optimization require that we reason about the different instances of a loop and their relations with each other. This situation is very different from data-flow analysis, where we combine information associated with all instances together.

For the problem of optimizing loops with array accesses, we use three kinds of spaces. Each space can be thought of as points on a grid of one or more dimensions.

1. The *iteration space* is the set of the dynamic execution instances in a computation, that is, the set of combinations of values taken on by the loop indexes.

2. The *data space* is the set of array elements accessed.

3. The *processor space* is the set of processors in the system. Normally, these processors are assigned integer numbers or vectors of integers to distinguish among them.

Given as input are a sequential order in which the iterations are executed and affine array-access functions (e.g., $X[i, j + 1]$) that specify which instances in the iteration space access which elements in the data space.

The output of the optimization, again represented as affine functions, defines what each processor does and when. To specify what each processor does, we use an affine function to assign instances in the original iteration space to processors. To specify when, we use an affine function to map instances in the iteration space to a new ordering. The schedule is derived by analyzing the array-access functions for data dependences and reuse patterns.

The following example will illustrate the three spaces — iteration, data, and processor. It will also introduce informally the important concepts and issues that need to be addressed in using these spaces to parallelize code. The concepts each will be covered in detail in later sections.

Example 11.4 : Figure 11.4 illustrates the different spaces and their relations used in the following program:

```
float Z[100];
for (i = 0; i < 10; i++)
    Z[i+10] = Z[i];
```

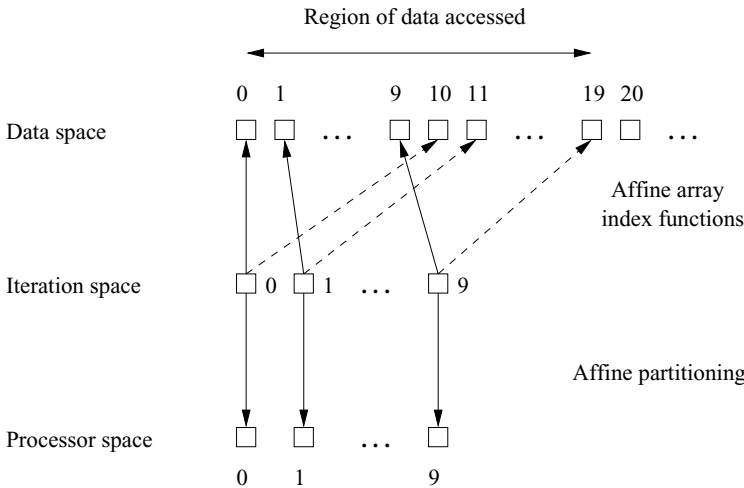The three spaces and the mappings among them are as follows:

Figure 11.4: Iteration, data, and processor space for Example 11.4

1. *Iteration Space*: The iteration space is the set of iterations, whose ID's are given by the values held by the loop index variables. A d-deep *loop nest* (i.e., d nested loops) has d index variables, and is thus modeled by a d-dimensional space. The space of iterations is bounded by the lower and upper bounds of the loop indexes. The loop of this example defines a one-dimensional space of 10 iterations, labeled by the loop index values: $i = 0, 1, \ldots, 9$.

2. *Data Space*: The data space is given directly by the array declarations. In this example, elements in the array are indexed by $a = 0, 1, \ldots, 99$. Even though all arrays are linearized in a program's address space, we treat n-dimensional arrays as n-dimensional spaces, and assume that the individual indexes stay within their bounds. In this example, the array is one-dimensional anyway.

3. *Processor Space*: We pretend that there are an unbounded number of virtual processors in the system as our initial parallelization target. The processors are organized in a multidimensional space, one dimension for each loop in the nest we wish to parallelize. After parallelization, if we have fewer physical processors than virtual processors, we divide the virtual processors into even blocks, and assign a block each to a processor. In this example, we need only ten processors, one for each iteration of the loop. We assume in Fig. 11.4 that processors are organized in a one-dimensional space and numbered $0, 1, \ldots, 9$, with loop iteration i assigned to processor i. If there were, say, only five processors, we could assign iterations 0 and 1 to processor 0, iterations 2 and 3 to processor 1, and so on. Since iterations are independent, it doesn't matter how we do the assignment, as long as each of the five processors gets two iterations.

4. *Affine Array-Index Function*: Each array access in the code specifies a mapping from an iteration in the iteration space to an array element in the data space. The access function is affine if it involves multiplying the loop index variables by constants and adding constants. Both the array index functions $i + 10$, and i are affine. From the access function, we can tell the *dimension* of the data accessed. In this case, since each index function has one loop variable, the space of accessed array elements is one dimensional.

5. *Affine Partitioning*: We parallelize a loop by using an affine function to assign iterations in an iteration space to processors in the processor space. In our example, we simply assign iteration i to processor i. We can also specify a new execution order with affine functions. If we wish to execute the loop above sequentially, but in reverse, we can specify the ordering function succinctly with an affine expression $10 - i$. Thus, iteration 9 is the 1st iteration to execute and so on.

6. *Region of Data Accessed*: To find the best affine partitioning, it useful to know the region of data accessed by an iteration. We can get the region of data accessed by combining the iteration space information with the array index function. In this case, the array access $Z[i + 10]$ touches the region $\{a \mid 10 \leq a < 20\}$ and the access $Z[i]$ touches the region $\{a | 0 \leq a < 10\}$.

7. *Data Dependence*: To determine if the loop is parallelizable, we ask if there is a data dependence that crosses the boundary of each iteration. For this example, we first consider the dependences of the write accesses in the loop. Since the access function $Z[i+10]$ maps different iterations to different array locations, there are no dependences regarding the order in which the various iterations write values to the array. Is there a dependence between the read and write accesses? Since only $Z[10], Z[11], \ldots, Z[19]$ are written (by the access $Z[i + 10]$), and only $Z[0], Z[1], \ldots, Z[9]$ are read (by the access $Z[i]$), there can be no dependencies regarding the relative order of a read and a write. Therefore, this loop is parallelizable. That

is, each iteration of the loop is independent of all other iterations, and we can execute the iterations in parallel, or in any order we choose. Notice, however, that if we made a small change, say by increasing the upper limit on loop index i to 10 or more, then there would be dependencies, as some elements of array Z would be written on one iteration and then read 10 iterations later. In that case, the loop could not be parallelized completely, and we would have to think carefully about how iterations were partitioned among processors and how we ordered iterations.

$\Box$

Formulating the problem in terms of multidimensional spaces and affine mappings between these spaces lets us use standard mathematical techniques to solve the parallelization and locality optimization problem generally. For example, the region of data accessed can be found by the elimination of variables using the Fourier-Motzkin elimination algorithm. Data dependence is shown to be equivalent to the problem of integer linear programming. Finally, finding the affine partitioning corresponds to solving a set of linear constraints. Don't worry if you are not familiar with these concepts, as they will be explained starting in Section 11.3.

11.2 Matrix Multiply: An In-Depth Example

We shall introduce many of the techniques used by parallel compilers in an extended example. In this section, we explore the familiar matrix-multiplication algorithm to show that it is nontrivial to optimize even a simple and easily parallelizable program. We shall see how rewriting the code can improve data locality; that is, processors are able to do their work with far less communication (with global memory or with other processors, depending on the architecture) than if the straightforward program is chosen. We shall also discuss how cognizance of the existence of cache lines that hold several consecutive data elements can improve the running time of programs such as matrix multiplication.

11.2.1 The Matrix-Multiplication Algorithm

In Fig. 11.5 we see a typical matrix-multiplication program.[2] It takes two $n \times n$ matrices, X and Y, and produces their product in a third $n \times n$ matrix Z. Recall that Z_{ij} — the element of matrix Z in row i and column j — must become $\sum_{k=1}^{n} X_{ik}Y_{kj}$.

The code of Fig. 11.5 generates n^2 results, each of which is an inner product between one row and one column of the two matrix operands. Clearly, the

---

[2] In programs of this chapter, we shall generally use C syntax, but to make multidimensional array accesses — the central issue for most of the chapter — easier to read, we shall use Fortran-style array references, that is, $Z[i,j]$ instead of $Z[i][j]$.

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++) {
      Z[i,j] = 0.0;
      for (k = 0; k < n; k++)
         Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
   }
```

Figure 11.5: The basic matrix-multiplication algorithm

calculations of each of the elements of Z are independent and can be executed in parallel.

The larger n is, the more times the algorithm touches each element. That is, there are $3n^2$ locations among the three matrices, but the algorithm performs n^3 operations, each of which multiplies an element of X by an element of Y and adds the product to an element of Z. Thus, the algorithm is computation-intensive and memory accesses should not, in principle, constitute a bottleneck.

Serial Execution of the Matrix Multiplication

Let us first consider how this program behaves when run sequentially on a uniprocessor. The innermost loop reads and writes the same element of Z, and uses a row of X and a column of Y. $Z[i,j]$ can easily be stored in a register and requires no memory accesses. Assume, without loss of generality, that the matrix is laid out in row-major order, and that c is the number of array elements in a cache line.
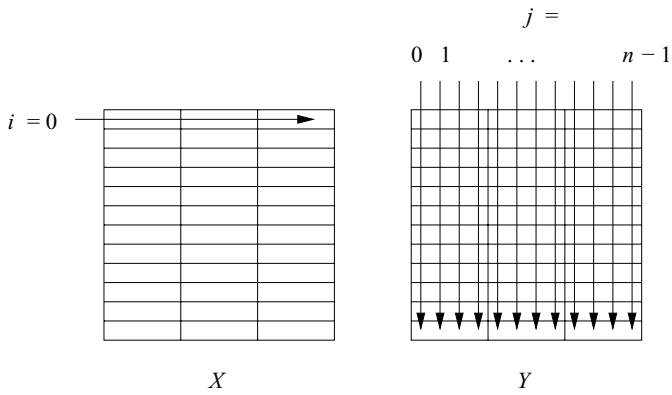

Figure 11.6: The data access pattern in matrix multiply

Figure 11.6 suggests the access pattern as we execute one iteration of the outer loop of Fig. 11.5. In particular, the picture shows the first iteration, with $i = 0$. Each time we move from one element of the first row of X to the next,

we visit each element in a single column of Y. We see in Fig. 11.6 the assumed organization of the matrices into cache lines. That is, each small rectangle represents a cache line holding four array elements (i.e., $c = 4$ and $n = 12$ in the picture).

Accessing X puts little burden on the cache. One row of X is spread among only n/c cache lines. Assuming these all fit in the cache, only n/c cache misses occur for a fixed value of index i, and the total number of misses for all of X is n^2/c, the minimum possible (we assume n is divisible by c, for convenience).

However, while using one row of X, the matrix-multiplication algorithm accesses all the elements of Y, column by column. That is, when $j = 0$, the inner loop brings to the cache the entire first column of Y. Notice that the elements of that column are stored among n different cache lines. If the cache is big enough (or n small enough) to hold n cache lines, and no other uses of the cache force some of these cache lines to be expelled, then the column for $j = 0$ will still be in the cache when we need the second column of Y. In that case, there will not be another n cache misses reading Y, until $j = c$, at which time we need to bring into the cache an entirely different set of cache lines for Y. Thus, to complete the first iteration of the outer loop (with $i = 0$) requires between n^2/c and n^2 cache misses, depending on whether columns of cache lines can survive from one iteration of the second loop to the next.

Moreover, as we complete the outer loop, for $i = 1, 2$, and so on, we may have many additional cache misses as we read Y, or none at all. If the cache is big enough that all n^2/c cache lines holding Y can reside together in the cache, then we need no more cache misses. The total number of cache misses is thus $2n^2/c$, half for X and half for Y. However, if the cache can hold one column of Y but not all of Y, then we need to bring all of Y into cache again, each time we perform an iteration of the outer loop. That is, the number of cache misses is $n^2/c + n^3/c$; the first term is for X and the second is for Y. Worst, if we cannot even hold one column of Y in the cache, then we have n^2 cache misses per iteration of the outer loop and a total of $n^2/c + n^3$ cache misses.

Row-by-Row Parallelization

Now, let us consider how we could use some number of processors, say p processors, to speed up the execution of Fig. 11.5. An obvious approach to parallelizing matrix multiplication is to assign different rows of Z to different processors. A processor is responsible for n/p consecutive rows (we assume n is divisible by p, for convenience). With this division of labor, each processor needs to access n/p rows of matrices X and Z, but the entire Y matrix. One processor will compute n^2/p elements of Z, performing n^3/p multiply-and-add operations to do so.

While the computation time thus decreases in proportion to p, the communication cost actually rises in proportion to p. That is, each of p processors has to read n^2/p elements of X, but all n^2 elements of Y. The total number of cache lines that must be delivered to the caches of the p processors is at least

$n^2/c + pn^2/c$; the two terms are for delivering X and copies of Y, respectively. As p approaches n, the computation time becomes $O(n^2)$ while the communication cost is $O(n^3)$. That is, the bus on which data is moved between memory and the processors' caches becomes the bottleneck. Thus, with the proposed data layout, using a large number of processors to share the computation can actually slow down the computation, rather than speed it up.

11.2.2 Optimizations

The matrix-multiplication algorithm of Fig. 11.5 shows that even though an algorithm may *reuse* the same data, it may have poor data locality. A reuse of data results in a cache hit only if the reuse happens soon enough, before the data is displaced from the cache. In this case, n^2 multiply-add operations separate the reuse of the same data element in matrix Y, so locality is poor. In fact, n operations separate the reuse of the same cache line in Y. In addition, on a multiprocessor, reuse may result in a cache hit only if the data is reused by the same processor. When we considered a parallel implementation in Section 11.2.1, we saw that elements of Y had to be used by every processor. Thus, the reuse of Y is not turned into locality.

Changing Data Layout

One way to improve the locality of a program is to change the layout of its data structures. For example, storing Y in column-major order would have improved the reuse of cache lines for matrix Y. The applicability of this approach is limited, because the same matrix normally is used in different operations. If Y played the role of X in another matrix multiplication, then it would suffer from being stored in column-major order, since the first matrix in a multiplication is better stored in row-major order.

Blocking

It is sometimes possible to change the execution order of the instructions to improve data locality. The technique of interchanging loops, however, does not improve the matrix-multiplication routine. Suppose the routine were written to generate a column of matrix Z at a time, instead of a row at a time. That is, make the j-loop the outer loop and the i-loop the second loop. Assuming matrices are still stored in row-major order, matrix Y enjoys better spatial and temporal locality, but only at the expense of matrix X.

$\quad$ *Blocking* is another way of reordering iterations in a loop that can greatly improve the locality of a program. Instead of computing the result a row or a column at a time, we divide the matrix up into submatrices, or *blocks*, as suggested by Fig. 11.7, and we order operations so an entire block is used over a short period of time. Typically, the blocks are squares with a side of length B. If B evenly divides n, then all the blocks are square. If B does not evenly

divide n, then the blocks on the lower and right edges will have one or both sides of length less than B.
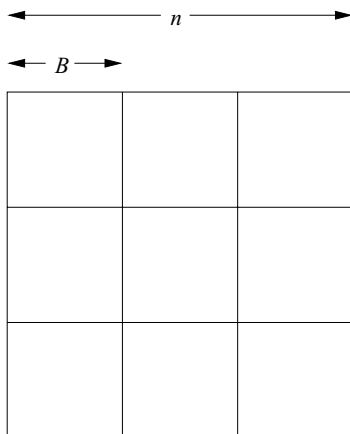

Figure 11.7: A matrix divided into blocks of side B

Figure 11.8 shows a version of the basic matrix-multiplication algorithm where all three matrices have been blocked into squares of side B. As in Fig. 11.5, Z is assumed to have been initialized to all 0's. We assume that B divides n; if not, then we need to modify line (4) so the upper limit is $\min(ii + B, n)$, and similarly for lines (5) and (6).

```
1)  for (ii = 0; ii < n; ii = ii+B)
2)      for (jj = 0; jj < n; jj = jj+B)
3)          for (kk = 0; kk < n; kk = kk+B)
4)              for (i = ii; i < ii+B; i++)
5)                  for (j = jj; j < jj+B; j++)
6)                      for (k = kk; k < kk+B; k++)
7)                          Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
```

Figure 11.8: Matrix multiplication with blocking

The outer three loops, lines (1) through (3), use indexes ii, jj, and kk, which are always incremented by B, and therefore always mark the left or upper edge of some blocks. With fixed values of ii, jj, and kk, lines (4) through (7) enable the blocks with upper-left corners $X[ii, kk]$ and $Y[kk, jj]$ to make all possible contributions to the block with upper-left corner $Z[ii, jj]$.

If we pick B properly, we can significantly decrease the number of cache misses, compared with the basic algorithm, when all of X, Y, or Z cannot fit in the cache. Choose B such that it is possible to fit one block from each of the matrices in the cache. Because of the order of the loops, we actually need each

---

Another View of Block-Based Matrix Multiplication

We can imagine that the matrices X, Y, and Z of Fig. 11.8 are not $n \times n$ matrices of floating-point numbers, but rather $(n/B) \times (n/B)$ matrices whose elements are themselves $B \times B$ matrices of floating-point numbers. Lines (1) through (3) of Fig. 11.8 are then like the three loops of the basic algorithm in Fig. 11.5, but with n/B as the size of the matrices, rather than n. We can then think of lines (4) through (7) of Fig. 11.8 as implementing a single multiply-and-add operation of Fig. 11.5. Notice that in this operation, the single multiply step is a matrix-multiply step, and it uses the basic algorithm of Fig. 11.5 on the floating-point numbers that are elements of the two matrices involved. The matrix addition is element-wise addition of floating-point numbers.

---

block of Z in cache only once, so (as in the analysis of the basic algorithm in Section 11.2.1) we shall not count the cache misses due to Z.

To bring a block of X or Y to the cache takes B^2/c cache misses; recall c is the number of elements in a cache line. However, with fixed blocks from X and Y, we perform B^3 multiply-and-add operations in lines (4) through (7) of Fig. 11.8. Since the entire matrix-multiplication requires n^3 multiply-and-add operations, the number of times we need to bring a pair of blocks to the cache is n^3/B^3. As we require $2B^2/c$ cache misses each time we do, the total number of cache misses is $2n^3/Bc$.

It is interesting to compare this figure $2n^3/Bc$ with the estimates given in Section 11.2.1. There, we said that if entire matrices can fit in the cache, then $O(n^2/c)$ cache misses suffice. However, in that case, we can pick $B = n$, i.e., make each matrix be a single block. We again get $O(n^2/c)$ as our estimate of cache misses. On the other hand, we observed that if entire matrices will not fit in cache, we require $O(n^3/c)$ cache misses, or even $O(n^3)$ cache misses. In that case, assuming that we can still pick a significantly large B (e.g., B could be 200, and we could still fit three blocks of 8-byte numbers in a one-megabyte cache), there is a great advantage to using blocking in matrix multiplication.

The blocking technique can be reapplied for each level of the memory hierarchy. For example, we may wish to optimize register usage by holding the operands of a 2×2 matrix multiplication in registers. We choose successively bigger block sizes for the different levels of caches and physical memory.

Similarly, we can distribute blocks between processors to minimize data traffic. Experiments showed that such optimizations can improve the performance of a uniprocessor by a factor of 3, and the speed up on a multiprocessor is close to linear with respect to the number of processors used.

11.2.3 Cache Interference

Unfortunately, there is somewhat more to the story of cache utilization. Most caches are not fully associative (see Section 7.4.2). In a direct-mapped cache, if n is a multiple of the cache size, then all the elements in the same row of an $n \times n$ array will be competing for the same cache location. In that case, bringing in the second element of a column will throw away the cache line of the first, even though the cache has the capacity to keep both of these lines at the same time. This situation is referred to as *cache interference*.

There are various solutions to this problem. The first is to rearrange the data once and for all so that the data accessed is laid out in consecutive data locations. The second is to embed the $n \times n$ array in a larger $m \times n$ array where m is chosen to minimize the interference problem. Third, in some cases we can choose a block size that is guaranteed to avoid interference.

11.2.4 Exercises for Section 11.2

Exercise 11.2.1 : The block-based matrix-multiplication algorithm of Fig. 11.8 does not have the initialization of the matrix Z to zero, as the code of Fig. 11.5 does. Add the steps that initialize Z to all zeros in Fig. 11.8.

11.3 Iteration Spaces

The motivation for this study is to exploit the techniques that, in simple settings like matrix multiplication as in Section 11.2, were quite straightforward. In the more general setting, the same techniques apply, but they are far less intuitive. But by applying some linear algebra, we can make everything work in the general setting.

As discussed in Section 11.1.5, there are three kinds of spaces in our transformation model: iteration space, data space, and processor space. Here we start with the iteration space. The iteration space of a loop nest is defined to be all the combinations of loop-index values in the nest.

Often, the iteration space is rectangular, as in the matrix-multiplication example of Fig. 11.5. There, each of the nested loops had a lower bound of 0 and an upper bound of $n - 1$. However, in more complicated, but still quite realistic, loop nests, the upper and/or lower bounds on one loop index can depend on the values of the indexes of the outer loops. We shall see an example shortly.

11.3.1 Constructing Iteration Spaces from Loop Nests

To begin, let us describe the sort of loop nests that can be handled by the techniques to be developed. Each loop has a single loop index, which we assume is incremented by 1 at each iteration. That assumption is without loss of generality, since if the incrementation is by integer $c > 1$, we can always replace

uses of the index i by uses of $ci + a$ for some positive or negative constant a, and then increment i by 1 in the loop. The bounds of the loop should be written as affine expressions of outer loop indices.

Example 11.5 : Consider the loop

```
for (i = 2; i <= 100; i = i+3)
    Z[i] = 0;
```

which increments i by 3 each time around the loop. The effect is to set to 0 each of the elements $Z[2], Z[5], Z[8], \ldots, Z[98]$. We can get the same effect with:

```
for (j = 0; j <= 32; j++)
    Z[3*j+2] = 0;
```

That is, we substitute $3j + 2$ for i. The lower limit $i = 2$ becomes $j = 0$ (just solve $3j + 2 = 2$ for j), and the upper limit $i \leq 100$ becomes $j \leq 32$ (simplify $3j + 2 \leq 100$ to get $j \leq 32.67$ and round down because j has to be an integer). □

Typically, we shall use for-loops in loop nests. A while-loop or repeat-loop can be replaced by a for-loop if there is an index and upper and lower bounds for the index, as would be the case in something like the loop of Fig. 11.9(a). A for-loop like `for (i=0; i<100; i++)` serves exactly the same purpose.

However, some while- or repeat-loops have no obvious limit. For example, Fig. 11.9(b) may or may not terminate, but there is no way to tell what condition on i in the unseen body of the loop causes the loop to break. Figure 11.9(c) is another problem case. Variable n might be a parameter of a function, for example. We know the loop iterates n times, but we don't know what n is at compile time, and in fact we may expect that different executions of the loop will execute different numbers of times. In cases like (b) and (c), we must treat the upper limit on i as infinity.

A d-deep loop nest can be modeled by a d-dimensional space. The dimensions are ordered, with the kth dimension representing the kth nested loop, counting from the outermost loop, inward. A point $(x_1, x_2, \ldots, x_d)$ in this space represents values for all the loop indexes; the outermost loop index has value x_1, the second loop index has value x_2, and so on. The innermost loop index has value x_d.

But not all points in this space represent combinations of indexes that actually occur during execution of the loop nest. As an affine function of outer loop indices, each lower and upper loop bound defines an inequality dividing the iteration space into two half spaces: those that are iterations in the loop (the *positive* half space), and those that are not (the *negative* half space). The conjunction (logical AND) of all the linear equalities represents the intersection of the positive half spaces, which defines a convex polyhedron, which we call the *iteration space* for the loop nest. A *convex polyhedron* has the property that if

```
i = 0;
while (i<100) {
    <some statements not involving i>
    i = i+1;
}
```

(a) A while-loop with obvious limits.

```
i = 0;
while (1) {
    <some statements>
    i = i+1;
}
```

(b) It is unclear when or if this loop terminates.

```
i = 0;
while (i<n) {
    <some statements not involving i or n>
    i = i+1;
}
```

(c) We don't know the value of n, so we don't
know when this loop terminates.

Figure 11.9: Some while-loops

two points are in the polyhedron, all points on the line between them are also in
the polyhedron. All the iterations in the loop are represented by the points with
integer coordinates found within the polyhedron described by the loop-bound
inequalities. And conversely, all integer points within the polyhedron represent
iterations of the loop nest at some time.

```
for (i = 0; i <= 5; i++)
    for (j = i; j <= 7; j++)
        Z[j,i] = 0;
```

Figure 11.10: A 2-dimensional loop nest

Example 11.6 : Consider the 2-dimensional loop nest in Fig. 11.10. We can
model this two-deep loop nest by the 2-dimensional polyhedron shown in Fig.
11.11. The two axes represent the values of the loop indexes i and j. Index i
can take on any integral value between 0 and 5; index j can take on any integral
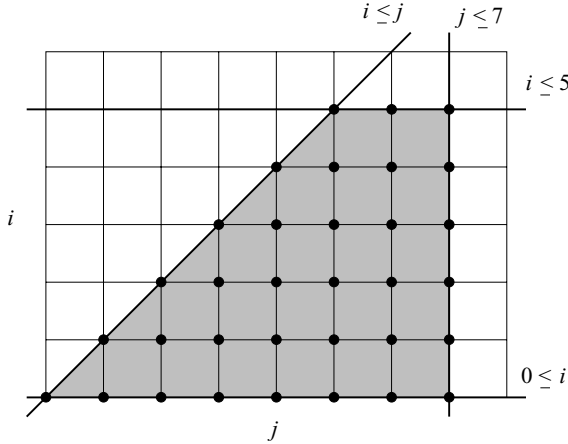value such that $i \leq j \leq 7$. □

Figure 11.11: The iteration space of Example 11.6

Iteration Spaces and Array-Accesses

In the code of Fig. 11.10, the iteration space is also the portion of the array A that the code accesses. That sort of access, where the array indexes are also loop indexes in some order, is very common. However, we should not confuse the space of iterations, whose dimensions are loop indexes, with the data space. If we had used in Fig. 11.10 an array access like $Z[2*i, i+j]$ instead of $Z[j, i]$, the difference would have been apparent.

11.3.2 Execution Order for Loop Nests

A sequential execution of a loop nest sweeps through iterations in its iteration space in an ascending lexicographic order. A vector $\mathbf{i} = [i_0, i_1, \ldots, i_n]$ is *lexicographically less than* another vector $\mathbf{i}' = [i'_0, i'_1, \ldots, i'_{n'}]$, written $\mathbf{i} \prec \mathbf{i}'$, if and only if there exists an $m < \min(n, n')$ such that $[i_0, i_1, \ldots, i_m] = [i'_0, i'_1, \ldots, i'_m]$ and $i_{m+1} < i'_{m+1}$. Note that $m = 0$ is possible, and in fact common.

Example 11.7: With i as the outer loop, the iterations in the loop nest in Example 11.6 are executed in the order shown in Fig. 11.12. □

11.3.3 Matrix Formulation of Inequalities

The iterations in a d-deep loop can be represented mathematically as

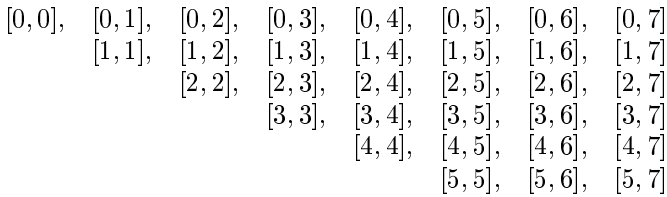$$\{\mathbf{i} \text{ in } Z^d \mid \mathbf{Bi} + \mathbf{b} \geq \mathbf{0}\} \tag{11.1}$$

$$
\begin{array}{cccccccc}
[0,0], & [0,1], & [0,2], & [0,3], & [0,4], & [0,5], & [0,6], & [0,7] \\
& [1,1], & [1,2], & [1,3], & [1,4], & [1,5], & [1,6], & [1,7] \\
& & [2,2], & [2,3], & [2,4], & [2,5], & [2,6], & [2,7] \\
& & & [3,3], & [3,4], & [3,5], & [3,6], & [3,7] \\
& & & & [4,4], & [4,5], & [4,6], & [4,7] \\
& & & & & [5,5], & [5,6], & [5,7]
\end{array}
$$

Figure 11.12: Iteration order for loop nest of Fig. 11.10

Here,

1. Z, as is conventional in mathematics, represents the set of integers — positive, negative, and zero,

2. $\mathbf{B}$ is a $d \times d$ integer matrix,

3. $\mathbf{b}$ is an integer vector of length d, and

4. $\mathbf{0}$ is a vector of d 0's.

Example 11.8 : We can write the inequalities of Example 11.6 as in Fig. 11.13. That is, the range of i is described by $i \geq 0$ and $i \leq 5$; the range of j is described by $j \geq i$ and $j \leq 7$. We need to put each of these inequalities in the form $ui + vj + w \geq 0$. Then, $[u, v]$ becomes a row of the matrix $\mathbf{B}$ in the inequality (11.1), and w becomes the corresponding component of the vector $\mathbf{b}$. For instance, $i \geq 0$ is of this form, with $u = 1$, $v = 0$, and $w = 0$. This inequality is represented by the first row of $\mathbf{B}$ and top element of $\mathbf{b}$ in Fig. 11.13.

$$
\begin{bmatrix}
1 & 0 \\
-1 & 0 \\
-1 & 1 \\
0 & -1
\end{bmatrix}
\begin{bmatrix}
i \\
j
\end{bmatrix}
+
\begin{bmatrix}
0 \\
5 \\
0 \\
7
\end{bmatrix}
\geq
\begin{bmatrix}
0 \\
0 \\
0 \\
0
\end{bmatrix}
$$

Figure 11.13: Matrix-vector multiplication and a vector inequality represents the inequalities defining an iteration space

As another example, the inequality $i \leq 5$ is equivalent to $(-1)i + (0)j + 5 \geq 0$, and is represented by the second row of $\mathbf{B}$ and $\mathbf{b}$ in Fig. 11.13. Also, $j \geq i$ becomes $(-1)i + (1)j + 0 \geq 0$ and is represented by the third row. Finally, $j \leq 7$ becomes $(0)i + (-1)j + 7 \geq 0$ and is the last row of the matrix and vector. $\quad\square$

Manipulating Inequalities

To convert inequalities, as in Example 11.8, we can perform transforma-
tions much as we do for equalities, e.g., adding or subtracting from both
sides, or multiplying both sides by a constant. The only special rule we
must remember is that when we multiply both sides by a negative number,
we have to reverse the direction of the inequality. Thus, $i \leq 5$, multiplied
by -1, becomes $-i \geq -5$. Adding 5 to both sides, gives $-i + 5 \geq 0$, which
is essentially the second row of Fig. 11.13.

11.3.4 Incorporating Symbolic Constants

Sometimes, we need to optimize a loop nest that involves certain variables that
are loop-invariant for all the loops in the nest. We call such variables *symbolic
constants*, but to describe the boundaries of an iteration space we need to treat
them as variables and create an entry for them in the vector of loop indexes,
i.e., the vector **i** in the general formulation of inequalities (11.1).

Example 11.9 : Consider the simple loop:

```
for (i = 0; i <= n; i++) {
    ...
}
```

This loop defines a one-dimensional iteration space, with index i, bounded by
$i \geq 0$ and $i \leq n$. Since n is a symbolic constant, we need to include it as a
variable, giving us a vector of loop indexes $[i, n]$. In matrix-vector form, this
iteration space is defined by

$$\left\{ i \text{ in } Z \ \middle| \ \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ n \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}.$$

Notice that, although the vector of array indexes has two dimensions, only the
first of these, representing i, is part of the output — the set of points lying with
the iteration space. □

11.3.5 Controlling the Order of Execution

The linear inequalities extracted from the lower and upper bounds of a loop
body define a set of iterations over a convex polyhedron. As such, the represen-
tation assumes no execution ordering between iterations within the iteration
space. The original program imposes one sequential order on the iterations,
which is the lexicographic order with respect to the loop index variables ordered
from the outermost to the innermost. However, the iterations in the space can
be executed in any order as long as their data dependences are honored (i.e.,

the order in which writes and reads of any array element are performed by the various assignment statements inside the loop nest do not change).

The problem of how we choose an ordering that honors the data dependences and optimizes for data locality and parallelism is hard and is dealt with later starting from Section 11.7. Here we assume that a legal and desirable ordering is given, and show how to generate code that enforce the ordering. Let us start by showing an alternative ordering for Example 11.6.

Example 11.10: There are no dependences between iterations in the program in Example 11.6. We can therefore execute the iterations in arbitrary order, sequentially or concurrently. Since iteration $[i, j]$ accesses element $Z[j, i]$ in the code, the original program visits the array in the order of Fig. 11.14(a). To improve spatial locality, we prefer to visit contiguous words in the array consecutively, as in Fig. 11.14(b).

This access pattern is obtained if we execute the iterations in the order shown in Fig. 11.14(c). That is, instead of sweeping the iteration space in Fig. 11.11 horizontally, we sweep the iteration space vertically, so j becomes the index of the outer loop. The code that executes the iterations in the above order is

```
for (j = 0; j <= 7; j++)
    for (i = 0; i <= min(5,j); i++)
        Z[j,i] = 0;
```

□

Given a convex polyhedron and an ordering of the index variables, how do we generate the loop bounds that sweep through the space in lexicographic order of the variables? In the example above, the constraint $i \leq j$ shows up as a lower bound for index j in the inner loop in the original program, but as an upper bound for index i, again in the inner loop, in the transformed program.

The bounds of the outermost loop, expressed as linear combinations of symbolic constants and constants, define the range of all the possible values it can take on. The bounds for inner loop variables are expressed as linear combinations of outer loop index variables, symbolic constants and constants. They define the range the variable can take on for each combination of values in outer loop variables.

Projection

Geometrically speaking, we can find the loop bounds of the outer loop index in a two-deep loop nest by *projecting* the convex polyhedron representing the iteration space onto the outer dimension of the space. The projection of a polyhedron on a lower-dimensional space is intuitively the shadow cast by the object onto that space. The projection of the two-dimensional iteration space in Fig. 11.11 onto the i axis is the vertical line from 0 to 5; and the projection onto

$Z[0,0]$, $Z[1,0]$, $Z[2,0]$, $Z[3,0]$, $Z[4,0]$, $Z[5,0]$, $Z[6,0]$, $Z[7,0]$
            $Z[1,1]$, $Z[2,1]$, $Z[3,1]$, $Z[4,1]$, $Z[5,1]$, $Z[6,1]$, $Z[1,7]$
                        $Z[2,2]$, $Z[3,2]$, $Z[4,2]$, $Z[5,2]$, $Z[6,2]$, $Z[7,2]$
                                    $Z[3,3]$, $Z[4,3]$, $Z[5,3]$, $Z[6,3]$, $Z[7,3]$
                                                $Z[4,4]$, $Z[5,4]$, $Z[6,4]$, $Z[7,4]$
                                                            $Z[5,5]$, $Z[6,5]$, $Z[7,5]$

(a) Original access order.

$Z[0,0]$
$Z[1,0]$, $Z[1,1]$
$Z[2,0]$, $Z[2,1]$, $Z[2,2]$
$Z[3,0]$, $Z[3,1]$, $Z[3,2]$, $Z[3,3]$
$Z[4,0]$, $Z[4,1]$, $Z[4,2]$, $Z[4,3]$, $Z[4,4]$
$Z[5,0]$, $Z[5,1]$, $Z[5,2]$, $Z[5,3]$, $Z[5,4]$, $Z[5,5]$
$Z[6,0]$, $Z[6,1]$, $Z[6,2]$, $Z[6,3]$, $Z[6,4]$, $Z[6,5]$
$Z[7,0]$, $Z[7,1]$, $Z[7,2]$, $Z[7,3]$, $Z[7,4]$, $Z[7,5]$

(b) Preferred order of access.

$[0,0]$
$[0,1]$, $[1,1]$
$[0,2]$, $[1,2]$, $[2,2]$
$[0,3]$, $[1,3]$, $[2,3]$, $[3,3]$
$[0,4]$, $[1,4]$, $[2,4]$, $[3,4]$, $[4,4]$
$[0,5]$, $[1,5]$, $[2,5]$, $[3,5]$, $[4,5]$, $[5,5]$
$[0,6]$, $[1,6]$, $[2,6]$, $[3,6]$, $[4,6]$, $[5,6]$
$[0,7]$, $[1,7]$, $[2,7]$, $[3,7]$, $[4,7]$, $[5,7]$

(c) Preferred order of iterations.

Figure 11.14: Reordering the accesses and iterations for a loop nest

the j axis is the horizontal line from 0 to 7. When we project a 3-dimensional object along the z axis onto a 2-dimensional x and y plane, we eliminate variable z, losing the height of the individual points and simply record the 2-dimensional footprint of the object in the x-y plane.

Loop bound generation is only one of the many uses of projection. Projection can be defined formally as follows. Let S be an n-dimensional polyhedron. The projection of S onto the first m of its dimensions is the set of points $(x_1, x_2, \ldots, x_m)$ such that for some $x_{m+1}, x_{m+2}, \ldots, x_n$, vector $[x_1, x_2, \ldots, x_n]$ is in S. We can compute projection using *Fourier-Motzkin elimination*, as follows:

Algorithm 11.11 : Fourier-Motzkin elimination.

INPUT: A polyhedron S with variables $x_1, x_2, \ldots, x_n$. That is, S is a set of linear constraints involving the variables x_i. One given variable x_m is specified to be the variable to be eliminated.

OUTPUT: A polyhedron S' with variables $x_1, \ldots, x_{m-1}, x_{m+1}, \ldots, x_n$ (i.e., all the variables of S except for x_m) that is the projection of S onto dimensions other than the mth.

METHOD: Let C be all the constraints in S involving x_m. Do the following:

1. For every pair of a lower bound and an upper bound on x_m in C, such as

$$L \leq \quad c_1 x_m$$
$$c_2 x_m \quad \leq U$$

   create the new constraint

$$c_2 L \leq c_1 U$$

   Note that c_1 and c_2 are integers, but L and U may be expressions with variables other than x_m.

2. If integers c_1 and c_2 have a common factor, divide both sides by that factor.

3. If the new constraint is not satisfiable, then there is no solution to S; i.e., the polyhedra S and S' are both empty spaces.

4. S' is the set of constraints $S - C$, plus all the constraints generated in step 2.

Note, incidentally, that if x_m has u lower bounds and v upper bounds, eliminating x_m produces up to uv inequalities, but no more. □

The constraints added in step (1) of Algorithm 11.11 correspond to the implications of constraints C on the remaining variables in the system. Therefore, there is a solution in S' if and only if there exists at least one corresponding

solution in S. Given a solution in S' the range of the corresponding x_m can be found by replacing all variables but x_m in the constraints C by their actual values.

Example 11.12: Consider the inequalities defining the iteration space in Fig. 11.11. Suppose we wish to use Fourier-Motzkin elimination to project the two-dimensional space away from the i dimension and onto the j dimension. There is one lower bound on i: $0 \leq i$ and two upper bounds: $i \leq j$ and $i \leq 5$. This generates two constraints: $0 \leq j$ and $0 \leq 5$. The latter is trivially true and can be ignored. The former gives the lower bound on j, and the original upper bound $j \leq 7$ gives the upper bound. □

Loop-Bounds Generation

Now that we have defined Fourier-Motzkin elimination, the algorithm to generate the loop bounds to iterate over a convex polyhedron (Algorithm 11.13) is straightforward. We compute the loop bounds in order, from the innermost to the outer loops. All the inequalities involving the innermost loop index variables are written as the variable's lower or upper bounds. We then project away the dimension representing the innermost loop and obtain a polyhedron with one fewer dimension. We repeat until the bounds for all the loop index variables are found.

Algorithm 11.13: Computing bounds for a given order of variables.

INPUT: A convex polyhedron S over variables $v_1, \ldots, v_n$.

OUTPUT: A set of lower bounds L_i and upper bounds U_i for each v_i, expressed only in terms of the v_j's, for $j < i$.

METHOD: The algorithm is described in Fig. 11.15. □

Example 11.14: We apply Algorithm 11.13 to generate the loop bounds that sweep the the iteration space of Fig. 11.11 vertically. The variables are ordered j, i. The algorithm generates these bounds:

$$
\begin{aligned}
L_i &: \quad 0 \\
U_i &: \quad 5, j \\
L_j &: \quad 0 \\
U_j &: \quad 7
\end{aligned}
$$

We need to satisfy all the constraints, thus the bound on i is $\min(5, j)$. There are no redundancies in this example. □

```
Sₙ = S; /* Use Algorithm 11.11 to find the bounds */
for ( i = n; i ≥ 1; i − − ) {
        L_vᵢ = all the lower bounds on vᵢ in Sᵢ;
        U_vᵢ = all the upper bounds on vᵢ in Sᵢ;
        S_{i−1} = Constraints returned by applying Algorithm 11.11
                    to eliminate vᵢ from the constraints Sᵢ;
}
/* Remove redundancies */
S′ = ∅;
for ( i = 1; i ≤ n; i + + ) {
        Remove any bounds in L_vᵢ and U_vᵢ implied by S′;
        Add the remaining constraints of L_vᵢ and U_vᵢ on vᵢ to S′;
}
```

Figure 11.15: Code to express variable bounds with respect to a given variable ordering

$$
\begin{array}{llllll}
[0,0], & [1,1], & [2,2], & [3,3], & [4,4], & [5,5] \\
[0,1], & [1,2], & [2,3], & [3,4], & [4,5], & [5,6] \\
[0,2], & [1,3], & [2,4], & [3,5], & [4,6], & [5,7] \\
[0,3], & [1,4], & [2,5], & [3,6], & [4,7] \\
[0,4], & [1,5], & [2,6], & [3,7] \\
[0,5], & [1,6], & [2,7] \\
[0,6], & [1,7] \\
[0,7]
\end{array}
$$

Figure 11.16: Diagonalwise ordering of the iteration space of Fig. 11.11

11.3.6 Changing Axes

Note that sweeping the iteration space horizontally and vertically, as discussed above, are just two of the most common ways of visiting the iteration space. There are many other possibilities; for example, we can sweep the iteration space in Example 11.6 diagonal by diagonal, as discussed below in Example 11.15.

Example 11.15 : We can sweep the iteration space shown in Fig. 11.11 diagonally using the order shown in Fig. 11.16. The difference between the coordinates j and i in each diagonal is a constant, starting with 0 and ending with 7. Thus, we define a new variable $k = j - i$ and sweep through the iteration space in lexicographic order with respect to k and j. Substituting $i = j - k$ in the inequalities we get:

$$
\begin{array}{ccc}
0 \leq & j - k & \leq 5 \\
j - k \leq & j & \leq 7
\end{array}
$$

To create the loop bounds for the order described above, we can apply Algorithm 11.13 to the above set of inequalities with variable ordering k, j.

$$\begin{array}{ll} L_j : & k \\ U_j : & 5+k, 7 \\ L_k : & 0 \\ U_k : & 7 \end{array}$$

From these inequalities, we generate the following code, replacing i by $j - k$ in array accesses.

```
for (k = 0; k <= 7; k++)
    for (j = k; j <= min(5+k,7); j++)
        Z[j,j-k] = 0;
```

□

In general, we can change the axes of a polyhedron by creating new loop index variables that represent affine combinations of the original variables, and defining an ordering on those variables. The hard problem lies in choosing the right axes to satisfy the data dependences while achieving the parallelism and locality objectives. We discuss this problem starting with Section 11.7. What we have established here is that once the axes are chosen, it is straightforward to generate the desired code, as shown in Example 11.15.

There are many other iteration-traversal orders not handled by this technique. For example, we may wish to visit all the odd rows in an iteration space before we visit the even rows. Or, we may want to start with the iterations in the middle of the iteration space and progress to the fringes. For applications that have affine access functions, however, the techniques described here cover most of the desirable iteration orderings.

11.3.7 Exercises for Section 11.3

Exercise 11.3.1 : Convert each of the following loops to a form where the loop indexes are each incremented by 1:

a) `for (i=10; i<50; i=i+7) X[i,i+1] = 0;`.

b) `for (i= -3; i<=10; i=i+2) X[i] = X[i+1];`.

c) `for (i=50; i>=10; i--) X[i] = 0;`.

Exercise 11.3.2 : Draw or describe the iteration spaces for each of the following loop nests:

a) The loop nest of Fig. 11.17(a).

b) The loop nest of Fig. 11.17(b).

```
for (i = 1; i < 30; i++)
    for (j = i+2; j < 40-i; j++)
        X[i,j] = 0;
```

(a) Loop nest for Exercise 11.3.2(a).

```
for (i = 10; i <= 1000; i++)
    for (j = i; j < i+10; j++)
        X[i,j] = 0;
```

(b) Loop nest for Exercise 11.3.2(b).

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100+i; j++)
        for (k = i+j; k < 100-i-j; k++)
            X[i,j,k] = 0;
```

(c) Loop nest for Exercise 11.3.2(c).

Figure 11.17: Loop nests for Exercise 11.3.2

c) The loop nest of Fig. 11.17(c).

Exercise 11.3.3 : Write the constraints implied by each of the loop nests of Fig. 11.17 in the form of (11.1). That is, give the values of the vectors **i** and **b** and the matrix **B**.

Exercise 11.3.4 : Reverse each of the loop-nesting orders for the nests of Fig. 11.17.

Exercise 11.3.5 : Use the Fourier-Motzkin elimination algorithm to eliminate i from each of the sets of constraints obtained in Exercise 11.3.3.

Exercise 11.3.6 : Use the Fourier-Motzkin elimination algorithm to eliminate j from each of the sets of constraints obtained in Exercise 11.3.3.

Exercise 11.3.7 : For each of the loop nests in Fig. 11.17, rewrite the code so the axis i is replaced by the major diagonal, i.e., the direction of the axis is characterized by $i = j$. The new axis should correspond to the outermost loop.

Exercise 11.3.8 : Repeat Exercise 11.3.7 for the following changes of axes:

a) Replace i by $i + j$; i.e., the direction of the axis is the lines for which $i + j$ is a constant. The new axis corresponds to the outermost loop.

b) Replace j by $i - 2j$. The new axis corresponds to the outermost loop.

! Exercise 11.3.9 : Let A, B, and C be integer constants in the following loop, with $C > 1$ and $B > A$:

```
for (i = A; i <= B; i = i + C)
    Z[i] = 0;
```

Rewrite the loop so the incrementation of the loop variable is 1 and the initialization is to 0, that is, to be of the form

```
for (j = 0; j <= D; j++)
    Z[E*j + F] = 0;
```

for integers D, E, and F. Express D, E, and F in terms of A, B, and C.

Exercise 11.3.10 : For a generic two-loop nest

```
for (i = 0; i <= A; i++)
    for(j = B*i+C; j <= D*i+E; j++)
```

with A through E integer constants, write the constraints that define the loop nest's iteration space in matrix-vector form, i.e., in the form $\mathbf{Bi} + \mathbf{b} = \mathbf{0}$.

Exercise 11.3.11 : Repeat Exercise 11.3.10 for a generic two-loop nest with symbolic integer constants m and n as in

```
for (i = 0; i <= m; i++)
    for(j = A*i+B; j <= C*i+n; j++)
```

As before, A, B, and C stand for specific integer constants. Only i, j, m, and n should be mentioned in the vector of unknowns. Also, remember that only i and j are output variables for the expression.

11.4 Affine Array Indexes

The focus of this chapter is on the class of affine array accesses, where each array index is expressed as affine expressions of loop indexes and symbolic constants. Affine functions provide a succinct mapping from the iteration space to the data space, making it easy to determine which iterations map to the same data or same cache line.

Just as the affine upper and lower bounds of a loop can be represented as a matrix-vector calculation, we can do the same for affine access functions. Once placed in the matrix-vector form, we can apply standard linear algebra to find pertinent information such as the dimensions of the data accessed, and which iterations refer to the same data.

11.4.1 Affine Accesses

We say that an array access in a loop is *affine* if

1. The bounds of the loop are expressed as affine expressions of the surrounding loop variables and symbolic constants, and

2. The index for each dimension of the array is also an affine expression of surrounding loop variables and symbolic constants.

Example 11.16: Suppose i and j are loop index variables bounded by affine expressions. Some examples of affine array accesses are $Z[i]$, $Z[i+j+1]$, $Z[0]$, $Z[i,i]$, and $Z[2*i+1, 3*j-10]$. If n is a symbolic constant for a loop nest, then $Z[3*n, n-j]$ is another example of an affine array access. However, $Z[i*j]$ and $Z[n*j]$ are not affine accesses. □

Each affine array access can be described by two matrices and two vectors. The first matrix-vector pair is the **B** and **b** that describe the iteration space for the access, as in the inequality of Equation (11.1). The second pair, which we usually refer to as **F** and **f**, represent the function(s) of the loop-index variables that produce the array index(es) used in the various dimensions of the array access.

Formally, we represent an array access in a loop nest that uses a vector of index variables **i** by the four-tuple $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$; it maps a vector **i** within the bounds

$$\mathbf{Bi} + \mathbf{b} \geq \mathbf{0}$$

to the array element location

$$\mathbf{Fi} + \mathbf{f}$$

Example 11.17: In Fig. 11.18 are some common array accesses, expressed in matrix notation. The two loop indexes are i and j, and these form the vector **i**. Also, X, Y, and Z are arrays with 1, 2, and 3 dimensions, respectively.

The first access, $X[i-1]$, is represented by a 1×2 matrix **F** and a vector **f** of length 1. Notice that when we perform the matrix-vector multiplication and add in the vector **f**, we are left with a single function, $i-1$, which is exactly the formula for the access to the one-dimensional array X. Also notice the third access, $Y[j, j+1]$, which, after matrix-vector multiplication and addition, yields a pair of functions, $(j, j+1)$. These are the indexes of the two dimensions of the array access.

Finally, let us observe the fourth access $Y[1, 2]$. This access is a constant, and unsurprisingly the matrix **F** is all 0's. Thus, the vector of loop indexes, **i**, does not appear in the access function. □

ACCESS	AFFINE EXPRESSION
X[i-1]	$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}$
Y[i,j]	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
Y[j,j+1]	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
Y[1,2]	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$
Z[1,i,2*i+j]	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

Figure 11.18: Some array accesses and their matrix-vector representations

11.4.2 Affine and Nonaffine Accesses in Practice

There are certain common data access patterns found in numerical programs that fail to be affine. Programs involving sparse matrices are one important example. One popular representation for sparse matrices is to store only the nonzero elements in a vector, and auxiliary index arrays are used to mark where a row starts and which columns contain nonzeros. Indirect array accesses are used in accessing such data. An access of this type, such as $X[Y[i]]$, is a nonaffine access to the array X. If the sparsity is regular, as in banded matrices having nonzeros only around the diagonal, then dense arrays can be used to represent the subregions with nonzero elements. In that case, accesses may be affine.

Another common example of nonaffine accesses is linearized arrays. Programmers sometimes use a linear array to store a logically multidimensional object. One reason why this is the case is that the dimensions of the array may not be known at compile time. An access that would normally look like $Z[i,j]$ would be expressed as $Z[i*n+j]$, which is a quadratic function. We can convert the linear access into a multidimensional access if every access can

be decomposed into separate dimensions with the guarantee that none of the components exceeds its bound. Finally, we note that induction-variable analyses can be used to convert some nonaffine accesses into affine ones, as shown in Example 11.18.

Example 11.18: We can rewrite the code

```
j = n;
for (i = 0; i <= n; i++) {
    Z[j] = 0;
    j = j+2;
}
```

as

```
j = n;
for (i = 0; i <= n; i++) {
    Z[n+2*i] = 0;
}
```

to make the access to matrix Z affine. □

11.4.3 Exercises for Section 11.4

Exercise 11.4.1: For each of the following array accesses, give the vector **f** and the matrix **F** that describe them. Assume that the vector of indexes **i** is $i, j, \ldots$, and that all loop indexes have affine limits.

  a) $X[2 * i + 3, 2 * j - i]$.

  b) $Y[i - j, j - k, k - i]$.

  c) $Z[3, 2 * j, k - 2 * i + 1]$.

11.5 Data Reuse

From array access functions we derive two kinds of information useful for locality optimization and parallelization:

  1. *Data reuse*: for locality optimization, we wish to identify sets of iterations that access the same data or the same cache line.

  2. *Data dependence*: for correctness of parallelization and locality loop transformations, we wish to identify *all* the data dependences in the code. Recall that two (not necessarily distinct) accesses have a data dependence if instances of the accesses may refer to the same memory location, and at least one of them is a write.

In many cases, whenever we identify iterations that reuse the same data, there are data dependences between them.

Whenever there is a data dependence, obviously the same data is reused. For example, in matrix multiplication, the same element in the output array is written $O(n)$ times. The write operations must be executed in the original execution order;[3] we can exploit the reuse by allocating a register to hold one element of the output array while it is being computed.

However, not all reuse can be exploited in locality optimizations; here is an example illustrating this issue.

Example 11.19: Consider the following loop:

```
for (i = 0; i < n; i++)
    Z[7*i+3] = Z[3*i+5];
```

We observe that the loop writes to a different location at each iteration, so there are no reuses or dependences on the different write operations. The loop, however, reads locations $5, 8, 11, 14, 17, \ldots$, and writes locations $3, 10, 17, 24, \ldots$. The read and write iterations access the same elements 17, 38, and 59 and so on. That is, the integers of the form $17 + 21j$ for $j = 0, 1, 2, \ldots$ are all those integers that can be written both as $7i_1 + 3$ and as $3i_2 + 5$, for some integers i_1 and i_2. However, this reuse occurs rarely, and cannot be exploited easily if at all. □

Data dependence is different from reuse analysis in that one of the accesses sharing a data dependence must be a write access. More importantly, data dependence needs to be both correct and precise. It needs to find all dependences for correctness, and it should not find spurious dependences because they can cause unnecessary serialization.

With data reuse, we only need to find where most of the exploitable reuses are. This problem is much simpler, so we take up this topic here in this section and tackle data dependences in the next. We simplify reuse analysis by ignoring loop bounds, because they seldom change the shape of the reuse. Much of the reuse exploitable by affine partitioning resides among instances of the same array accesses, and accesses that share the same *coefficient matrix* (what we have typically called **F** in the affine index function). As shown above, access patterns like $7i + 3$ and $3i + 5$ have no reuse of interest.

11.5.1 Types of Reuse

We first start with Example 11.20 to illustrate the different kinds of data reuses. In the following, we need to distinguish between the access as an instruction in

---

[3] There is a subtle point here. Because of the commutativity of addition, we would get the same answer to the sum regardless of the order in which we performed the sum. However, this case is very special. In general, it is far too complex for the compiler to determine what computation is being performed by a sequence of arithmetic steps followed by writes, and we cannot rely on there being any algebraic rules that will help us reorder the steps safely.

a program, e.g., `x = Z[i,j]`, from the execution of this instruction many times, as we execute the loop nest. For emphasis, we may refer to the statement itself as a *static access*, while the various iterations of the statement as we execute its loop nest are called *dynamic accesses*.

Reuses can be classified as *self* versus *group*. If iterations reusing the same data come from the same static access, we refer to the reuse as self reuse; if they come from different accesses, we refer to it as group reuse. The reuse is *temporal* if the same exact location is referenced; it is *spatial* if the same cache line is referenced.

Example 11.20: Consider the following loop nest:

```
float Z[n];
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        Z[j+1] = (Z[j] + Z[j+1] + Z[j+2])/3;
```

Accesses $Z[j]$, $Z[j + 1]$, and $Z[j + 2]$ each have self-spatial reuse because consecutive iterations of the same access refer to contiguous array elements. Presumably contiguous elements are very likely to reside on the same cache line. In addition, they all have self-temporal reuse, since the exact elements are used over and over again in each iteration in the outer loop. In addition, they all have the same coefficient matrix, and thus have group reuse. There is group reuse, both temporal and spatial, between the different accesses. Although there are $4n^2$ accesses in this code, if the reuse can be exploited, we only need to bring in about n/c cache lines into the cache, where c is the number of words in a cache line. We drop a factor of n due to self-spatial reuse, a factor of c to due to spatial locality, and finally a factor of 4 due to group reuse. $\square$

In the following, we show how we can use linear algebra to extract the reuse information from affine array accesses. We are interested in not just finding how much potential savings there are, but also which iterations are reusing the data so that we can try to move them close together to exploit the reuse.

11.5.2 Self Reuse

There can be substantial savings in memory accesses by exploiting self reuse. If the data referenced by a static access has k dimensions and the access is nested in a loop d deep, for some $d > k$, then the same data can be reused n^{d-k} times, where n is the number of iterations in each loop. For example, if a 3-deep loop nest accesses one column of an array, then there is a potential savings factor of n^2 accesses. It turns out that the dimensionality of an access corresponds to the concept of the *rank* of the coefficient matrix in the access, and we can find which iterations refer to the same location by finding the *null space* of the matrix, as explained below.

Rank of a Matrix

The rank of a matrix $\mathbf{F}$ is the largest number of columns (or equivalently, rows) of $\mathbf{F}$ that are linearly independent. A set of vectors is *linearly independent* if none of the vectors can be written as a linear combination of finitely many other vectors in the set.

Example 11.21: Consider the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

Notice that the second row is the sum of the first and third rows, while the fourth row is the third row minus twice the first row. However, the first and third rows are linearly independent; neither is a multiple of the other. Thus, the rank of the matrix is 2.

We could also draw this conclusion by examining the columns. The third column is twice the second column minus the first column. On the other hand, any two columns are linearly independent. Again, we conclude that the rank is 2. □

Example 11.22: Let us look at the array accesses in Fig. 11.18. The first access, $X[i-1]$, has dimension 1, because the rank of the matrix [1 0] is 1. That is, the one row is linearly independent, as is the first column.

The second access, $Y[i, j]$, has dimension 2. The reason is that the matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

has two independent rows (and therefore two independent columns, of course). The third access, $Y[j, j+1]$, is of dimension 1, because the matrix

$$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$

has rank 1. Note that the two rows are identical, so only one is linearly independent. Equivalently, the first column is 0 times the second column, so the columns are not independent. Intuitively, in a large, square array Y, the only elements accessed lie along a one-dimensional line, just above the main diagonal.

The fourth access, $Y[1, 2]$ has dimension 0, because a matrix of all 0's has rank 0. Note that for such a matrix, we cannot find a linear sum of even one row that is nonzero. Finally, the last access, $Z[i, i, 2 * i + j]$, has dimension 2. Note that in the matrix for this access

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

the last two rows are linearly independent; neither is a multiple of the other. However, the first row is a linear "sum" of the other two rows, with both coefficients 0.   □

Null Space of a Matrix

A reference in a d-deep loop nest with rank r accesses $O(n^r)$ data elements in $O(n^d)$ iterations, so on average, $O(n^{d-r})$ iterations must refer to the same array element. Which iterations access the same data? Suppose an access in this loop nest is represented by matrix-vector combination $\mathbf{F}$ and $\mathbf{f}$. Let $\mathbf{i}$ and $\mathbf{i}'$ be two iterations that refer to the same array element. Then $\mathbf{Fi} + \mathbf{f} = \mathbf{Fi}' + \mathbf{f}$. Rearranging terms, we get

$$\mathbf{F}(\mathbf{i} - \mathbf{i}') = \mathbf{0}.$$

There is a well-known concept from linear algebra that characterizes when $\mathbf{i}$ and $\mathbf{i}'$ satisfy the above equation. The set of all solutions to the equation $\mathbf{Fv} = \mathbf{0}$ is called the *null space* of $\mathbf{F}$. Thus, two iterations refer to the same array element if the difference of their loop-index vectors belongs to the null space of matrix $\mathbf{F}$.

It is easy to see that the null vector, $\mathbf{v} = \mathbf{0}$, always satisfies $\mathbf{Fv} = \mathbf{0}$. That is, two iterations surely refer to the same array element if their difference is $\mathbf{0}$; in other words, if they are really the same iteration. Also, the null space is truly a vector space. That is, if $\mathbf{Fv}_1 = \mathbf{0}$ and $\mathbf{Fv}_2 = \mathbf{0}$, then $\mathbf{F}(\mathbf{v}_1 + \mathbf{v}_2) = \mathbf{0}$ and $\mathbf{F}(c\mathbf{v}_1) = \mathbf{0}$.

If the matrix $\mathbf{F}$ is *fully ranked*, that is, its rank is d, then the null space of $\mathbf{F}$ consists of only the null vector. In that case, iterations in a loop nest all refer to different data. In general, the dimension of the null space, also known as the *nullity*, is $d - r$. If $d > r$, then for each element there is a $(d - r)$-dimensional space of iterations that access that element.

The null space can be represented by its basis vectors. A k-dimensional null space is represented by k independent vectors; any vector that can be expressed as a linear combination of the basis vectors belongs to the null space.

Example 11.23: Let us reconsider the matrix of Example 11.21:

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

We determined in that example that the rank of the matrix is 2; thus the nullity is $3 - 2 = 1$. To find a basis for the null space, which in this case must be a single nonzero vector of length 3, we may suppose a vector in the null space to

be $[x, y, z]$ and try to solve the equation

$$
\begin{bmatrix}
1 & 2 & 3 \\
5 & 7 & 9 \\
4 & 5 & 6 \\
2 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix}
=
\begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix}.
$$

If we multiply the first two rows by the vector of unknowns, we get the two equations

$$
\begin{aligned}
x + 2y + 3z &= 0 \\
5x + 7y + 9z &= 0
\end{aligned}
$$

We could write the equations that come from the third and fourth rows as well, but because there are no three linearly independent rows, we know that the additional equations add no new constraints on x, y, and z. For instance, the equation we get from the third row, $4x + 5y + 6z = 0$ can be obtained by subtracting the first equation from the second.

We must eliminate as many variables as we can from the above equations. Start by using the first equation to solve for x; that is, $x = -2y - 3z$. Then substitute for x in the second equation, to get $-3y = 6z$, or $y = -2z$. Since $x = -2y - 3z$, and $y = -2z$, it follows that $x = z$. Thus, the vector $[x, y, z]$ is really $[z, -2z, z]$. We may pick any nonzero value of z to form the one and only basis vector for the null space. For example, we may choose $z = 1$ and use $[1, -2, 1]$ as the basis of the null space. $\quad\Box$

Example 11.24: The rank, nullity, and null space for each of the references in Example 11.17 are shown in Fig. 11.19. Observe that the sum of the rank and nullity in all the cases is the depth of the loop nest, 2. Since the accesses $Y[i, j]$ and $Z[1, i, 2 * i + j]$ have a rank of 2, all iterations refer to different locations.

Accesses $X[i-1]$ and $Y[j, j+1]$ both have rank-1 matrices, so $O(n)$ iterations refer to the same location. In the former case, entire rows in the iteration space refer to the same location. In other words, iterations that differ only in the j dimension share the same location, which is succinctly represented by the basis of the null space, [0,1]. For $Y[j, j + 1]$, entire columns in the iteration space refer to the same location, and this fact is succinctly represented by the basis of the null space, [1,0].

Finally, the access $Y[1, 2]$ refers to the same location in all the iterations. The null space corresponding has 2 basis vectors, [0,1], [1,0], meaning that all pairs of iterations in the loop nest refer to exactly the same location. $\quad\Box$

11.5.3 Self-Spatial Reuse

The analysis of spatial reuse depends on the data layout of the matrix. C matrices are laid out in row-major order and Fortran matrices are laid out in column-major order. In other words, array elements $X[i, j]$ and $X[i, j + 1]$ are

ACCESS	AFFINE EXPRESSION	RANK	NULL-ITY	BASIS OF NULL SPACE
X[i-1]	$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}$	1	1	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$
Y[i,j]	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	2	0	
Y[j,j+1]	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	1	1	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$
Y[1,2]	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$	0	2	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
Z[1,i,2*i+j]	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$	2	0	

Figure 11.19: Rank and nullity of affine accesses

contiguous in C and $X[i, j]$ and $X[i + 1, j]$ are contiguous in Fortran. Without loss of generality, in the rest of the chapter, we shall adopt the C (row-major) array layout.

As a first approximation, we consider two array elements to share the same cache line if and only if they share the same row in a two-dimensional array. More generally, in an array of d dimensions, we take array elements to share a cache line if they differ only in the last dimension. Since for a typical array and cache, many array elements can fit in one cache line, there is significant speedup to be had by accessing an entire row in order, even though, strictly speaking, we occasionally have to wait to load a new cache line.

The trick to discovering and taking advantage of self-spatial reuse is to drop the last row from the coefficient matrix **F**. If the resulting *truncated* matrix has rank that is less than the depth of the loop nest, then we can assure spatial locality by making sure that the innermost loop varies only the last coordinate of the array.

Example 11.25: Consider the last access, $Z[1, i, 2 * i + j]$, in Fig. 11.19. If we

delete the last row, we are left with the truncated matrix

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

The rank of this matrix is evidently 1, and since the loop nest has depth 2, there is the opportunity for spatial reuse. In this case, since j is the inner-loop index, the inner loop visits contiguous elements of the array Z stored in row-major order. Making i the inner-loop index will not yield spatial locality, since as i changes, both the second and third dimensions change. □

The general rule for determining whether there is self-spatial reuse is as follows. As always, we assume that the loop indexes correspond to columns of the coefficient matrix in order, with the outermost loop first, and the innermost loop last. Then in order for there to be spatial reuse, the vector $[0, 0, \ldots, 0, 1]$ must be in the null space of the truncated matrix. The reason is that if this vector is in the null space, then when we fix all loop indexes but the innermost one, we know that all dynamic accesses during one run through the inner loop vary in only the last array index. If the array is stored in row-major order, then these elements are all near one another, perhaps in the same cache line.

Example 11.26 : Note that $[0, 1]$ (transposed as a column vector) is in the null space of the truncated matrix of Example 11.25. Thus, as mentioned there, we expect that with j as the inner-loop index, there will be spatial locality. On the other hand, if we reverse the order of the loops, so i is the inner loop, then the coefficient matrix becomes

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Now, $[0, 1]$ is not in the null space of this matrix. Rather, the null space is generated by the basis vector $[1, 0]$. Thus, as we suggested in Example 11.25, we do not expect spatial locality if i is the inner loop.

We should observe, however, that the test for $[0, 0, \ldots, 0, 1]$ being in the null space is not quite sufficient to assure spatial locality. For instance, suppose the access were not $Z[1, i, 2 * i + j]$ but $Z[1, i, 2 * i + 50 * j]$. Then, only every fiftieth element of Z would be accessed during one run of the inner loop, and we would not reuse a cache line unless it were long enough to hold more than 50 elements. □

11.5.4 Group Reuse

We compute group reuse only among accesses in a loop sharing the same coefficient matrix. Given two dynamic accesses $\mathbf{F}\mathbf{i}_1 + \mathbf{f}_1$ and $\mathbf{F}\mathbf{i}_2 + \mathbf{f}_2$, reuse of the same data requires that

$$\mathbf{F}\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}\mathbf{i}_2 + \mathbf{f}_2$$

or

$$\mathbf{F}(\mathbf{i}_1 - \mathbf{i}_2) = (\mathbf{f}_2 - \mathbf{f}_1).$$

Suppose $\mathbf{v}$ is one solution to this equation. Then if $\mathbf{w}$ is any vector in the null space of $\mathbf{F}$, $\mathbf{w} + \mathbf{v}$ is also a solution, and in fact those are all the solutions to the equation.

Example 11.27: The following 2-deep loop nest

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        Z[i,j] = Z[i-1,j];
```

has two array accesses, $Z[i, j]$ and $Z[i - 1, j]$. Observe that these two accesses are both characterized by the coefficient matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

like the second access, $Y[i, j]$ in Fig. 11.19. This matrix has rank 2, so there is no self-temporal reuse.

However, each access exhibits self-spatial reuse. As described in Section 11.5.3, when we delete the bottom row of the matrix, we are left with only the top row, $[1, 0]$, which has rank 1. Since $[0, 1]$ is in the null space of this truncated matrix, we expect spatial reuse. As each incrementation of inner-loop index j increases the second array index by one, we in fact do access adjacent array elements, and will make maximum use of each cache line.

Although there is no self-temporal reuse for either access, observe that the two references $Z[i, j]$ and $Z[i-1, j]$ access almost the same set of array elements. That is, there is group-temporal reuse because the data read by access $Z[i-1, j]$ is the same as the data written by access $Z[i, j]$, except for the case $i = 1$. This simple pattern applies to the entire iteration space and can be exploited to improve data locality in the code. Formally, discounting the loop bounds, the two accesses $Z[i, j]$ and $Z[i-1, j]$ refer to the same location in iterations (i_1, j_1) and (i_2, j_2), respectively, provided

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

Rewriting the terms, we get

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

That is, $j_1 = j_2$ and $i_2 = i_1 + 1$.

Notice that the reuse occurs along the i-axis of the iteration space. That is, the iteration (i_2, j_2) occurs n iterations (of the inner loop) after the iteration

(i_1, j_1). Thus, many iterations are executed before the data written is reused. This data may or may not still be in the cache. If the cache manages to hold two consecutive rows of matrix Z, then access $Z[i-1, j]$ does not miss in the cache, and the total number of cache misses for the entire loop nest is n^2/c, where c is the number of elements per cache line. Otherwise, there will be twice as many misses, since both static accesses require a new cache line for each c dynamic accesses. $\square$

Example 11.28 : Suppose there are two accesses

$$A[i, j, i+j] \text{ and } A[i+1, j-1, i+j]$$

in a 3-deep loop nest, with indexes i, j, and k, from the outer to the inner loop. Then two accesses $\mathbf{i}_1 = [i_1, j_1, k_1]$ and $\mathbf{i}_2 = [i_2, j_2, k_2]$ reuse the same element whenever

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}.$$

One solution to this equation for a vector $\mathbf{v} = [i_1 - i_2, j_1 - j_2, k_1 - k_2]$ is $\mathbf{v} = [1, -1, 0]$; that is, $i_1 = i_2 + 1$, $j_1 = j_2 - 1$, and $k_1 = k_2$.[4] However, the null space of the matrix

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

is generated by the basis vector $[0, 0, 1]$; that is, the third loop index, k, can be arbitrary. Thus, $\mathbf{v}$, the solution to the above equation, is any vector $[1, -1, m]$ for some m. Put another way, a dynamic access to $A[i, j, i+j]$, in a loop nest with indexes i, j, and k, is reused not only by other dynamic accesses $A[i, j, i+j]$ with the same values of i and j and a different value of k, but also by dynamic accesses $A[i+1, j-1, i+j]$ with loop index values $i+1$, $j-1$, and any value of k. $\square$

Although we shall not do so here, we can reason about group-spatial reuse analogously. As per the discussion of self-spatial reuse, we simply drop the last dimension from consideration.

The extent of reuse is different for the different categories of reuse. Self-temporal reuse gives the most benefit: a reference with a k-dimensional null space reuses the same data $O(n^k)$ times. The extent of self-spatial reuse is limited by the length of the cache line. Finally, the extent of group reuse is limited by the number of references in a group sharing the reuse.

---

[4] It is interesting to observe that, although there is a solution in this case, there would be no solution if we changed one of the third components from $i+j$ to $i+j+1$. That is, in the example as given, both accesses touch those array elements that lie in the 2-dimensional subspace S defined by "the third component is the sum of the first two components." If we changed $i+j$ to $i+j+1$, none of the elements touched by the second access would lie in S, and there would be no reuse at all.

11.5.5 Exercises for Section 11.5

Exercise 11.5.1: Compute the ranks of each of the matrices in Fig. 11.20. Give both a maximal set of linearly independent columns and a maximal set of linearly independent rows.

$$
\begin{bmatrix}
0 & 1 & 5 \\
1 & 2 & 6 \\
2 & 3 & 7 \\
3 & 4 & 8
\end{bmatrix}
\qquad
\begin{bmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 12 & 15 \\
3 & 2 & 2 & 3
\end{bmatrix}
\qquad
\begin{bmatrix}
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 1 & 1 \\
5 & 6 & 3
\end{bmatrix}
$$

(a) \qquad\qquad\qquad (b) \qquad\qquad\qquad (c)

Figure 11.20: Compute the ranks and null spaces of these matrices

Exercise 11.5.2: Find a basis for the null space of each matrix in Fig. 11.20.

Exercise 11.5.3: Assume that the iteration space has dimensions (variables) i, j, and k. For each of the accesses below, describe the subspaces that refer to the following single elements of the array:

a) $A[i, j, i + j]$

b) $A[i, i + 1, i + 2]$

! c) $A[i, i, j + k]$

! d) $A[i - j, j - k, k - i]$

! **Exercise 11.5.4:** Suppose array A is stored in row-major order and accessed inside the following loop nest:

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
        for (k = 0; k < 100; k++)
            <some access to A>
```

Indicate for each of the following accesses whether it is possible to rewrite the loops so that the access to A exhibits self-spatial reuse; that is, entire cache lines are used consecutively. Show how to rewrite the loops, if so. Note: the rewriting of the loops may involve both reordering and introduction of new loop indexes. However, you may not change the layout of the array, e.g., by changing it to column-major order. Also note: in general, reordering of loop indexes may be legal or illegal, depending on criteria we develop in the next section. However, in this case, where the effect of each access is simply to set an array element to 0, you do not have to worry about the effect of reordering loops as far as the semantics of the program is concerned.

  a) `A[i+1,i+k,j] = 0.`

!! b) `A[j+k,i,i] = 0.`

  c) `A[i,j,k,i+j+k] = 0.`

!! d) `A[i,j-k,i+j,i+k] = 0.`

Exercise 11.5.5: In Section 11.5.3 we commented that we get spatial locality if the innermost loop varies only as the last coordinate of an array access. However, that assertion depended on our assumption that the array was stored in row-major order. What condition would assure spatial locality if the array were stored in column-major order?

! **Exercise 11.5.6:** In Example 11.28 we observed that the existence of reuse between two similar accesses depended heavily on the particular expressions for the coordinates of the array. Generalize our observation there to determine for which functions $f(i, j)$ there is reuse between the accesses $A[i, j, i + j]$ and $A[i + 1, j - 1, f(i, j)]$.

! **Exercise 11.5.7:** In Example 11.27 we suggested that there will be more cache misses than necessary if rows of the matrix Z are so long that they do not fit in the cache. If that is the case, how could you rewrite the loop nest in order to guarantee group-spatial reuse?

11.6 Array Data-Dependence Analysis

Parallelization or locality optimizations frequently reorder the operations executed in the original program. As with all optimizations, operations can be reordered only if the reordering does not change the program's output. Since we cannot, in general, understand deeply what a program does, code optimization generally adopts a simpler, conservative test for when we can be sure that the program output is not affected: we check that the operations on any memory location are done in the same order in the original and modified programs. In the present study, we focus on array accesses, so the array elements are the memory locations of concern.

Two accesses, whether read or write, are clearly *independent* (can be reordered) if they refer to two different locations. In addition, read operations do not change the memory state and therefore are also independent. Following Section 11.5, we say that two accesses are *data dependent* if they refer to the same memory location and at least one of them is a write operation. To be sure that the modified program does the same as the original, the relative execution ordering between every pair of data-dependent operations in the original program must be preserved in the new program.

Recall from Section 10.2.1 that there are three flavors of data dependence:

1. *True dependence*, where a write is followed by a read of the same location.

2. *Antidependence*, where a read is followed by a write to the same location.

3. *Output dependence*, which is two writes to the same location.

In the discussion above, data dependence is defined for dynamic accesses. We say that a static access in a program depends on another as long as there exists a dynamic instance of the first access that depends on some instance of the second.[5]

It is easy to see how data dependence can be used in parallelization. For example, if no data dependences are found in the accesses of a loop, we can easily assign each iteration to a different processor. Section 11.7 discusses how we use this information systematically in parallelization.

11.6.1 Definition of Data Dependence of Array Accesses

Let us consider two static accesses to the same array in possibly different loops. The first is represented by access function and bounds $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$ and is in a d-deep loop nest; the second is represented by $\mathcal{F}' = \langle \mathbf{F}', \mathbf{f}', \mathbf{B}', \mathbf{b}' \rangle$ and is in a d'-deep loop nest. These accesses are data dependent if

1. At least one of them is a write reference and

2. There exist vectors $\mathbf{i}$ in Z^d and $\mathbf{i}'$ in $Z^{d'}$ such that

   (a) $\mathbf{Bi} \geq \mathbf{0}$,
   (b) $\mathbf{B}'\mathbf{i}' \geq \mathbf{0}$, and
   (c) $\mathbf{Fi} + \mathbf{f} = \mathbf{F}'\mathbf{i}' + \mathbf{f}'$.

Since a static access normally embodies many dynamic accesses, it is also meaningful to ask if its dynamic accesses may refer to the same memory location. To search for dependencies between instances of the same static access, we assume $\mathcal{F} = \mathcal{F}'$ and augment the definition above with the additional constraint that $\mathbf{i} \neq \mathbf{i}'$ to rule out the trivial solution.

Example 11.29: Consider the following 1-deep loop nest:

```
for (i = 1; i <= 10; i++) {
    Z[i] = Z[i-1];
}
```

This loop has two accesses: $Z[i-1]$ and $Z[i]$; the first is a read reference and the second a write. To find all the data dependences in this program, we need to check if the write reference shares a dependence with itself and with the read reference:

---

[5] Recall the difference between static and dynamic accesses. A static access is an array reference at a particular location in a program, while a dynamic access is one execution of that reference.

1. *Data dependence between* Z[i − 1] *and* Z[i]. Except for the first iteration, each iteration reads the value written in the previous iteration. Mathematically, we know that there is a dependence because there exist integers i and i' such that

$$1 \le i \le 10,\ 1 \le i' \le 10,\ \text{and}\ i - 1 = i'.$$

   There are nine solutions to the above system of constraints: $(i = 2, i' = 1)$, $(i = 3, i' = 2)$, and so forth.

2. *Data dependence between* Z[i] *and itself.* It is easy to see that different iterations in the loop write to different locations; that is, there are no data dependencies among the instances of the write reference $Z[i]$. Mathematically, we know that there does not exist a dependence because there do not exist integers i and i' satisfying

$$1 \le i \le 10,\ 1 \le i' \le 10,\ i = i',\ \text{and}\ i \ne i'.$$

   Notice that the third condition, $i = i'$, comes from the requirement that $Z[i]$ and $Z[i']$ are the same memory location. The contradictory fourth condition, $i \ne i'$, comes from the requirement that the dependence be nontrivial — between different dynamic accesses.

It is not necessary to consider data dependences between the read reference $Z[i − 1]$ and itself because any two read accesses are independent. □

11.6.2 Integer Linear Programming

Data dependence requires finding whether there exist integers that satisfy a system consisting of equalities and inequalities. The equalities are derived from the matrices and vectors representing the accesses; the inequalities are derived from the loop bounds. Equalities can be expressed as inequalities: an equality $x = y$ can be replaced by two inequalities, $x \ge y$ and $y \ge x$.

Thus, data dependence may be phrased as a search for integer solutions that satisfy a set of linear inequalities, which is precisely the well-known problem of *integer linear programming*. Integer linear programming is an NP-complete problem. While no polynomial algorithm is known, heuristics have been developed to solve linear programs involving many variables, and they can be quite fast in many cases. Unfortunately, such standard heuristics are inappropriate for data dependence analysis, where the challenge is to solve many small and simple integer linear programs rather than large complicated integer linear programs.

The data dependence analysis algorithm consists of three parts:

1. Apply the GCD (Greatest Common Divisor) test, which checks if there is an integer solution to the equalities, using the theory of linear Diophantine equations. If there are no integer solutions, then there are no data dependences. Otherwise, we use the equalities to substitute for some of the variables thereby obtaining simpler inequalities.

2. Use a set of simple heuristics to handle the large numbers of typical inequalities.

3. In the rare case where the heuristics do not work, we use a linear integer programming solver that uses a branch-and-bound approach based on Fourier-Motzkin elimination.

11.6.3   The GCD Test

The first subproblem is to check for the existence of integer solutions to the equalities. Equations with the stipulation that solutions must be integers are known as *Diophantine equations*. The following example shows how the issue of integer solutions arises; it also demonstrates that even though many of our examples involve a single loop nest at a time, the data dependence formulation applies to accesses in possibly different loops.

Example 11.30: Consider the following code fragment:

```
for (i = 1; i < 10; i++) {
    Z[2*i] = 10;
}
for (j = 1; j < 10; j++) {
    Z[2*j+1] = 20;
}
```

The access $Z[2 * i]$ only touches even elements of Z, while access $Z[2 * j + 1]$ touches only odd elements. Clearly, these two accesses share no data dependence regardless of the loop bounds. We can execute iterations in the second loop before the first, or interleave the iterations. This example is not as contrived as it may look. An example where even and odd numbers are treated differently is an array of complex numbers, where the real and imaginary components are laid out side by side.

To prove the absence of data dependences in this example, we reason as follows. Suppose there were integers i and j such that $Z[2 * i]$ and $Z[2 * j + 1]$ are the same array element. We get the Diophantine equation

$$2i = 2j + 1.$$

There are no integers i and j that can satisfy the above equation. The proof is that if i is an integer, then $2i$ is even. If j is an integer, then $2j$ is even, so $2j + 1$ is odd. No even number is also an odd number. Therefore, the equation

has no integer solutions, and thus there is no dependence between the read and write accesses. $\quad\Box$

To describe when there is a solution to a linear Diophantine equation, we need the concept of the *greatest common divisor* (GCD) of two or more integers. The GCD of integers $a_1, a_2, \ldots, a_n$, denoted $\gcd(a_1, a_2, \ldots, a_n)$, is the largest integer that evenly divides all these integers. GCD's can be computed efficiently by the well-known Euclidean algorithm (see the box on the subject).

Example 11.31: $\gcd(24, 36, 54) = 6$, because $24/6$, $36/6$, and $54/6$ each have remainder 0, yet any integer larger than 6 must leave a nonzero remainder when dividing at least one of 24, 36, and 54. For instance, 12 divides 24 and 36 evenly, but not 54. $\quad\Box$

The importance of the GCD is in the following theorem.

Theorem 11.32: The linear Diophantine equation

$$a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = c$$

has an integer solution for $x_1, x_2, \ldots, x_n$ if and only if $\gcd(a_1, a_2, \ldots, a_n)$ divides c. $\quad\Box$

Example 11.33: We observed in Example 11.30 that the linear Diophantine equation $2i = 2j + 1$ has no solution. We can write this equation as

$$2i - 2j = 1.$$

Now $\gcd(2, -2) = 2$, and 2 does not divide 1 evenly. Thus, there is no solution.
For another example, consider the equation

$$24x + 36y + 54z = 30.$$

Since $\gcd(24, 36, 54) = 6$, and $30/6 = 5$, there is a solution in integers for x, y, and z. One solution is $x = -1$, $y = 0$, and $z = 1$, but there are an infinity of other solutions. $\quad\Box$

The first step to the data dependence problem is to use a standard method such as Gaussian elimination to solve the given equalities. Every time a linear equation is constructed, apply Theorem 11.32 to rule out, if possible, the existence of an integer solution. If we can rule out such solutions, then answer "no". Otherwise, we use the solution of the equations to reduce the number of variables in the inequalities.

Example 11.34: Consider the two equalities

$$\begin{array}{rcl} x - 2y + z & = & 0 \\ 3x + 2y + z & = & 5 \end{array}$$

The Euclidean Algorithm

The *Euclidean algorithm* for finding $\gcd(a, b)$ works as follows. First, assume that a and b are positive integers, and $a \geq b$. Note that the GCD of negative numbers, or the GCD of a negative and a positive number is the same as the GCD of their absolute values, so we can assume all integers are positive.

If $a = b$, then $\gcd(a, b) = a$. If $a > b$, let c be the remainder of a/b. If $c = 0$, then b evenly divides a, so $\gcd(a, b) = b$. Otherwise, compute $\gcd(b, c)$; this result will also be $\gcd(a, b)$.

To compute $\gcd(a_1, a_2, \ldots, a_n)$, for $n > 2$, use the Euclidean algorithm to compute $\gcd(a_1, a_2) = c$. Then recursively compute $\gcd(c, a_3, a_4, \ldots, a_n)$.

Looking at each equality by itself, it appears there might be a solution. For the first equality, $\gcd(1, -2, 1) = 1$ divides 0, and for the second equality, $\gcd(3, 2, 1) = 1$ divides 5. However, if we use the first equality to solve for $z = 2y - x$ and substitute for z in the second equality, we get $2x + 4y = 5$. This Diophantine equation has no solution, since $\gcd(2, 4) = 2$ does not divide 5 evenly. □

11.6.4 Heuristics for Solving Integer Linear Programs

The data dependence problem requires many simple integer linear programs be solved. We now discuss several techniques to handle simple inequalities and a technique to take advantage of the similarity found in data dependence analysis.

Independent-Variables Test

Many of the integer linear programs from data dependence consist of inequalities that involve only one unknown. The programs can be solved simply by testing if there are integers between the constant upper bounds and constant lower bounds independently.

Example 11.35: Consider the nested loop

```
for (i = 0; i <= 10; i++)
    for (j = 0; j <= 10; j++)
        Z[i,j] = Z[j+10,i+11];
```

To find if there is a data dependence between $Z[i, j]$ and $Z[j + 10, i + 11]$, we ask if there exist integers i, j, i', and j' such that

$$0 \leq i, j, i', j' \leq 10$$
$$i = j' + 10$$
$$j = i' + 11$$

The GCD test, applied to the two equalities above, will determine that there *may* be an integer solution. The integer solutions to the equalities are expressed by

$$i = t_1, j = t_2, i' = t_2 - 11, j' = t_1 - 10$$

for any integers t_1 and t_2. Substituting the variables t_1 and t_2 into the linear inequalities, we get

$$
\begin{array}{rcl}
0 \leq & t_1 & \leq 10 \\
0 \leq & t_2 & \leq 10 \\
0 \leq & t_2 - 11 & \leq 10 \\
0 \leq & t_1 - 10 & \leq 10
\end{array}
$$

Thus, combining the lower bounds from the last two inequalities with the upper bounds from the first two, we deduce

$$10 \leq t_1 \leq 10$$
$$11 \leq t_2 \leq 10$$

Since the lower bound on t_2 is greater than its upper bound, there is no integer solution, and hence no data dependence. This example shows that even if there are equalities involving several variables, the GCD test may still create linear inequalities that involve one variable at a time. □

Acyclic Test

Another simple heuristic is to find if there exists a variable that is bounded below or above by a constant. In certain circumstances, we can safely replace the variable by the constant; the simplified inequalities have a solution if and only if the original inequalities have a solution. Specifically, suppose every lower bound on v_i is of the form

$$c_0 \leq c_i v_i \text{ for some } c_i > 0$$

while the upper bounds on v_i are all of the form

$$c_i v_i \leq c_0 + c_1 v_1 + \ldots + c_{i-1} v_{i-1} + c_{i+1} v_{i+1} + \ldots + c_n v_n$$

where c_i is nonnegative. Then we can replace variable v_i by its smallest possible integer value. If there is no such lower bound, we simply replace v_i with $-\infty$.

Similarly, if every constraint involving v_i can be expressed in the two forms above, but with the directions of the inequalities reversed, then we can replace variable v_i with the largest possible integer value, or by ∞ if there is no constant upper bound. This step can be repeated to simplify the inequalities and in some cases determine if there is a solution.

Example 11.36 : Consider the following inequalities:

$$
\begin{array}{rcccl}
1 & \leq & v_1, v_2 & \leq & 10 \\
0 & \leq & v_3 & \leq & 4 \\
v_2 & \leq & v_1 & & \\
& & v_1 & \leq & v_3 + 4
\end{array}
$$

Variable v_1 is bounded from below by v_2 and from above by $v_3 + 4$. However, v_2 is bounded from below only by the constant 1, and v_3 is bounded from above only by the constant 4. Thus, replacing v_2 by 1 and v_3 by 4 in the inequalities, we obtain

$$
\begin{array}{rcccl}
1 & \leq & v_1 & \leq & 10 \\
1 & \leq & v_1 & & \\
& & v_1 & \leq & 8
\end{array}
$$

which can now be solved easily with the independent-variables test. $\Box$

The Loop-Residue Test

Let us now consider the case where every variable is bounded from below and above by other variables. It is commonly the case in data dependence analysis that constraints have the form $v_i \leq v_j + c$, which can be solved using a simplified version of the *loop-residue test* due to Shostak. A set of these constraints can be represented by a directed graph whose nodes are labeled with variables. There is an edge from v_i to v_j labeled c whenever there is a constraint $v_i \leq v_j + c$.

We define the *weight* of a path to be the sum of the labels of all the edges along the path. Each path in the graph represents a combination of the constraints in the system. That is, we can infer that $v \leq v' + c$ whenever there exists a path from v to v' with weight c. A cycle in the graph with weight c represents the constraint $v \leq v + c$ for each node v on the cycle. If we can find a negatively weighted cycle in the graph, then we can infer $v < v$, which is impossible. In this case, we can conclude that there is no solution and thus no dependence.

We can also incorporate into the loop-residue test constraints of the form $c \leq v$ and $v \leq c$ for variable v and constant c. We introduce into the system of inequalities a new dummy variable v_0, which is added to each constant upper and lower bound. Of course, v_0 must have value 0, but since the loop-residue test only looks for cycles, the actual values of the variables never becomes significant. To handle constant bounds, we replace

$$v \leq c \text{ by } v \leq v_0 + c$$
$$c \leq v \text{ by } v_0 \leq v - c.$$

Example 11.37: Consider the inequalities

$$
\begin{array}{ccccc}
1 & \leq & v_1, v_2 & \leq & 10 \\
0 & \leq & v_3 & \leq & 4 \\
v_2 & \leq & v_1 & & \\
& & 2v_1 & \leq & 2v_3 - 7
\end{array}
$$

The constant upper and lower bounds on v_1 become $v_0 \leq v_1 - 1$ and $v_1 \leq v_0 + 10$; the constant bounds on v_2 and v_3 are handled similarly. Then, converting the last constraint to $v_1 \leq v_3 - 4$, we can create the graph shown in Fig. 11.21. The cycle v_1, v_3, v_0, v_1 has weight -1, so there is no solution to this set of inequalities. □
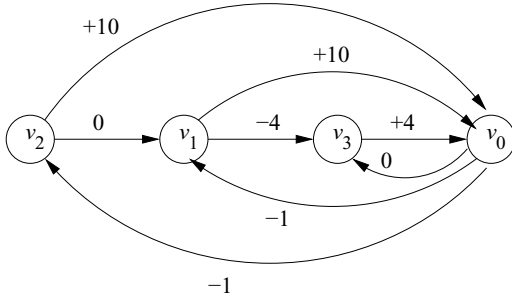
Figure 11.21: Graph for the constraints of Example 11.37

Memoization

Often, similar data dependence problems are solved repeatedly, because simple access patterns are repeated throughout the program. One important technique to speed up data dependence processing is to use *memoization*. Memoization tabulates the results to the problems as they are generated. The table of stored solutions is consulted as each problem is presented; the problem needs to be solved only if the result to the problem cannot be found in the table.

11.6.5 Solving General Integer Linear Programs

We now describe a general approach to solving the integer linear programming problem. The problem is NP-complete; our algorithm uses a branch-and-bound approach that can take an exponential amount of time in the worst case. However, it is rare that the heuristics of Section 11.6.4 cannot resolve the problem, and even if we do need to apply the algorithm of this section, it seldom needs to perform the branch-and-bound step.

The approach is to first check for the existence of rational solutions to the inequalities. This problem is the classical linear-programming problem. If there is no rational solution to the inequalities, then the regions of data touched by the accesses in question do not overlap, and there surely is no data dependence. If there is a rational solution, we first try to prove that there is an integer solution, which is commonly the case. Failing that, we then split the polyhedron bounded by the inequalities into two smaller problems and recurse.

Example 11.38: Consider the following simple loop:

```
for (i = 1; i < 10; i++)
    Z[i] = Z[i+10];
```

The elements touched by access $Z[i]$ are $Z[1], \ldots, Z[9]$, while the elements touched by $Z[i + 10]$ are $Z[11], \ldots, Z[19]$. The ranges do not overlap and therefore there are no data dependences. More formally, we need to show that there are no two dynamic accesses i and i', with $1 \leq i \leq 9$, $1 \leq i' \leq 9$, and $i = i' + 10$. If there were such integers i and i', then we could substitute $i' + 10$ for i and get the four constraints on i': $1 \leq i' \leq 9$ and $1 \leq i' + 10 \leq 9$. However, $i' + 10 \leq 9$ implies $i' \leq -1$, which contradicts $1 \leq i'$. Thus, no such integers i and i' exist. □

Algorithm 11.39 describes how to determine if an integer solution can be found for a set of linear inequalities based on the Fourier-Motzkin elimination algorithm.

Algorithm 11.39: Branch-and-bound solution to integer linear programming problems.

INPUT: A convex polyhedron S_n over variables $v_1, \ldots, v_n$.

OUTPUT: "yes" if S_n has an integer solution, "no" otherwise.

METHOD: The algorithm is shown in Fig. 11.22. □

Lines (1) through (3) attempt to find a rational solution to the inequalities. If there is no rational solution, there is no integer solution. If a rational solution is found, this means that the inequalities define a nonempty polyhedron. It is relatively rare for such a polyhedron not to include any integer solutions — for that to happen, the polyhedron must be relatively thin along some dimension and fit between integer points.

Thus, lines (4) through (9) try to check quickly if there is an integer solution. Each step of the Fourier-Motzkin elimination algorithm produces a polyhedron with one fewer dimension than the previous one. We consider the polyhedra in reverse order. We start with the polyhedron with one variable and assign to that variable an integer solution roughly in the middle of the range of possible values if possible. We then substitute the value for the variable in all other polyhedra, decreasing their unknown variables by one. We repeat the same process until

1)   apply Algorithm 11.11 to S_n to project away variables
        $v_n, v_{n-1}, \ldots, v_1$ in that order;
2)   let S_i be the polyhedron after projecting away v_{i+1}, for
        $i = n - 1, n - 2, \ldots, 0$;
3)   **if** S_0 is empty **return** "no";
        /* There is no rational solution if S_0, which involves
        only constants, has unsatisfiable constraints */
4)   **for** $(i = 1; i \leq n; i{+}{+})$ {
5)   **if** (S_i does not include an integer value) **break**;
6)   pick c_i, an integer in the middle of the range for v_i in S_i;
7)   modify S_i by replacing v_i by c_i;
8)   }
9)   **if** $(i == n + 1)$ **return** "yes";
10)   **if** $(i == 1)$ **return** "no";
11)   let the lower and upper bounds on v_i in S_i be
        l_i and u_i, respectively;
12)   recursively apply this algorithm to $S_n \cup \{v_i \leq \lfloor l_i \rfloor\}$ and
        $$S_n \cup \{v_i \geq \lceil u_i \rceil\};$$
13)   **if** (either returns "yes") **return** "yes" **else return** "no";

Figure 11.22: Finding an integer solution in inequalities

we have processed all the polyhedra, in which case an integer solution is found, or we have found a variable for which there is no integer solution.

If we cannot find an integer value for even the first variable, there is no integer solution (line 10). Otherwise, all we know is that there is no integer solution including the combination of specific integers we have picked so far, and the result is inconclusive. Lines (11) through (13) represent the branch-and-bound step. If variable v_i is found to have a rational but not integer solution, we split the polyhedron into two with the first requiring that v_i must be an integer smaller than the rational solution found, and the second requiring that v_i must be an integer greater than the rational solution found. If neither has a solution, then there is no dependence.

11.6.6   Summary

We have shown that essential pieces of information that a compiler can glean from array references are equivalent to certain standard mathematical concepts. Given an access function $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$:

1. The dimension of the data region accessed is given by the rank of the matrix $\mathbf{F}$. The dimension of the space of accesses to the same location is given by the nullity of $\mathbf{F}$. Iterations whose differences belong to the null space of $\mathbf{F}$ refer to the same array elements.

2. Iterations that share self-temporal reuse of an access are separated by vectors in the null space of $\mathbf{F}$. Self-spatial reuse can be computed similarly by asking when two iterations use the same row, rather than the same element. Two accesses $\mathbf{F}\mathbf{i}_1 + \mathbf{f}_1$ and $\mathbf{F}\mathbf{i}_2 + \mathbf{f}_2$ share easily exploitable locality along the $\mathbf{d}$ direction, if $\mathbf{d}$ is the particular solution to the equation $\mathbf{F}\mathbf{d} = (\mathbf{f}_1 - \mathbf{f}_2)$. In particular, if $\mathbf{d}$ is the direction corresponding to the innermost loop, i.e., the vector $[0, 0, \ldots, 0, 1]$, then there is spatial locality if the array is stored in row-major form.

3. The data dependence problem — whether two references can refer to the same location — is equivalent to integer linear programming. Two access functions share a data dependence if there are integer-valued vectors $\mathbf{i}$ and $\mathbf{i}'$ such that $\mathbf{B}\mathbf{i} \geq \mathbf{0}$, $\mathbf{B}'\mathbf{i}' \geq \mathbf{0}$, and $\mathbf{F}\mathbf{i} + \mathbf{f} = \mathbf{F}'\mathbf{i}' + \mathbf{f}'$.

11.6.7 Exercises for Section 11.6

Exercise 11.6.1 : Find the GCD's of the following sets of integers:

a) $\{16, 24, 56\}$.

b) $\{-45, 105, 240\}$.

! c) $\{84, 105, 180, 315, 350\}$.

Exercise 11.6.2 : For the following loop

```
for (i = 0; i < 10; i++)
    A[i] = A[10-i];
```

indicate all the

a) True dependences (write followed by read of the same location).

b) Antidependences (read followed by write to the same location).

c) Output dependences (write followed by another write to the same location).

! **Exercise 11.6.3 :** In the box on the Euclidean algorithm, we made a number of assertions without proof. Prove each of the following:

a) The Euclidean algorithm as stated always works. In particular, $\gcd(b, c) = \gcd(a, b)$, where c is the nonzero remainder of a/b.

b) $\gcd(a, b) = \gcd(a, -b)$.

c) $\gcd(a_1, a_2, \ldots, a_n) = \gcd(\gcd(a_1, a_2), a_3, a_4, \ldots, a_n)$ for $n > 2$.

d) The GCD is really a function on sets of integers; i.e., order doesn't matter. Show the *commutative law* for GCD: $\gcd(a, b) = \gcd(b, a)$. Then, show the more difficult statement, the *associative law* for GCD: $\gcd\big(\gcd(a, b), c\big) = \gcd\big(a, \gcd(b, c)\big)$. Finally, show that together these laws imply that the GCD of a set of integers is the same, regardless of the order in which the GCD's of pairs of integers are computed.

e) If S and T are sets of integers, then $\gcd(S \cup T) = \gcd\big(\gcd(S), \gcd(T)\big)$.

! **Exercise 11.6.4:** Find another solution to the second Diophantine equation in Example 11.33.

Exercise 11.6.5: Apply the independent-variables test in the following situation. The loop nest is

```
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        for (k=0; k<100; k++)
```

and inside the nest is an assignment involving array accesses. Determine if there are any data dependences due to each of the following statements:

a) `A[i,j,k] = A[i+100,j+100,k+100]`.

b) `A[i,j,k] = A[j+100,k+100,i+100]`.

c) `A[i,j,k] = A[j-50,k-50,i-50]`.

d) `A[i,j,k] = A[i+99,k+100,j]`.

Exercise 11.6.6: In the two constraints

$$
\begin{array}{rcl}
1 & \le & x & \le y - 100 \\
3 & \le & x & \le 2y - 50
\end{array}
$$

eliminate x by replacing it by a constant lower bound on y.

Exercise 11.6.7: Apply the loop-residue test to the following set of constraints:

$$
\begin{array}{ll}
0 \le x \le 99 & y \le x - 50 \\
0 \le y \le 99 & z \le y - 60 \\
0 \le z \le 99 &
\end{array}
$$

Exercise 11.6.8: Apply the loop-residue test to the following set of constraints:

$$
\begin{array}{ll}
0 \le x \le 99 & y \le x - 50 \\
0 \le y \le 99 & z \le y + 40 \\
0 \le z \le 99 & x \le z + 20
\end{array}
$$

Exercise 11.6.9 : Apply the loop-residue test to the following set of constraints:

$$0 \le x \le 99 \quad y \le x - 100$$
$$0 \le y \le 99 \quad z \le y + 60$$
$$0 \le z \le 99 \quad x \le z + 50$$

11.7   Finding Synchronization-Free Parallelism

Having developed the theory of affine array accesses, their reuse of data, and the dependences among them, we shall now begin to apply this theory to parallelization and optimization of real programs. As discussed in Section 11.1.4, it is important that we find parallelism while minimizing communication among processors. Let us start by studying the problem of parallelizing an application without allowing any communication or synchronization between processors at all. This constraint may appear to be a purely academic exercise; how often can we find programs and routines that have such a form of parallelism? In fact, many such programs exist in real life, and the algorithm for solving this problem is useful in its own right. In addition, the concepts used to solve this problem can be extended to handle synchronization and communication.

11.7.1   An Introductory Example

Shown in Fig. 11.23 is an excerpt of a C translation (with Fortran-style array accesses retained for clarity) from a 5000-line Fortran multigrid algorithm to solve three-dimensional Euler equations. The program spends most its time in a small number of routines like the one shown in the figure. It is typical of many numerical programs. These often consist of numerous for-loops, with different nesting levels, and they have many array accesses, all of which are affine expressions of surrounding loop indexes. To keep the example short, we have elided lines from the original program with similar characteristics.

The code of Fig. 11.23 operates on the scalar variable T and a number of different arrays with different dimensions. Let us first examine the use of variable T. Because each iteration in the loop uses the same variable T, we cannot execute the iterations in parallel. However, T is used only as a way to hold a common subexpression used twice in the same iteration. In the first two of the three loop nests in Fig. 11.23, each iteration of the innermost loop writes a value into T and uses the value immediately after twice, in the same iteration. We can eliminate the dependences by replacing each use of T by the right-hand-side expression in the previous assignment of T, without changing the semantics of the program. Or, we can replace the scalar T by an array. We then have each iteration (j, i) use its own array element $T[j, i]$.

With this modification, the computation of an array element in each assignment statement depends only on other array elements with the same values for the last two components (j and i, respectively). We can thus group all

```
for (j = 2; j <= jl; j++)
   for (i = 2, i <= il, i++) {
      AP[j,i]     = ...;
      T           = 1.0/(1.0 + AP[j,i]);
      D[2,j,i]    = T*AP[j,i];
      DW[1,2,j,i] = T*DW[1,2,j,i];
   }
for (k = 3; k <= kl-1; k++)
   for (j = 2; j <= jl; j++)
      for (i = 2; i <= il; i++) {
         AM[j,i]      = AP[j,i];
         AP[j,i]      = ...;
         T            = ...AP[j,i] - AM[j,i]*D[k-1,j,i]...;
         D[k,j,i]     = T*AP[j,i];
         DW[1,k,j,i] = T*(DW[1,k,j,i] + DW[1,k-1,j,i])...;
      }
...
for (k = kl-1; k >= 2; k--)
   for (j = 2; j <= jl; j++)
      for (i = 2; i <= il; i++)
         DW[1,k,j,i] = DW[1,k,j,i] + D[k,j,i]*DW[1,k+1,j,i];
```

Figure 11.23: Code excerpt of a multigrid algorithm

operations that operate on the (j, i)th element of all arrays into one computation unit, and execute them in the original sequential order. This modification produces $(jl - 1) \times (il - 1)$ units of computation that are all independent of one another. Notice that second and third nests in the source program involve a third loop, with index k. However, because there is no dependence between dynamic accesses with the same values for j and i, we can safely perform the loops on k inside the loops on j and i — that is, within a computation unit.

Knowing that these computation units are independent enables a number of legal transforms on this code. For example, instead of executing the code as originally written, a uniprocessor can perform the same computation by executing the units of independent operation one unit at a time. The resulting code, shown in Fig. 11.24, has improved temporal locality, because results produced are consumed immediately.

The independent units of computation can also be assigned to different processors and executed in parallel, without requiring any synchronization or communication. Since there are $(jl - 1) \times (il - 1)$ independent units of computation, we can utilize at most $(jl - 1) \times (il - 1)$ processors. By organizing the processors as if they were in a 2-dimensional array, with ID's (j, i), where $2 \leq j < jl$ and $2 \leq i < il$, the SPMD program to be executed by each processor is simply the body in the inner loop in Fig. 11.24.

```
for (j = 2; j <= jl; j++)
   for (i = 2; i <= il; i++) {
      AP[j,i]        = ...;
      T[j,i]         = 1.0/(1.0 + AP[j,i]);
      D[2,j,i]       = T[j,i]*AP[j,i];
      DW[1,2,j,i]    = T[j,i]*DW[1,2,j,i];
      for (k = 3; k <= kl-1; k++) {
         AM[j,i]        = AP[j,i];
         AP[j,i]        = ...;
         T[j,i]         = ...AP[j,i] - AM[j,i]*D[k-1,j,i]...;
         D[k,j,i]       = T[j,i]*AP[j,i];
         DW[1,k,j,i]    = T[j,i]*(DW[1,k,j,i] + DW[1,k-1,j,i])...;
      }
      ...
      for (k = kl-1; k >= 2; k--)
         DW[1,k,j,i] = DW[1,k,j,i] + D[k,j,i]*DW[1,k+1,j,i];
   }
```

Figure 11.24: Code of Fig. 11.23 transformed to carry outermost parallel loops

The above example illustrates the basic approach to finding synchronization-free parallelism. We first split the computation into as many independent units as possible. This partitioning exposes the scheduling choices available. We then assign computation units to the processors, depending on the number of processors we have. Finally, we generate an SPMD program that is executed on each processor.

11.7.2   Affine Space Partitions

A loop nest is said to have k degrees of parallelism if it has, within the nest, k parallelizable loops — that is, loops such that there are no data dependencies between different iterations of the loops. For example, the code in Fig. 11.24 has 2 degrees of parallelism. It is convenient to assign the operations in a computation with k degrees of parallelism to a processor array with k dimensions.

We shall assume initially that each dimension of the processor array has as many processors as there are iterations of the corresponding loop. After all the independent computation units have been found, we shall map these "virtual" processors to the actual processors. In practice, each processor should be responsible for a fairly large number of iterations, because otherwise there is not enough work to amortize away the overhead of parallelization.

We break down the program to be parallelized into elementary statements, such as 3-address statements. For each statement, we find an *affine space partition* that maps each dynamic instance of the statement, as identified by its loop indexes, to a processor ID.

Example 11.40 : As discussed above, the code of Fig. 11.24 has two degrees of parallelism. We view the processor array as a 2-dimensional space. Let (p_1, p_2) be the ID of a processor in the array. The parallelization scheme discussed in Section 11.7.1 can be described by simple affine partition functions. All the statements in the first loop nest have this same affine partition:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

All the statements in the second and third loop nests have the following same affine partition:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k \\ j \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

□

The algorithm to find synchronization-free parallelism consists of three steps:

1. Find, for each statement in the program, an affine partition that maximizes the degree of parallelism. Note that we generally treat the statement, rather than the single access, as the unit of computation. The same affine partition must apply to each access in the statement. This grouping of accesses makes sense, since there is almost always dependence among accesses of the same statement anyway.

2. Assign the resulting independent computation units among the processors, and choose an interleaving of the steps on each processor. This assignment is driven by locality considerations.

3. Generate an SPMD program to be executed on each processor.

We shall discuss next how to find the affine partition functions, how to generate a sequential program that executes the partitions serially, and how to generate an SPMD program that executes each partition on a different processor. After we discuss how parallelism with synchronizations is handled in Sections 11.8 through 11.9.9, we return to Step 2 above in Section 11.10 and discuss the optimization of locality for uniprocessors and multiprocessors.

11.7.3 Space-Partition Constraints

To require no communication, each pair of operations that share a data dependence must be assigned to the same processor. We refer to these constraints as "space-partition constraints." Any mapping that satisfies these constraints creates partitions that are independent of one another. Note that such constraints can be satisfied by putting all the operations in a single partition. Unfortunately, that "solution" does not yield any parallelism. Our goal is to create

as many independent partitions as possible while satisfying the space-partition constraints; that is, operations are not placed on the same processor unless it is necessary.

When we restrict ourselves to affine partitions, then instead of maximizing the number of independent units, we may maximize the degree (number of dimensions) of parallelism. It is sometimes possible to create more independent units if we can use *piecewise* affine partitions. A piecewise affine partition divides instances of a single access into different sets and allows a different affine partition for each set. However, we shall not consider such an option here.

Formally, an affine partition of a program is *synchronization free* if and only if for every two (not necessarily distinct) accesses sharing a dependence, $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ in statement s_1 nested in d_1 loops, and $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ in statement s_2 nested in d_2 loops, the partitions $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ and $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ for statements s_1 and s_2, respectively, satisfy the following *space-partition constraints*:

- For all $\mathbf{i}_1$ in Z^{d_1} and $\mathbf{i}_2$ in Z^{d_2} such that

    a) $\mathbf{B}_1 \mathbf{i}_1 + \mathbf{b}_1 \geq \mathbf{0}$,

    b) $\mathbf{B}_2 \mathbf{i}_2 + \mathbf{b}_2 \geq \mathbf{0}$, and

    c) $\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$,

    it is the case that $\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2$.

The goal of the parallelization algorithm is to find, for each statement, the partition with the highest rank that satisfies these constraints.

Shown in Fig. 11.25 is a diagram illustrating the essence of the space-partition constraints. Suppose there are two static accesses in two loop nests with index vectors $\mathbf{i}_1$ and $\mathbf{i}_2$. These accesses are dependent in the sense that they access at least one array element in common, and at least one of them is a write. The figure shows particular dynamic accesses in the two loops that happen to access the same array element, according to the affine access functions $\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1$ and $\mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$. Synchronization is necessary unless the affine partitions for the two static accesses, $\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1$ and $\mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2$, assign the dynamic accesses to the same processor.

If we choose an affine partition whose rank is the maximum of the ranks of all statements, we get the maximum possible parallelism. However, under this partitioning some processors may be idle at times, while other processors are executing statements whose affine partitions have a smaller rank. This situation may be acceptable if the time taken to execute those statements is relatively short. Otherwise, we can choose an affine partition whose rank is smaller than the maximum possible, as long as that rank is greater than 0.

We show in Example 11.41 a small program designed to illustrate the power of the technique. Real applications are usually much simpler than this, but may have boundary conditions resembling some of the issues shown here. We shall use this example throughout this chapter to illustrate that programs with
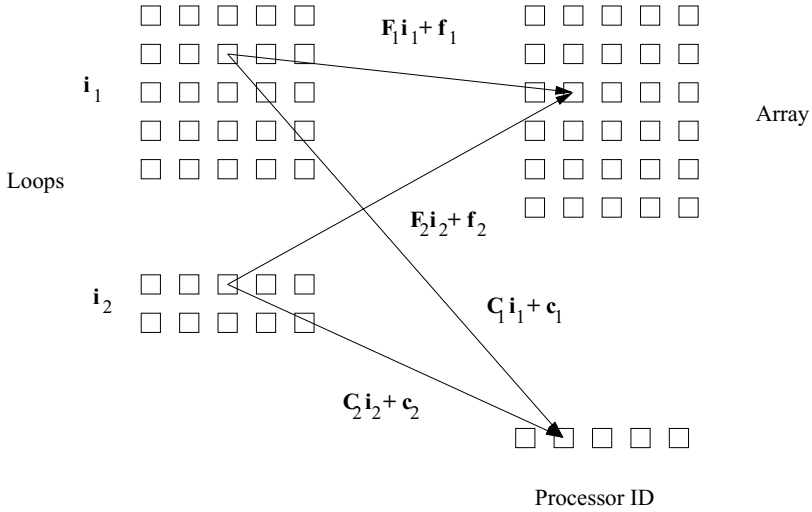
Figure 11.25: Space-partition constraints

affine accesses have relatively simple space-partition constraints, that these constraints can be solved using standard linear algebra techniques, and that the desired SPMD program can be generated mechanically from the affine partitions.

Example 11.41 : This example shows how we formulate the space-partition constraints for the program consisting of the small loop nest with two statements, s_1 and s_2, shown in Figure 11.26.

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */
        Y[i,j] = Y[i,j] + X[i,j-1];  /* (s2) */
    }
```

Figure 11.26: A loop nest exhibiting long chains of dependent operations

We show the data dependences in the program in Figure 11.27. That is, each black dot represents an instance of statement s_1, and each white dot represents an instance of statement s_2. The dot located at coordinates (i, j) represents the instance of the statement that is executed for those values of the loop indexes. Note, however, that the instance of s_2 is located just below the instance of s_1 for the same (i, j) pair, so the vertical scale of j is greater than the horizontal scale of i.

Notice that $X[i, j]$ is written by $s_1(i, j)$, that is, by the instance of statement s_1 with index values i and j. It is later read by $s_2(i, j + 1)$, so $s_1(i, j)$ must
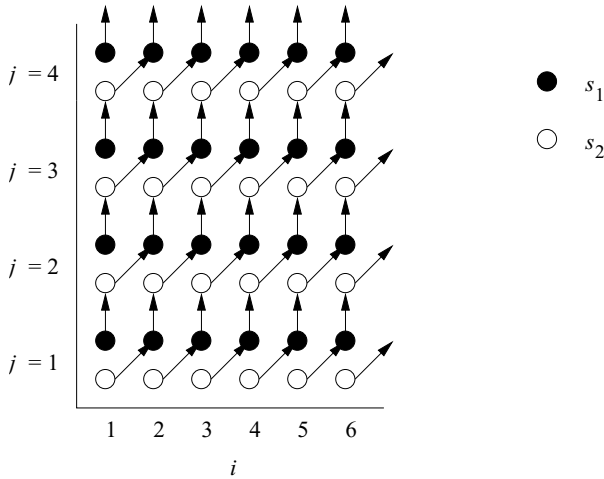
Figure 11.27: Dependences of the code in Example 11.41

precede $s_2(i, j + 1)$. This observation explains the vertical arrows from black dots to white dots. Similarly, $Y[i, j]$ is written by $s_2(i, j)$ and later read by $s_1(i + 1, j)$. Thus, $s_2(i, j)$ must precede $s_1(i + 1, j)$, which explains the arrows from white dots to black.

It is easy to see from this diagram that this code can be parallelized without synchronization by assigning each chain of dependent operations to the same processor. However, it is not easy to write the SPMD program that implements this mapping scheme. While the loops in the original program have 100 iterations each, there are 200 chains, with half originating and ending with statement s_1 and the other half originating and ending with s_2. The lengths of the chains vary from 1 to 100 iterations.

Since there are two statements, we are seeking two affine partitions, one for each statement. We only need to express the space-partition constraints for one-dimensional affine partitions. These constraints will be used later by the solution method that tries to find all the independent one-dimensional affine partitions and combine them to get multidimensional affine partitions. We can thus represent the affine partition for each statement by a 1×2 matrix and a 1×1 vector to translate the vector of indexes $[i, j]$ into a single processor number. Let $\langle [C_{11} C_{12}], [c_1] \rangle, \langle [C_{21} C_{22}], [c_2] \rangle$, be the one-dimensional affine partitions for the statements s_1 and s_2, respectively.

We apply six data dependence tests:

1. Write access $X[i, j]$ and itself in statement s_1,

2. Write access $X[i, j]$ with read access $X[i, j]$ in statement s_1,

3. Write access $X[i, j]$ in statement s_1 with read access $X[i, j - 1]$ in statement s_2,

4. Write access $Y[i, j]$ and itself in statement s_2,

5. Write access $Y[i, j]$ with read access $Y[i, j]$ in statement s_2,

6. Write access $Y[i, j]$ in statement s_2 with read access $Y[i-1, j]$ in statement s_1.

We see that the dependence tests are all simple and highly repetitive. The only dependences present in this code occur in case (3) between instances of accesses $X[i, j]$ and $X[i, j-1]$ and in case (6) between $Y[i, j]$ and $Y[i-1, j]$.

The space-partition constraints imposed by the data dependence between $X[i, j]$ in statement s_1 and $X[i, j-1]$ in statement s_2 can be expressed in the following terms:

For all (i, j) and (i', j') such that

$$
\begin{array}{ll}
1 \le i \le 100 & 1 \le j \le 100 \\
1 \le i' \le 100 & 1 \le j' \le 100 \\
i = i' & j = j' - 1
\end{array}
$$

we have

$$
\left[\begin{array}{cc} C_{11} & C_{12} \end{array}\right] \left[\begin{array}{c} i \\ j \end{array}\right] + \left[\begin{array}{c} c_1 \end{array}\right] = \left[\begin{array}{cc} C_{21} & C_{22} \end{array}\right] \left[\begin{array}{c} i' \\ j' \end{array}\right] + \left[\begin{array}{c} c_2 \end{array}\right]
$$

That is, the first four conditions say that (i, j) and (i', j') lie within the iteration space of the loop nest, and the last two conditions say that the dynamic accesses $X[i, j]$ and $X[i, j-1]$ touch the same array element. We can derive the space-partition constraint for accesses $Y[i-1, j]$ in statement s_2 and $Y[i, j]$ in statement s_1 in a similar manner. $\quad\Box$

11.7.4 Solving Space-Partition Constraints

Once the space-partition constraints have been extracted, standard linear algebra techniques can be used to find the affine partitions satisfying the constraints. Let us first show how we find the solution to Example 11.41.

Example 11.42: We can find the affine partitions for Example 11.41 with the following steps:

1. Create the space-partition constraints shown in Example 11.41. We use the loop bounds in determining the data dependences, but they are not used in the rest of the algorithm otherwise.

2. The unknown variables in the equalities are i, i', j, j', C_{11}, C_{12}, c_1, C_{21}, C_{22}, and c_2. Reduce the number of unknowns by using the equalities due to the access functions: $i = i'$ and $j = j' - 1$. We do so using Gaussian elimination, which reduces the four variables to two: say $t_1 = i = i'$, and $t_2 = j = j' - 1$. The equality for the partition becomes

$$
\left[\begin{array}{cc} C_{11} - C_{21} & C_{12} - C_{22} \end{array}\right] \left[\begin{array}{c} t_1 \\ t_2 \end{array}\right] + \left[\begin{array}{c} c_1 - c_2 - C_{22} \end{array}\right] = 0
$$

3. The equation above holds for all combinations of t_1 and t_2. Thus, it must be that

$$
\begin{aligned}
C_{11} - C_{21} &= 0 \\
C_{12} - C_{22} &= 0 \\
c_1 - c_2 - C_{22} &= 0
\end{aligned}
$$

If we perform the same steps on the constraint between the accesses $Y[i-1,j]$ and $Y[i,j]$, we get

$$
\begin{aligned}
C_{11} - C_{21} &= 0 \\
C_{12} - C_{22} &= 0 \\
c_1 - c_2 + C_{21} &= 0
\end{aligned}
$$

Simplifying all the constraints together, we obtain the following relationships:

$$
C_{11} = C_{21} = -C_{22} = -C_{12} = c_2 - c_1.
$$

4. Find all the independent solutions to the equations involving only unknowns in the coefficient matrix, ignoring the unknowns in the constant vectors in this step. There is only one independent choice in the coefficient matrix, so the affine partitions we seek can have at most a rank of one. We keep the partition as simple as possible by setting $C_{11} = 1$. We cannot assign 0 to C_{11} because that will create a rank-0 coefficient matrix, which maps all iterations to the same processor. It then follows that $C_{21} = 1$, $C_{22} = -1$, $C_{12} = -1$.

5. Find the constant terms. We know that the difference between the constant terms, $c_2 - c_1$, must be -1. We get to pick the actual values, however. To keep the partitions simple, we pick $c_2 = 0$; thus $c_1 = -1$.

Let p be the ID of the processor executing iteration (i, j). In terms of p, the affine partition is

$$
s_1 : \begin{bmatrix} p \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}
$$

$$
s_2 : \begin{bmatrix} p \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix}
$$

That is, the (i, j)th iteration of s_1 is assigned to the processor $p = i - j - 1$, and the (i, j)th iteration of s_2 is assigned to processor $p = i - j$. □

Algorithm 11.43: Finding a highest-ranked synchronization-free affine partition for a program.

INPUT: A program with affine array accesses.

OUTPUT: A partition.

METHOD: Do the following:

1. Find all data-dependent pairs of accesses in a program for each pair of data-dependent accesses, $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ in statement s_1 nested in d_1 loops and $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ in statement s_2 nested in d_2 loops. Let $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ and $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ represent the (currently unknown) partitions of statements s_1 and s_2, respectively. The space-partition constraints state that if

$$\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$$

   then

$$\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2$$

   for all $\mathbf{i}_1$ and $\mathbf{i}_2$, within their respective loop bounds. We shall generalize the domain of iterations to include all $\mathbf{i}_1$ in Z^{d_1} and $\mathbf{i}_2$ in Z^{d_2}; that is, the bounds are all assumed to be minus infinity to infinity. This assumption makes sense, since an affine partition cannot make use of the fact that an index variable can take on only a limited set of integer values.

2. For each pair of dependent accesses, we reduce the number of unknowns in the index vectors.

   (a) Note that $\mathbf{F}\mathbf{i} + \mathbf{f}$ is the same vector as

$$\begin{bmatrix} \mathbf{F} & \mathbf{f} \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ 1 \end{bmatrix}$$

   That is, by adding an extra component 1 at the bottom of column-vector $\mathbf{i}$, we can make the column-vector $\mathbf{f}$ be an additional, last column of the matrix $\mathbf{F}$. We may thus rewrite the equality of the access functions $\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$ as

$$\begin{bmatrix} \mathbf{F}_1 & -\mathbf{F}_2 & (\mathbf{f}_1 - \mathbf{f}_2) \end{bmatrix} \begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \\ 1 \end{bmatrix} = 0$$

   (b) The above equations will in general have more than one solution. However, we may still use Gaussian elimination to solve the equations for the components of $\mathbf{i}_1$ and $\mathbf{i}_2$ as best we can. That is, eliminate as many variables as possible until we are left with only variables that cannot be eliminated. The resulting solution for $\mathbf{i}_1$ and $\mathbf{i}_2$ will have the form

$$\begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \\ 1 \end{bmatrix} = \mathbf{U} \begin{bmatrix} \mathbf{t} \\ 1 \end{bmatrix}$$

where $\mathbf{U}$ is an upper-triangular matrix and $\mathbf{t}$ is a vector of free variables ranging over all integers.

(c) We may use the same trick as in Step (2a) to rewrite the equality of the partitions. Substituting the vector $(\mathbf{i}_1, \mathbf{i}_2, 1)$ with the result from Step (2b), we can write the constraints on the partitions as

$$\left[\begin{array}{ccc} \mathbf{C}_1 & -\mathbf{C}_2 & (\mathbf{c}_1 - \mathbf{c}_2) \end{array}\right] \mathbf{U} \left[\begin{array}{c} \mathbf{t} \\ 1 \end{array}\right] = 0$$

3. Drop the nonpartition variables. The equations above hold for all combinations of $\mathbf{t}$ if

$$\left[\begin{array}{ccc} \mathbf{C}_1 & -\mathbf{C}_2 & (\mathbf{c}_1 - \mathbf{c}_2) \end{array}\right] \mathbf{U} = \mathbf{0}.$$

Rewrite these equations in the form $\mathbf{A}\mathbf{x} = \mathbf{0}$, where $\mathbf{x}$ is a vector of all the unknown coefficients of the affine partitions.

4. Find the rank of the affine partition and solve for the coefficient matrices. Since the rank of an affine partition is independent of the value of the constant terms in the partition, we eliminate all the unknowns that come from the constant vectors like $\mathbf{c}_1$ or $\mathbf{c}_2$, thus replacing $\mathbf{A}\mathbf{x} = \mathbf{0}$ by simplified constraints $\mathbf{A}'\mathbf{x}' = \mathbf{0}$. Find the solutions to $\mathbf{A}'\mathbf{x}' = \mathbf{0}$, expressing them as $\mathbf{B}$, a set of basis vectors spanning the null space of $\mathbf{A}'$.

5. Find the constant terms. Derive one row of the desired affine partition from each basis vector in $\mathbf{B}$, and derive the constant terms using $\mathbf{A}\mathbf{x} = \mathbf{0}$.

$\square$

Note that Step 3 ignores the constraints imposed by the loop bounds on variables $\mathbf{t}$. The constraints are only stricter as a result, and the algorithm must therefore be safe. That is, we place constraints on the $\mathbf{C}$'s and $\mathbf{c}$'s assuming $\mathbf{t}$ is arbitrary. Conceivably, there would be other solutions for the $\mathbf{C}$'s and $\mathbf{c}$'s that are valid only because some values of $\mathbf{t}$ are impossible. Not searching for these other solutions may cause us to miss an optimization, but cannot cause the program to be changed to a program that does something different from what the original program does.

11.7.5 A Simple Code-Generation Algorithm

Algorithm 11.43 generates affine partitions that split computations into independent partitions. Partitions can be assigned arbitrarily to different processors, since they are independent of one another. A processor may be assigned more than one partition and can interleave the execution of its partitions, as long as operations within each partition, which normally have data dependences, are executed sequentially.

It is relatively easy to generate a correct program given an affine partition. We first introduce Algorithm 11.45, a simple approach to generating code for a single processor that executes each of the independent partitions sequentially. Such code optimizes temporal locality, since array accesses that have several uses are very close in time. Moreover, the code easily can be turned into an SPMD program that executes each partition on a different processor. The code generated is, unfortunately, inefficient; we shall next discuss optimizations to make the code execute efficiently.

The essential idea is as follows. We are given bounds for the index variables of a loop nest, and we have determined, in Algorithm 11.43, a partition for the accesses of a particular statement s. Suppose we wish to generate sequential code that performs the action of each processor sequentially. We create an outermost loop that iterates through the processor IDs. That is, each iteration of this loop performs the operations assigned to a particular processor ID. The original program is inserted as the loop body of this loop; in addition, a test is added to guard each operation in the code to ensure that each processor only executes the operations assigned to it. In this way, we guarantee that the processor executes all the instructions assigned to it, and does so in the original sequential order.

Example 11.44: Let us generate code that executes the independent partitions in Example 11.41 sequentially. The original sequential program is from Fig. 11.26 is repeated here as Fig. 11.28.

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */
        Y[i,j] = Y[i,j] + X[i,j-1];  /* (s2) */
    }
```

Figure 11.28: Repeat of Fig. 11.26

In Example 11.42, the affine partitioning algorithm found one degree of parallelism. Thus, the processor space can be represented by a single variable p. Recall also from that example that we selected an affine partition that, for all values of index variables i and j with $1 \le i \le 100$ and $1 \le j \le 100$, assigned

1. Instance (i, j) of statement s_1 to processor $p = i - j - 1$, and

2. Instance (i, j) of statement s_2 to processor $p = i - j$.

We can generate the code in three steps:

1. For each statement, find all the processor IDs participating in the computation. We combine the constraints $1 \le i \le 100$ and $1 \le j \le 100$ with one of the equations $p = i - j - 1$ or $p = i - j$, and project away i and j to get the new constraints

(a) $-100 \le p \le 98$ if we use the function $p = i - j - 1$ that we get for statement s_1, and

(b) $-99 \le p \le 99$ if we use $p = i - j$ from statement s_2.

2. Find all the processor IDs participating in any of the statements. When we take the union of these ranges, we get $-100 \le p \le 99$; these bounds are sufficient to cover all instances of both statements s_1 and s_2.

3. Generate the code to iterate through the computations in each partition sequentially. The code, shown in Fig. 11.29, has an outer loop that iterates through all the partition IDs participating in the computation (line (1)). Each partition goes through the motion of generating the indexes of all the iterations in the original sequential program in lines (2) and (3) so that it can pick out the iterations the processor p is supposed to execute. The tests of lines (4) and (6) make sure that statements s_1 and s_2 are executed only when the processor p would execute them.

The generated code, while correct, is extremely inefficient. First, even though each processor executes computation from at most 99 iterations, it generates loop indexes for 100×100 iterations, an order of magnitude more than necessary. Second, each addition in the innermost loop is guarded by a test, creating another constant factor of overhead. These two kinds of inefficiencies are dealt with in Sections 11.7.6 and 11.7.7, respectively.   □

```
1)    for (p = -100; p <= 99; p++)
2)        for (i = 1; i <= 100; i++)
3)            for (j = 1; j <= 100; j++) {
4)                if (p == i-j-1)
5)                    X[i,j] = X[i,j] + Y[i-1,j];   /* (s1) */
6)                if (p == i-j)
7)                    Y[i,j] = X[i,j-1] + Y[i,j];   /* (s2) */
8)            }
```

Figure 11.29: A simple rewriting of Fig. 11.28 that iterates over processor space

Although the code of Fig. 11.29 appears designed to execute on a uniprocessor, we could take the inner loops, lines (2) through (8), and execute them on 200 different processors, each of which had a different value for p, from -100 to 99. Or, we could partition the responsibility for the inner loops among any number of processors less than 200, as long as we arranged that each processor knew what values of p it was responsible for and executed lines (2) through (8) for just those values of p.

Algorithm 11.45 : Generating code that executes partitions of a program sequentially.

INPUT: A program P with affine array accesses. Each statement s in the program has associated bounds of the form $\mathbf{B}_s\mathbf{i} + \mathbf{b}_s \geq \mathbf{0}$, where $\mathbf{i}$ is the vector of loop indexes for the loop nest in which statement s appears. Also associated with statement s is a partition $\mathbf{C}_s\mathbf{i} + \mathbf{c}_s = \mathbf{p}$ where $\mathbf{p}$ is an m-dimensional vector of variables representing a processor ID; m is the maximum, over all statements in program P, of the rank of the partition for that statement.

OUTPUT: A program equivalent to P but iterating over the processor space rather than over loop indexes.

METHOD: Do the following:

1. For each statement, use Fourier-Motzkin elimination to project out all the loop index variables from the bounds.

2. Use Algorithm 11.13 to determine bounds on the partition ID's.

3. Generate loops, one for each of the m dimensions of processor space. Let $\mathbf{p} = [p_1, p_2, \ldots, p_m]$ be the vector of variables for these loops; that is, there is one variable for each dimension of the processor space. Each loop variable p_i ranges over the union of the partition spaces for all statements in the program P.

Note that the union of the partition spaces is not necessarily convex. To keep the algorithm simple, instead of enumerating only those partitions that have a nonempty computation to perform, set the lower bound of each p_i to the minimum of all the lower bounds imposed by all statements and the upper bound of each p_i to the maximum of all the upper bounds imposed by all statements. Some values of $\mathbf{p}$ may thereby have no operations.

The code to be executed by each partition is the original sequential program. However, every statement is guarded by a predicate so that only those operations belonging to the partition are executed. $\quad\Box$

An example of Algorithm 11.45 will follow shortly. Bear in mind, however, that we are still far from the optimal code for typical examples.

11.7.6 Eliminating Empty Iterations

We now discuss the first of the two transformations necessary to generate efficient SPMD code. The code executed by each processor cycles through all the iterations in the original program and picks out the operations that it is supposed to execute. If the code has k degrees of parallelism, the effect is that each processor performs k orders of magnitude more work. The purpose of the first transformation is to tighten the bounds of the loops to eliminate all the empty iterations.

We begin by considering the statements in the program one at a time. A statement's iteration space to be executed by each partition is the original iteration space plus the constraint imposed by the affine partition. We can generate

tight bounds for each statement by applying Algorithm 11.13 to the new iteration space; the new index vector is like the original sequential index vector, with processor ID's added as outermost indexes. Recall that the algorithm will generate tight bounds for each index in terms of surrounding loop indexes.

After finding the iteration spaces of the different statements, we combine them, loop by loop, making the bounds the union of those for each statement. Some loops end up having a single iteration, as illustrated by Example 11.46 below, and we can simply eliminate the loop and simply set the loop index to the value for that iteration.

Example 11.46: For the loop of Fig. 11.30(a), Algorithm 11.43 will create the affine partition

$$s_1 : p = i$$
$$s_2 : p = j$$

Algorithm 11.45 will create the code of Fig. 11.30(b). Applying Algorithm 11.13 to statement s_1 produces the bound: $p \leq i \leq p$, or simply $i = p$. Similarly, the algorithm determines $j = p$ for statement s_2. Thus, we get the code of Fig. 11.30(c). Copy propagation of variables i and j will eliminate the unnecessary test and produce the code of Fig. 11.30(d). □

We now return to Example 11.44 and illustrate the step to merge multiple iteration spaces from different statements together.

Example 11.47: Let us now tighten the loop bounds of the code in Example 11.44. The iteration space executed by partition p for statement s_1 is defined by the following equalities and inequalities:

$$
\begin{array}{ccccc}
-100 & \leq & p & \leq & 99 \\
1 & \leq & i & \leq & 100 \\
1 & \leq & j & \leq & 100
\end{array}
$$

$$i - p - 1 = j$$

Applying Algorithm 11.13 to the above creates the constraints shown in Fig. 11.31(a). Algorithm 11.13 generates the constraint $p + 2 \leq i \leq 100 + p + 1$ from $i - p - 1 = j$ and $1 \leq j \leq 100$, and tightens the upper bound of p to 98. Likewise, the bounds for each of the variables for statement s_2 are shown in Fig. 11.31(b).

The iteration spaces for s_1 and s_2 in Fig. 11.31 are similar, but as expected from Fig. 11.27, certain limits differ by 1 between the two. The code in Fig. 11.32 executes over this union of iteration spaces. For example, for i use $\max(1, p + 1)$ as the lower bound and $\min(100, 101 + p)$ as the upper bound. Note that the innermost loop has 2 iterations except that it has only one the first and last time it is executed. The overhead in generating loop indexes is thus reduced by an order of magnitude. Since the iteration space executed is larger than either that of s_1 and s_2, conditionals are still necessary to select when these statements are executed. □

```
for (i=1; i<=N; i++)
    Y[i] = Z[i]; /* (s1) */
for (j=1; j<=N; j++)
    X[j] = Y[j]; /* (s2) */
```

(a) Initial code.

```
for (p=1; p<=N; p++) {
    for (i=1; i<=N; i++)
        if (p == i)
            Y[i] = Z[i]; /* (s1) */
    for (j=1; j<=N; j++)
        if (p == j)
            X[j] = Y[j]; /* (s2) */
}
```

(b) Result of applying Algorithm 11.45.

```
for (p=1; p<=N; p++) {
    i = p;
    if (p == i)
        Y[i] = Z[i]; /* (s1) */
    j = p;
    if (p == j)
        X[j] = Y[j]; /* (s2) */
}
```

(c) After applying Algorithm 11.13.

```
for (p=1; p<=N; p++) {
    Y[p] = Z[p]; /* (s1) */
    X[p] = Y[p]; /* (s2) */
}
```

(d) Final code.

Figure 11.30: Code for Example 11.46

$$j: \quad i - p - 1 \quad \leq \quad j \quad \leq \quad i - p - 1$$
$$1 \quad \leq \quad j \quad \leq \quad 100$$

$$i: \quad p + 2 \quad \leq \quad i \quad \leq \quad 100 + p + 1$$
$$1 \quad \leq \quad i \quad \leq \quad 100$$

$$p: \quad -100 \quad \leq \quad p \quad \leq \quad 98$$

(a) Bounds for statement s_1.

$$j: \quad i - p \quad \leq \quad j \quad \leq \quad i - p$$
$$1 \quad \leq \quad j \quad \leq \quad 100$$

$$i: \quad p + 1 \quad \leq \quad i \quad \leq \quad 100 + p$$
$$1 \quad \leq \quad i \quad \leq \quad 100$$

$$p: \quad -99 \quad \leq \quad p \quad \leq \quad 99$$

(b) Bounds for statement s_2.

Figure 11.31: Tighter bounds on p, i, and j for Fig. 11.29

11.7.7 Eliminating Tests from Innermost Loops

The second transformation is to remove conditional tests from the inner loops. As seen from the examples above, conditional tests remain if the iteration spaces of statements in the loop intersect but not completely. To avoid the need for conditional tests, we split the iteration space into subspaces, each of which executes the same set of statements. This optimization requires code to be duplicated and should only be used to remove conditionals in the inner loops.

To split an iteration space to reduce tests in inner loops, we apply the following steps repeatedly until we remove all the tests in the inner loops:

1. Select a loop that consists of statements with different bounds.

2. Split the loop using a condition such that some statement is excluded from at least one of its components. We choose the condition from among the boundaries of the overlapping different polyhedra. If some statement has all its iterations in only one of the half planes of the condition, then such a condition is useful.

3. Generate code for each of these iteration spaces separately.

Example 11.48: Let us remove the conditionals from the code of Fig. 11.32. Statements s_1 and s_2 are mapped to the same set of partition ID's except for

```
for (p = -100; p <= 99; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,i-p); j++) {
            if (p == i-j-1)
                X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */
            if (p == i-j)
                Y[i,j] = X[i,j-1] + Y[i,j];  /* (s2) */
        }
```

Figure 11.32: Code of Fig. 11.29 improved by tighter loop bounds

the boundary partitions at either end. Thus, we separate the partition space into three subspaces:

1. $p = -100$,

2. $-99 \leq p \leq 98$, and

3. $p = 99$.

The code for each subspace can then be specialized for the value(s) of p contained. Figure 11.33 shows the resulting code for each of the three iteration spaces.

Notice that the first and third spaces do not need loops on i or j, because for the particular value of p that defines each space, these loops are degenerate; they have only one iteration. For example, in space (1), substituting $p = -100$ in the loop bounds restricts i to 1, and subsequently j to 100. The assignments to p in spaces (1) and (3) are evidently dead code and can be eliminated.

Next we split the loop with index i in space (2). Again, the first and last iterations of loop index i are different. Thus, we split the loop into three subspaces:

a) $\max(1, p + 1) \leq i < p + 2$, where only s_2 is executed,

b) $\max(1, p+2) \leq i \leq \min(100, 100+p)$, where both s_1 and s_2 are executed, and

c) $101 + p < i \leq \min(101 + p, 100)$, where only s_1 is executed.

The loop nest for space (2) in Fig. 11.33 can thus be written as in Fig. 11.34(a).

Figure 11.34(b) shows the optimized program. We have substituted Fig. 11.34(a) for the loop nest in Fig. 11.33. We also propagated out assignments to p, i, and j into the array accesses. When optimizing at the intermediate-code level, some of these assignments will be identified as common subexpressions and re-extracted from the array-access code.   □

```
/* space (1) */
p = -100;
i = 1;
j = 100;
X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */

/* space (2) */
for (p = -99; p <= 98; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,i-p); j++) {
            if (p == i-j-1)
                X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */
            if (p == i-j)
                Y[i,j] = X[i,j-1] + Y[i,j];  /* (s2) */
        }

/* space (3) */
p = 99;
i = 100;
j = 1;
Y[i,j] = X[i,j-1] + Y[i,j];  /* (s2) */
```

Figure 11.33: Splitting the iteration space on the value of p

11.7.8 Source-Code Transforms

We have seen how we can derive from simple affine partitions for each statement programs that are significantly different from the original source. It is not apparent from the examples seen so far how affine partitions correlate with changes at the source level. This section shows that we can reason about source code changes relatively easily by breaking down affine partitions into a series of primitive transforms.

Seven Primitive Affine Transforms

Every affine partition can be expressed as a series of primitive affine transforms, each of which corresponds to a simple change at the source level. There are seven kinds of primitive transforms: the first four primitives are illustrated in Fig. 11.35, the last three, also known as *unimodular transforms*, are illustrated in Fig. 11.36.

   The figure shows one example for each primitive: a source, an affine partition, and the resulting code. We also draw the data dependences for the code before and after the transforms. From the data dependence diagrams, we see that each primitive corresponds to a simple geometric transform and induces a relatively simple code transform. The seven primitives are:

```
    /* space (2) */
    for (p = -99; p <= 98; p++) {
        /* space (2a) */
        if (p >= 0) {
            i = p+1;
            j = 1;
            Y[i,j] = X[i,j-1] + Y[i,j];  /* (s2) */
        }
        /* space (2b) */
        for (i = max(1,p+2); i <= min(100,100+p); i++) {
            j = i-p-1;
            X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */
            j = i-p;
            Y[i,j] = X[i,j-1] + Y[i,j];  /* (s2) */
        }
        /* space (2c) */
        if (p <= -1) {
            i = 101+p;
            j = 100;
            X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */
        }
    }
```

(a) Splitting space (2) on the value of i.

```
/* space (1); p = -100  */
X[1,100] = X[1,100] + Y[0,100];                         /* (s1) */


/* space (2) */
for (p = -99; p <= 98; p++) {
    if (p >= 0)
        Y[p+1,1] = X[p+1,0] + Y[p+1,1];                 /* (s2) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        X[i,i-p-1] = X[i,i-p-1] + Y[i-1,i-p-1];         /* (s1) */
        Y[i,i-p] = X[i,i-p-1] + Y[i,i-p];               /* (s2) */
    }
    if (p <= -1)
        X[101+p,100] = X[101+p,100] + Y[101+p-1,100]; /* (s1) */
}
/*   space (3); p = 99   */
Y[100,1] = X[100,0] + Y[100,1];                         /* (s2) */
```

(b) Optimized code equivalent to Fig. 11.28.

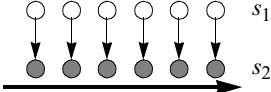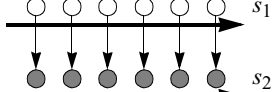Figure 11.34: Code for Example 11.48
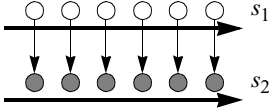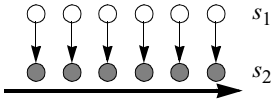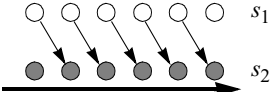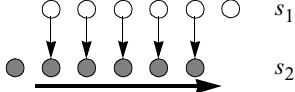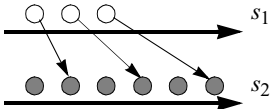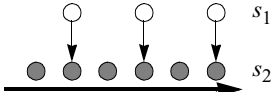
Source Code	Partition	Transformed Code
```for (i=1; i<=N; i++) Y[i] = Z[i]; /*s1*/ for (j=1; j<=N; j++) X[j] = Y[j]; /*s2*/```	Fusion $s_1 : p = i$ $s_2 : p = j$	```for (p=1; p<=N; p++){ Y[p] = Z[p]; X[p] = Y[p]; }```
```for (p=1; p<=N; p++){ Y[p] = Z[p]; X[p] = Y[p]; }```	Fission $s_1 : i = p$ $s_2 : j = p$	```for (i=1; i<=N; i++) Y[i] = Z[i]; /*s1*/ for (j=1; j<=N; j++) X[j] = Y[j]; /*s2*/```
```for (i=1; i<=N; i++) { Y[i] = Z[i]; /*s1*/ X[i] = Y[i-1]; /*s2*/ }```	Re-indexing $s_1 : p = i$ $s_2 : p = i - 1$	```if (N>=1) X[1]=Y[0]; for (p=1; p<=N-1; p++){ Y[p]=Z[p]; X[p+1]=Y[p]; } if (N>=1) Y[N]=Z[N];```
```for (i=1; i<=N; i++) Y[2*i] = Z[2*i]; /*s1*/ for (j=1; j<=2N; j++) X[j]=Y[j]; /*s2*/```	Scaling $s_1 : p = 2 * i$ $(s_2 : p = j)$	```for (p=1; p<=2*N; p++){ if (p mod 2 == 0) Y[p] = Z[p]; X[p] = Y[p]; }```

Figure 11.35: Primitive affine transforms (I)
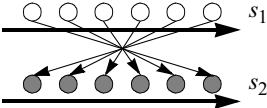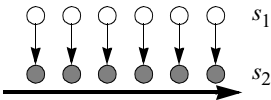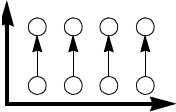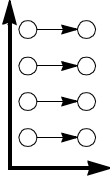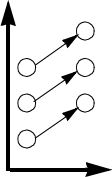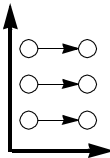
Source Code	Partition	Transformed Code
```for (i=0; i<=N; i++) Y[N-i] = Z[i]; /*s1*/ for (j=0; j<=N; j++) X[j] = Y[j]; /*s2*/```	Reversal $s_1 : p = N - i$ $(s_2 : p = j)$	```for (p=0; p<=N; p++){ Y[p] = Z[N-p]; X[p] = Y[p]; }```
```for (i=1; i<=N; i++) for (j=0; j<=M; j++) Z[i,j] = Z[i-1,j];```	Permutation $$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$	```for (p=0; p<=M; p++) for (q=1; q<=N; i++) Z[q,p] = Z[q-1,p];```
```for (i=1; i<=N+M-1; i++) for (j=max(1,i+N); j<=min(i,M); j++) Z[i,j] = Z[i-1,j-1];```	Skewing $$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$	```for (p=1; p<=N; p++) for (q=1; q<=M; q++) Z[p,q-p] = Z[p-1,q-p-1];```

Figure 11.36: Primitive affine transforms (II)

Unimodular Transforms

A unimodular transform is represented by just a unimodular coefficient matrix and no constant vector. A *unimodular matrix* is a square matrix whose determinant is ± 1. The significance of a unimodular transform is that it maps an n-dimensional iteration space to another n-dimensional polyhedron, where there is a one-to-one correspondence between iterations of the two spaces.

---

1. *Fusion.* The fusion transform is characterized by mapping multiple loop indexes in the original program to the same loop index. The new loop fuses statements from different loops.

2. *Fission.* Fission is the inverse of fusion. It maps the same loop index for different statements to different loop indexes in the transformed code. This splits the original loop into multiple loops.

3. *Re-indexing.* Re-indexing shifts the dynamic executions of a statement by a constant number of iterations. The affine transform has a constant term.

4. *Scaling.* Consecutive iterations in the source program are spaced apart by a constant factor. The affine transform has a positive nonunit coefficient.

5. *Reversal.* Execute iterations in a loop in reverse order. Reversal is characterized by having -1 as a coefficient.

6. *Permutation.* Permute the inner and outer loops. The affine transform consists of permuted rows of the identity matrix.

7. *Skewing.* Iterate through the iteration space in the loops at an angle. The affine transform is a unimodular matrix with 1's on the diagonal.

A Geometric Interpretation of Parallelization

The affine transforms shown in all but the fission example are derived by applying the synchronization-free affine partition algorithm to the respective source codes. (We shall discuss how fission can parallelize code with synchronization in the next section.) In each of the examples, the generated code has an (outermost) parallelizable loop whose iterations can be assigned to different processors and no synchronization is necessary.

These examples illustrate that there is a simple geometric interpretation of how parallelization works. Dependence edges always point from an earlier instance to a later instance. So, dependences between separate statements not nested in any common loop follows the lexical order; dependences between

statements nested in the same loop follow the lexicographic order. Geometrically, dependences of a two-dimensional loop nest always point within the range $[0°, 180°)$, meaning that the angle of the dependence must be below $180°$, but no less than $0°$.

The affine transforms change the ordering of iterations such that all the dependences are found only between operations nested within the same iteration of the outermost loop. In other words, there are no dependence edges at the boundaries of iterations in the outermost loop. We can parallelize simple source codes by drawing their dependences and finding such transforms geometrically.

11.7.9 Exercises for Section 11.7

Exercise 11.7.1 : For the following loop

```
for (i = 2; i < 100; i++)
    A[i] = A[i-2];
```

a) What is the largest number of processors that can be used effectively to execute this loop?

b) Rewrite the code with processor p as a parameter.

c) Set up and find one solution to the space-partition constraints for this loop.

d) What is the affine partition of highest rank for this loop?

Exercise 11.7.2 : Repeat Exercise 11.7.1 for the loop nests in Fig. 11.37.

Exercise 11.7.3 : Rewrite the following code

```
for (i = 0; i < 100; i++)
    A[i] = 2*A[i];
for (j = 0; j < 100; j++)
    A[j] = A[j] + 1;
```

so it consists of a single loop. Rewrite the loop in terms of a processor number p so the code can be partitioned among 100 processors, with iteration p executed by processor p.

Exercise 11.7.4 : In the following code

```
    for (i = 1; i < 100; i++)
        for (j = 1; j < 100; j++)
/* (s) */    A[i,j] =
                (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1])/4;
```

```
for (i = 0; i <= 97; i++)
    A[i] = A[i+2];
```

(a)

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++)
        for (k = 1; k <= 100; k++) {
            A[i,j,k] = A[i,j,k] + B[i-1,j,k];
            B[i,j,k] = B[i,j,k] + C[i,j-1,k];
            C[i,j,k] = C[i,j,k] + A[i,j,k-1];
        }
```

!(b)

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++)
        for (k = 1; k <= 100; k++) {
            A[i,j,k] = A[i,j,k] + B[i-1,j,k];
            B[i,j,k] = B[i,j,k] + A[i,j-1,k];
            C[i,j,k] = C[i,j,k] + A[i,j,k-1] + B[i,j,k];
        }
```

!(c)

Figure 11.37: Code for Exercise 11.7.2

the only constraints are that the statement s that forms the body of the loop nest must execute iterations $s(i-1,j)$ and $s(i,j-1)$ before executing iteration $s(i,j)$. Verify that these are the only necessary constraints. Then rewrite the code so that the outer loop has index variable p, and on the pth iteration of the outer loop, all instances of $s(i,j)$ such that $i+j=p$ are executed.

Exercise 11.7.5 : Repeat Exercise 11.7.4, but arrange that on the pth iteration of the outer loop, instances of s such that $i-j=p$ are executed.

! **Exercise 11.7.6 :** Combine the following loops

```
for (i = 0; i < 100; i++)
    A[i] = B[i];
for (j = 98; j >= 0; j = j-2)
    B[i] = i;
```

into a single loop, preserving all dependencies.

Exercise 11.7.7: Show that the matrix

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

is unimodular. Describe the transformation it performs on a two-dimensional loop nest.

Exercise 11.7.8: Repeat Exercise 11.7.7 on the matrix

$$\begin{bmatrix} 1 & 3 \\ 2 & 5 \end{bmatrix}$$

11.8 Synchronization Between Parallel Loops

Most programs have no parallelism if we do not allow processors to perform any synchronizations. But adding even a small constant number of synchronization operations to a program can expose more parallelism. We shall first discuss parallelism made possible by a constant number of synchronizations in this section and the general case, where we embed synchronization operations in loops, in the next.

11.8.1 A Constant Number of Synchronizations

Programs with no synchronization-free parallelism may contain a sequence of loops, some of which are parallelizable if they are considered independently. We can parallelize such loops by introducing synchronization barriers before and after their execution. Example 11.49 illustrates the point.

```
for (i = 1; i < n; i++)
    for (j = 0; j < n; j++)
        X[i,j] = f(X[i,j] + X[i-1,j]);
for (i = 0; i < n; i++)
    for (j = 1; j < n; j++)
        X[i,j] = g(X[i,j] + X[i,j-1]);
```

Figure 11.38: Two sequential loop nests

Example 11.49: In Fig. 11.38 is a program representative of an ADI (Alternating Direction Implicit) integration algorithm. There is no synchronization-free parallelism. Dependences in the first loop nest require that each processor works on a column of array X; however, dependences in the second loop nest require that each processor works on a row of array X. For there to be no communication, the entire array has to reside on the same processor, hence there

is no parallelism. We observe, however, that both loops are independently parallelizable.

One way to parallelize the code is to have different processors work on different columns of the array in the first loop, synchronize and wait for all processors to finish, and then operate on the individual rows. In this way, all the computation in the algorithm can be parallelized with the introduction of just one synchronization operation. However, we note that while only one synchronization is performed, this parallelization requires almost all the data in matrix X to be transferred between processors. It is possible to reduce the amount of communication by introducing more synchronizations, which we shall discuss in Section 11.9.9. □

It may appear that this approach is applicable only to programs consisting of a sequence of loop nests. However, we can create additional opportunities for the optimization through code transforms. We can apply loop fission to decompose loops in the original program into several smaller loops, which can then be parallelized individually by separating them with barriers. We illustrate this technique with Example 11.50.

Example 11.50: Consider the following loop:

```
for (i=1; i<=n; i++) {
    X[i] = Y[i] + Z[i];    /* (s1) */
    W[A[i]] = X[i];        /* (s2) */
}
```

Without knowledge of the values in array A, we must assume that the access in statement s_2 may write to any of the elements of W. Thus, the instances of s_2 must be executed sequentially in the order they are executed in the original program.

There is no synchronization-free parallelism, and Algorithm 11.43 will simply assign all the computation to the same processor. However, at the least, instances of statement s_1 can be executed in parallel. We can parallelize part of this code by having different processors perform difference instances of statement s_1. Then, in a separate sequential loop, one processor, say numbered 0, executes s_2, as in the SPMD code shown in Fig. 11.39. □

11.8.2 Program-Dependence Graphs

To find all the parallelism made possible by a constant number of synchronizations, we can apply fission to the original program greedily. Break up loops into as many separate loops as possible, and then parallelize each loop independently.

To expose all the opportunities for loop fission, we use the abstraction of a *program-dependence graph* (PDG). A program dependence graph of a program

```
X[p] = Y[p] + Z[p];     /* (s1) */
/* synchronization barrier */
if (p == 0)
    for (i=1; i<=n; i++)
        W[A[i]] = X[i];  /* (s2) */
```

Figure 11.39: SPMD code for the loop in Example 11.50, with p being a variable holding the processor ID

is a graph whose nodes are the assignment statements of the program and whose edges capture the data dependences, and the directions of the data dependence, between statements. An edge from statement s_1 to statement s_2 exists whenever some dynamic instance of s_1 shares a data dependence with a *later* dynamic instance of s_2.

To construct the PDG for a program, we first find the data dependences between every pair of (not necessarily distinct) static accesses in every pair of (not necessarily distinct) statements. Suppose we determine that there is a dependence between access $\mathcal{F}_1$ in statement s_1 and access $\mathcal{F}_2$ in statement s_2. Recall that an instance of a statement is specified by an index vector $\mathbf{i} = [i_1, i_2, \ldots, i_m]$ where i_k is the loop index of the kth outermost loop in which the statement is embedded.

1. If there exists a data-dependent pair of instances, $\mathbf{i_1}$ of s_1 and $\mathbf{i_2}$ of s_2, and $\mathbf{i_1}$ is executed before $\mathbf{i_2}$ in the original program, written $\mathbf{i_1} \prec_{s_1 s_2} \mathbf{i_2}$, then there is an edge from s_1 to s_2.

2. Similarly, if there exists a data-dependent pair of instances, $\mathbf{i_1}$ of s_1 and $\mathbf{i_2}$ of s_2, and $\mathbf{i_2} \prec_{s_1 s_2} \mathbf{i_1}$, then there is an edge from s_2 to s_1.

Note that it is possible for a data dependence between two statements s_1 and s_2 to generate both an edge from s_1 to s_2 and an edge from s_2 back to s_1.

In the special case where statements s_1 and s_2 are not distinct, $\mathbf{i_1} \prec_{s_1 s_2} \mathbf{i_2}$ if and only if $\mathbf{i_1} \prec \mathbf{i_2}$ ($\mathbf{i_1}$ is lexicographically less than $\mathbf{i_2}$). In the general case, s_1 and s_2 may be different statements, possibly belonging to different loop nests.

Example 11.51: For the program of Example 11.50, there are no dependences among the instances of statement s_1. However, the ith instance of statement s_2 must follow the ith instance of statement s_1. Worse, since the reference $W[A[i]]$ may write any element of array W, the ith instance of s_2 depends on all previous instances of s_2. That is, statement s_2 depends on itself. The PDG for the program of Example 11.50 is shown in Fig. 11.40. Note that there is one cycle in the graph, containing s_2 only. □

The program-dependence graph makes it easy to determine if we can split statements in a loop. Statements connected in a cycle in a PDG cannot be
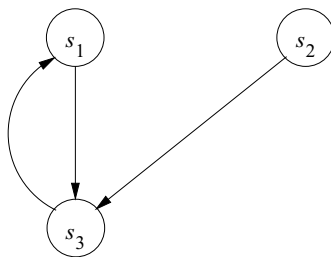
Figure 11.40: Program-dependence graph for the program of Example 11.50

split. If $s_1 \rightarrow s_2$ is a dependence between two statements in a cycle, then some instance of s_1 must execute before some instance of s_2, and vice versa. Note that this mutual dependence occurs only if s_1 and s_2 are embedded in some common loop. Because of the mutual dependence, we cannot execute all instances of one statement before the other, and therefore loop fission is not allowed. On the other hand, if the dependence $s_1 \rightarrow s_2$ is unidirectional, we can split up the loop and execute all the instances of s_1 first, then those of s_2.

```
for (i = 0; i < n; i++) {
    Z[i] = Z[i] / W[i];            /* (s1) */
    for (j = i; j < n; j++) {
        X[i,j] = Y[i,j]*Y[i,j];    /* (s2) */
        Z[j] = Z[j] + X[i,j];      /* (s3) */
    }
}
```

(a) A program.

(b) Its dependence graph.

Figure 11.41: Program and dependence graph for Example 11.52.

Example 11.52: Figure 11.41(b) shows the program-dependence graph for the program of Fig. 11.41(a). Statements s_1 and s_3 belong to a cycle in the graph and therefore cannot be placed in separate loops. We can, however, split statement s_2 out and execute all its instances before executing the rest of the computation, as in Fig. 11.42. The first loop is parallelizable, but the second is not. We can parallelize the first loop by placing barriers before and after its parallel execution. □

```
for (i = 0; i < n; i++)
   for (j = i; j < n; j++)
      X[i,j] = Y[i,j]*Y[i,j];    /* (s2) */
for (i = 0; i < n; i++) {
   Z[i] = Z[i] / W[i];           /* (s1) */
   for (j = i; j < n; j++)
      Z[j] = Z[j] + X[i,j];      /* (s3) */
}
```

Figure 11.42: Grouping strongly connected components of a loop nest

11.8.3 Hierarchical Time

While the relation $\prec_{s_1 s_2}$ can be very hard to compute in general, there is a
family of programs to which the optimizations of this section are commonly
applied, and for which there is a straightforward way to compute dependencies.
Assume that the program is block structured, consisting of loops and simple
arithmetic operations and no other control constructs. A statement in the
program is either an assignment statement, a sequence of statements, or a loop
construct whose body is a statement. The control structure thus represents a
hierarchy. At the top of the hierarchy is the node representing the statement of
the whole program. An assignment statement is a leaf node. If a statement is
a sequence, then its children are the statements within the sequence, laid out
from left to right according to their lexical order. If a statement is a loop, then
its children are the components of the loop body, which is typically a sequence
of one or more statements.

```
s0;
L1: for (i = 0; ...) {
       s1;
       L2: for (j = 0; ...) {
               s2;
               s3;
           }
       L3: for (k = 0; ... )
               s4;
       s5;
    }
```

Figure 11.43: A hierarchically structured program

Example 11.53: The hierarchical structure of the program in Fig. 11.43 is
shown in Fig. 11.44. The hierarchical nature of the execution sequence is high-

lighted in Fig. 11.45. The single instance of s_0 precedes all other operations, because it is the first statement executed. Next, we execute all instructions from the first iteration of the outer loop before those in the second iteration and so forth. For all dynamic instances whose loop index i has value 0, the statements s_1, L_2, L_3, and s_5 are executed in lexical order. We can repeat the same argument to generate the rest of the execution order. □
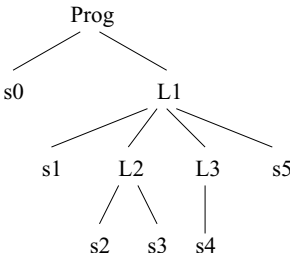
Figure 11.44: Hierarchical structure of the program in Example 11.53.

$$
\begin{array}{rlll}
1: & s_0 & & \\
2: & L_1 \quad i = 0 & s_1 & \\
3: & & L_2 \quad j = 0 & s_2 \\
4: & & & s_3 \\
5: & & j = 1 & s_2 \\
6: & & & s_3 \\
7: & & \ldots & \\
8: & & L_3 \quad k = 0 & s_4 \\
9: & & k = 1 & s_4 \\
10: & & \ldots & \\
11: & & s_5 & \\
12: & i = 1 & s_1 & \\
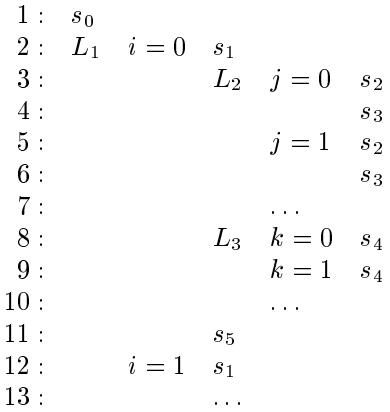13: & & \ldots & \\
\end{array}
$$

Figure 11.45: Execution order of the program in Example 11.53.

We can resolve the ordering of two instances from two different statements in a hierarchical manner. If the statements share common loops, we compare the values of their common loop indexes, starting with the outermost loop. As soon as we find a difference between their index values, the difference determines the ordering. Only if the index values for the outer loops are the same do we need to compare the indexes of the next inner loop. This process is analogous to how we would compare time expressed in terms of hours, minutes and seconds. To compare two times, we first compare the hours, and only if they refer to

the same hour would we compare the minutes and so forth. If the index values are the same for all common loops, then we resolve the order based on their relative lexical placement. Thus, the execution order for the simple nested-loop programs we have been discussing is often referred to as "hierarchical time."

Let s_1 be a statement nested in a d_1-deep loop, and s_2 in a d_2-deep loop, sharing d common (outer) loops; note $d \leq d_1$ and $d \leq d_2$ certainly. Suppose $\mathbf{i} = [i_1, i_2, \ldots, i_{d_1}]$ is an instance of s_1 and $\mathbf{j} = [j_1, j_2, \ldots, j_{d_2}]$ is an instance of s_2.

$\mathbf{i} \prec_{s_1 s_2} \mathbf{j}$ if and only if either

1. $[i_1, i_2, \ldots, i_d] \prec [j_1, j_2, \ldots, j_d]$, or

2. $[i_1, i_2, \ldots, i_d] = [j_1, j_2, \ldots, j_d]$, and s_1 appears lexically before s_2.

The predicate $[i_1, i_2, \ldots, i_d] \prec [j_1, j_2, \ldots, j_d]$ can be written as a disjunction of linear inequalities:

$$(i_1 < j_1) \vee (i_1 = j_1 \wedge i_2 < j_2) \vee \ldots \vee (i_1 = j_1 \wedge \ldots \wedge i_{d-1} = j_{d-1} \wedge i_d < j_d)$$

A PDG edge from s_1 to s_2 exists as long as the data-dependence condition and one of the disjunctive clauses can be made true simultaneously. Thus, we may need to solve up to d or $d + 1$ linear integer programs, depending on whether s_1 appears lexically before s_2, to determine the existence of one edge.

11.8.4 The Parallelization Algorithm

We now present a simple algorithm that first splits up the computation into as many different loops as possible, then parallelizes them independently.

Algorithm 11.54: Maximize the degree of parallelism allowed by $O(1)$ synchronizations.

INPUT: A program with array accesses.

OUTPUT: SPMD code with a constant number of synchronization barriers.

METHOD:

1. Construct the program-dependence graph and partition the statements into strongly connected components (SCC's). Recall from Section 10.5.8 that a strongly connected component is a maximal subgraph of the original whose every node in the subgraph can reach every other node.

2. Transform the code to execute SCC's in a topological order by applying fission if necessary.

3. Apply Algorithm 11.43 to each SCC to find all of its synchronization-free parallelism. Barriers are inserted before and after each parallelized SCC.

$\square$

While Algorithm 11.54 finds all degrees of parallelism with $O(1)$ synchronizations, it has a number of weaknesses. First, it may introduce unnecessary synchronizations. As a matter of fact, if we apply this algorithm to a program that can be parallelized without synchronization, the algorithm will parallelize each statement independently and introduce a synchronization barrier between the parallel loops executing each statement. Second, while there may only be a constant number of synchronizations, the parallelization scheme may transfer a lot of data among processors with each synchronization. In some cases, the cost of communication makes the parallelism too expensive, and we may even be better off executing the program sequentially on a uniprocessor. In the following sections, we shall next take up ways to increase data locality, and thus reduce the amount of communication.

11.8.5 Exercises for Section 11.8

Exercise 11.8.1: Apply Algorithm 11.54 to the code of Fig. 11.46.

```
for (i=0; i<100; i++)
    A[i] = A[i] + X[i];  /* (s1) */
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        B[i,j] = Y[i,j] + A[i] + A[j];  /* (s2) */
```

Figure 11.46: Code for Exercise 11.8.1

Exercise 11.8.2: Apply Algorithm 11.54 to the code of Fig. 11.47.

```
for (i=0; i<100; i++)
    A[i] = A[i] + X[i];  /* (s1) */
for (i=0; i<100; i++) {
    B[i] = B[i] + A[i];  /* (s2) */
    for (j=0; j<100; j++)
        C[j] = Y[j] + B[j];  /* (s3) */
}
```

Figure 11.47: Code for Exercise 11.8.2

Exercise 11.8.3: Apply Algorithm 11.54 to the code of Fig. 11.48.

```
for (i=0; i<100; i++)
    A[i] = A[i] + X[i];  /* (s1) */
for (i=0; i<100; i++) {
    for (j=0; j<100; j++)
        B[j] = A[i] + Y[j];  /* (s2) */
    C[i] = B[i] + Z[i];  /* (s3) */
    for (j=0; j<100; j++)
        D[i,j] = A[i] + B[j];  /* (s4) */
}
```

Figure 11.48: Code for Exercise 11.8.3

11.9 Pipelining

In pipelining, a task is decomposed into a number of stages to be performed on different processors. For example, a task computed using a loop of n iterations can be structured as a pipeline of n stages. Each stage is assigned to a different processor; when one processor is finished with its stage, the results are passed as input to the next processor in the pipeline.

In the following, we start by explaining the concept of pipelining in more detail. We then show a real-life numerical algorithm, known as successive over-relaxation, to illustrate the conditions under which pipelining can be applied, in Section 11.9.2. We then formally define the constraints that need to be solved in Section 11.9.6, and describe an algorithm for solving them in Section 11.9.7. Programs that have multiple independent solutions to the time-partition constraints are known as having outermost *fully permutable loops*; such loops can be pipelined easily, as discussed in Section 11.9.8.

11.9.1 What is Pipelining?

Our initial attempts to parallelize loops partitioned the iterations of a loop nest so that two iterations that shared data were assigned to the same processor. Pipelining allows processors to share data, but generally does so only in a "local," way, with data passed from one processor to another that is adjacent in the processor space. Here is a simple example.

Example 11.55: Consider the loop:

```
for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
        X[i] = X[i] + Y[i,j];
```

This code sums up the ith row of Y and adds it to the ith element of X. The inner loop, corresponding to the summation, must be performed sequentially

Time	Processors		
	1	2	3
1	$X[1]+=Y[1,1]$		
2	$X[2]+=Y[2,1]$	$X[1]+=Y[1,2]$	
3	$X[3]+=Y[3,1]$	$X[2]+=Y[2,2]$	$X[1]+=Y[1,3]$
4	$X[4]+=Y[4,1]$	$X[3]+=Y[3,2]$	$X[2]+=Y[2,3]$
5		$X[4]+=Y[4,2]$	$X[3]+=Y[3,3]$
6			$X[4]+=Y[4,3]$

Figure 11.49: Pipelined execution of Example 11.55 with $m = 4$ and $n = 3$.

because of the data dependence;[6] however, the different summation tasks are independent. We can parallelize this code by having each processor perform a separate summation. Processor i accesses row i of Y and updates the ith element of X.

Alternatively, we can structure the processors to execute the summation in a pipeline, and derive parallelism by overlapping the execution of the summations, as shown in Fig. 11.49. More specifically, each iteration of the inner loop can be treated as a stage of a pipeline: stage j takes an element of X generated in the previous stage, adds to it an element of Y, and passes the result to the next stage. Notice that in this case, each processor accesses a column, instead of a row, of Y. If Y is stored in column-major form, there is a gain in locality by partitioning according to columns, rather than by rows.

We can initiate a new task as soon as the first processor is done with the first stage of the previous task. At the beginning, the pipeline is empty and only the first processor is executing the first stage. After it completes, the results are passed to the second processor, while the first processor starts on the second task, and so on. In this way, the pipeline gradually fills until all the processors are busy. When the first processor finishes with the last task, the pipeline starts to drain, with more and more processors becoming idle until the last processor finishes the last task. In the steady state, n tasks can be executed concurrently in a pipeline of n processors.   □

It is interesting to contrast pipelining with simple parallelism, where different processors execute different tasks:

- Pipelining can only be applied to nests of depth at least two. We can treat each iteration of the outer loop as a task and the iterations in the inner loop as stages of that task.

- Tasks executed on a pipeline may share dependences. Information pertaining to the same stage of each task is held on the same processor; thus results generated by the ith stage of a task can be used by the ith stage

---

[6] Remember that we do not take advantage of the assumed commutativity and associativity of addition.

of subsequent tasks with no communication cost. Similarly, each input data element used by a single stage of different tasks needs to reside only on one processor, as illustrated by Example 11.55.

• If the tasks are independent, then simple parallelization has better processor utilization because processors can execute all at once without having to pay for the overhead of filling and draining the pipeline. However, as shown in Example 11.55, the pattern of data accesses in a pipelined scheme is different from that of simple parallelization. Pipelining may be preferable if it reduces communication.

11.9.2 Successive Over-Relaxation (SOR): An Example

Successive over-relaxation (SOR) is a technique for accelerating the convergence of relaxation methods for solving sets of simultaneous linear equations. A relatively simple template illustrating its data-access pattern is shown in Fig. 11.50(a). Here, the new value of an element in the array depends on the values of elements in its neighborhood. Such an operation is performed repeatedly, until some convergence criterion is met.

```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Original source.
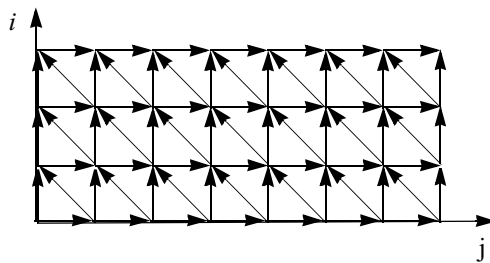
(b) Data dependences in the code.

Figure 11.50: An example of successive over-relaxation (SOR)

Shown in Fig. 11.50(b) is a picture of the key data dependences. We do not show dependences that can be inferred by the dependences already included in the figure. For example, iteration $[i, j]$ depends on iterations $[i, j-1], [i, j-2]$ and so on. It is clear from the dependences that there is no synchronization-free parallelism. Since the longest chain of dependences consists of $O(m+n)$ edges, by introducing synchronization, we should be able to find one degree of parallelism and execute the $O(mn)$ operations in $O(m+n)$ unit time.

In particular, we observe that iterations that lie along the 150° diagonals[7] in Fig. 11.50(b) do not share any dependences. They only depend on the iterations that lie along diagonals closer to the origin. Therefore we can parallelize this code by executing iterations on each diagonal in order, starting at the origin and proceeding outwards. We refer to the iterations along each diagonal as a *wavefront*, and such a parallelization scheme as *wavefronting*.

11.9.3 Fully Permutable Loops

We first introduce the notion of *full permutability*, a concept useful for pipelining and other optimizations. Loops are *fully permutable* if they can be permuted arbitrarily without changing the semantics of the original program. Once loops are put in a fully permutable form, we can easily pipeline the code and apply transformations such as blocking to improve data locality.

The SOR code, as it written in Fig. 11.50(a), is not fully permutable. As shown in Section 11.7.8, permuting two loops means that iterations in the original iteration space are executed column by column instead of row by row. For instance, the original computation in iteration [2,3] would execute before that of [1,4], violating the dependences shown in Fig. 11.50(b).

We can, however, transform the code to make it fully permutable. Applying the affine transform

$$\left[\begin{array}{cc} 1 & 0 \\ 1 & 1 \end{array} \right]$$

to the code yields the code shown in Fig. 11.51(a). This transformed code is fully permutable, and its permuted version is shown in Fig. 11.51(c). We also show the iteration space and data dependences of these two programs in Fig. 11.51(b) and (d), respectively. From the figure, we can easily see that this ordering preserves the relative ordering between every data-dependent pair of accesses.

When we permute loops, we change the set of operations executed in each iteration of the outermost loop drastically. The fact that we have this degree of freedom in scheduling means that there is a lot of slack in the ordering of operations in the program. Slack in scheduling means opportunities for parallelization. We show later in this section that if a nest has k outermost fully permutable loops, by introducing just $O(n)$ synchronizations, we can get $O(k-1)$ degrees of parallelism (n is the number of iterations in a loop).

11.9.4 Pipelining Fully Permutable Loops

A loop with k outermost fully permutable loops can be structured as a pipeline with $O(k-1)$ dimensions. In the SOR example, $k = 2$, so we can structure the processors as a linear pipeline.

---

[7] I.e., the sequences of points formed by repeatedly moving down 1 and right 2.

```
for (i = 0; i <= m; i++)
    for (j = i; j <= i+n; j++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])
```

(a) The code in Fig. 11.50 transformed by $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.
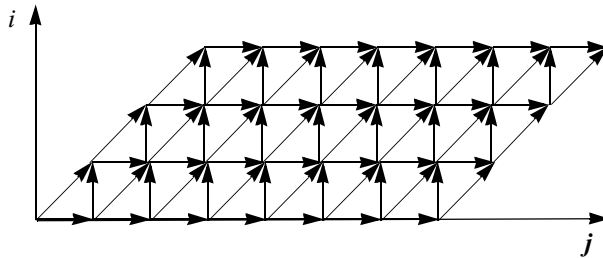
(b) Data dependences of the code in (a).

```
for (j = 0; j <= m+n; j++)
    for (i = max(0,j); i <= min(m,j), i++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])
```

(c) A permutation of the loops in (a).

(d) Data dependences of the code in (b).

Figure 11.51: Fully permutable version of the code Fig. 11.50

marks the end of the source file and is different from any possible character of the source program.

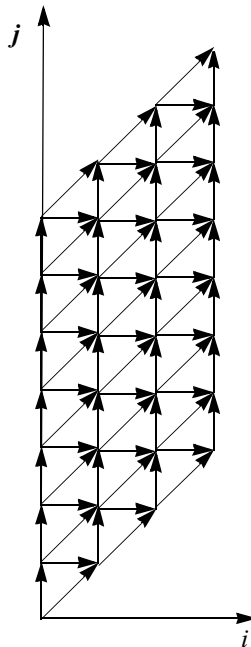Two pointers to the input are maintained:

1. Pointer `lexemeBegin`, marks the beginning of the current lexeme, whose extent we are attempting to determine.

2. Pointer `forward` scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, `forward` is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, `lexemeBegin` is set to the character immediately after the lexeme just found. In Fig. 3.3, we see `forward` has passed the end of the next lexeme, `**` (the Fortran exponentiation operator), and must be retracted one position to its left.

Advancing `forward` requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move `forward` to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

3.2.2   Sentinels

If we use the scheme of Section 3.2.1 as described, we must check, each time we advance `forward`, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a *sentinel* character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**.

Figure 3.4 shows the same arrangement as Fig. 3.3, but with the sentinels added. Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of a buffer means that the input is at an end. Figure 3.5 summarizes the algorithm for advancing `forward`. Notice how the first test, which can be part of a multiway branch based on the character pointed to by `forward`, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

3.3   Specification of Tokens

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in spec-

---

Can We Run Out of Buffer Space?

In most modern languages, lexemes are short, and one or two characters of lookahead is sufficient. Thus a buffer size N in the thousands is ample, and the double-buffer scheme of Section 3.2.1 works without problem. However, there are some risks. For example, if character strings can be very long, extending over many lines, then we could face the possibility that a lexeme is longer than N. To avoid problems with long character strings, we can treat them as a concatenation of components, one from each line over which the string is written. For instance, in Java it is conventional to represent long strings by writing a piece on each line and concatenating pieces with a + operator at the end of each piece.

A more difficult problem occurs when arbitrarily long lookahead may be needed. For example, some languages like PL/I do not treat keywords as *reserved*; that is, you can use identifiers with the same name as a keyword like DECLARE. If the lexical analyzer is presented with text of a PL/I program that begins DECLARE (ARG1, ARG2,... it cannot be sure whether DECLARE is a keyword, and ARG1 and so on are variables being declared, or whether DECLARE is a procedure name with its arguments. For this reason, modern languages tend to reserve their keywords. However, if not, one can treat a keyword like DECLARE as an ambiguous identifier, and let the parser resolve the issue, perhaps in conjunction with symbol-table lookup.

---

ifying those types of patterns that we actually need for tokens. In this section we shall study the formal notation for regular expressions, and in Section 3.5 we shall see how these expressions are used in a lexical-analyzer generator. Then, Section 3.7 shows how to build the lexical analyzer by converting regular expressions to automata that perform the recognition of the specified tokens.

3.3.1 Strings and Languages

An *alphabet* is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set $\{0, 1\}$ is the *binary alphabet*. ASCII is an important example of an alphabet; it is used in many software systems. Uni-
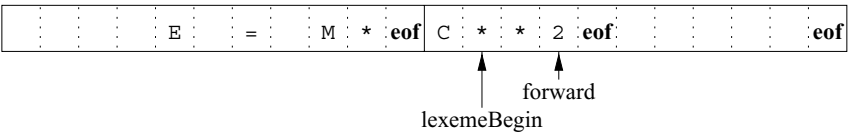

Figure 3.4: Sentinels at the end of each buffer

```
switch ( *forward++ ) {
      case eof:
            if (forward is at end of first buffer ) {
                  reload second buffer;
                  forward = beginning of second buffer;
            }
            else if (forward is at end of second buffer ) {
                  reload first buffer;
                  forward = beginning of first buffer;
            }
            else /* eof within a buffer marks the end of input */
                  terminate lexical analysis;
            break;
      Cases for the other characters
}
```

Figure 3.5: Lookahead code with sentinels

Implementing Multiway Branches

We might imagine that the switch in Fig. 3.5 requires many steps to ex-
ecute, and that placing the case **eof** first is not a wise choice. Actually,
it doesn't matter in what order we list the cases for each character. In
practice, a multiway branch depending on the input character is made in
one step by jumping to an address found in an array of addresses, indexed
by characters.

---

code, which includes approximately 100,000 characters from alphabets around
the world, is another important example of an alphabet.

A *string* over an alphabet is a finite sequence of symbols drawn from that
alphabet. In language theory, the terms "sentence" and "word" are often used
as synonyms for "string." The length of a string s, usually written $|s|$, is the
number of occurrences of symbols in s. For example, banana is a string of
length six. The *empty string*, denoted ϵ, is the string of length zero.

A *language* is any countable set of strings over some fixed alphabet. This
definition is very broad. Abstract languages like $\emptyset$, the *empty set*, or $\{\epsilon\}$, the
set containing only the empty string, are languages under this definition. So
too are the set of all syntactically well-formed C programs and the set of all
grammatically correct English sentences, although the latter two languages are
difficult to specify exactly. Note that the definition of "language" does not
require that any meaning be ascribed to the strings in the language. Methods
for defining the "meaning" of strings are discussed in Chapter 5.

<div style="border: 1px solid black; padding: 20px;">

Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string s is any string obtained by removing zero or more symbols from the end of s. For example, `ban`, `banana`, and ϵ are prefixes of `banana`.

2. A *suffix* of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, `nana`, `banana`, and ϵ are suffixes of `banana`.

3. A *substring* of s is obtained by deleting any prefix and any suffix from s. For instance, `banana`, `nan`, and ϵ are substrings of `banana`.

4. The *proper* prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ϵ or not equal to s itself.

5. A *subsequence* of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, `baan` is a subsequence of `banana`.

</div>

If x and y are strings, then the *concatenation* of x and y, denoted xy, is the string formed by appending y to x. For example, if $x = $ `dog` and $y = $ `house`, then $xy = $ `doghouse`. The empty string is the identity under concatenation; that is, for any string s, $\epsilon s = s\epsilon = s$.

If we think of concatenation as a product, we can define the "exponentiation" of strings as follows. Define s^0 to be ϵ, and for all $i > 0$, define s^i to be $s^{i-1}s$. Since $\epsilon s = s$, it follows that $s^1 = s$. Then $s^2 = ss$, $s^3 = sss$, and so on.

3.3.2 Operations on Languages

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 3.6. Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. The (*Kleene*) *closure* of a language L, denoted L^*, is the set of strings you get by concatenating L zero or more times. Note that L^0, the "concatenation of L zero times," is defined to be $\{\epsilon\}$, and inductively, L^i is $L^{i-1}L$. Finally, the positive closure, denoted L^+, is the same as the Kleene closure, but without the term L^0. That is, ϵ will not be in L^+ unless it is in L itself.

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Figure 3.6: Definitions of operations on languages

Example 3.3 : Let L be the set of letters $\{A, B, \ldots, Z, a, b, \ldots, z\}$ and let D be the set of digits $\{0, 1, \ldots 9\}$. We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators of Fig. 3.6:

1. $L \cup D$ is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.

2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.

3. L^4 is the set of all 4-letter strings.

4. L^* is the set of all strings of letters, including ϵ, the empty string.

5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.

6. D^+ is the set of all strings of one or more digits.

□

3.3.3   Regular Expressions

Suppose we wanted to describe the set of valid C identifiers. It is almost exactly the language described in item (5) above; the only difference is that the underscore is included among the letters.

   In Example 3.3, we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure. This process is so useful that a notation called *regular expressions* has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if *letter_* is established to stand for any letter or the underscore, and *digit* is

established to stand for any digit, then we could describe the language of C identifiers by:

$$letter\_ \ (\ letter\_ \ | \ digit \ )^*$$

The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and the juxtaposition of *letter_* with the remainder of the expression signifies concatenation.

The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language $L(r)$, which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.

2. If a is a symbol in Σ, then **a** is a regular expression, and $L(\mathbf{a}) = \{a\}$, that is, the language with one string, of length one, with a in its one position. Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.[1]

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages $L(r)$ and $L(s)$, respectively.

1. $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.

2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.

3. $(r)^*$ is a regular expression denoting $\big(L(r)\big)^*$.

4. (r) is a regular expression denoting $L(r)$. This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

a) The unary operator $*$ has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

---

[1] However, when talking about specific characters from the ASCII character set, we shall generally use teletype font for both the character and its regular expression.

c) | has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression $(\mathbf{a})|((\mathbf{b})^*(\mathbf{c}))$ by $\mathbf{a}|\mathbf{b}^*\mathbf{c}$. Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.

2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.

3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \dots\}$.

4. $(\mathbf{a}|\mathbf{b})^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.

5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \dots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

□

A language that can be defined by a regular expression is called a *regular set*. If two regular expressions r and s denote the same regular set, we say they are *equivalent* and write $r = s$. For instance, $(\mathbf{a}|\mathbf{b}) = (\mathbf{b}|\mathbf{a})$. There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure 3.7 shows some of the algebraic laws that hold for arbitrary regular expressions r, s, and t.

LAW	DESCRIPTION
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over |
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Figure 3.7: Algebraic laws for regular expressions

3.3.4 Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$$
\begin{aligned}
d_1 & \rightarrow & r_1 \\
d_2 & \rightarrow & r_2 \\
& \cdots & \\
d_n & \rightarrow & r_n
\end{aligned}
$$

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and

2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

By restricting r_i to Σ and the previously defined d's, we avoid recursive definitions, and we can construct a regular expression over Σ alone, for each r_i. We do so by first replacing uses of d_1 in r_2 (which cannot use any of the d's except for d_1), then replacing uses of d_1 and d_2 in r_3 by r_1 and (the substituted) r_2, and so on. Finally, in r_n we replace each d_i, for $i = 1, 2, \ldots, n-1$, by the substituted version of r_i, each of which has only symbols of Σ.

Example 3.5 : C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$
\begin{aligned}
letter\_ & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
id & \rightarrow & letter\_ \ (\ letter\_ \mid digit \ )^*
\end{aligned}
$$

□

Example 3.6 : Unsigned numbers (integer or floating point) are strings such as `5280`, `0.01234`, `6.336E4`, or `1.89E-4`. The regular definition

$$
\begin{aligned}
digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
digits & \rightarrow & digit \ digit^* \\
optionalFraction & \rightarrow & \texttt{.} \ digits \mid \epsilon \\
optionalExponent & \rightarrow & (\ \texttt{E} \ (\ \texttt{+} \mid \texttt{-} \mid \epsilon \ ) \ digits \ ) \mid \epsilon \\
number & \rightarrow & digits \ optionalFraction \ optionalExponent
\end{aligned}
$$

is a precise specification for this set of strings. That is, an *optionalFraction* is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string). An *optionalExponent*, if not missing, is the letter E followed by an optional + or − sign, followed by one or more digits. Note that at least one digit must follow the dot, so *number* does not match `1.`, but does match `1.0`.
□

3.3.5   Extensions of Regular Expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns. Here we mention a few notational extensions that were first incorporated into Unix utilities such as Lex that are particularly useful in the specification lexical analyzers. The references to this chapter contain a discussion of some regular-expression variants in use today.

1. *One or more instances.* The unary, postfix operator $^+$ represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then $(r)^+$ denotes the language $\big(L(r)\big)^+$. The operator $^+$ has the same precedence and associativity as the operator $*$. Two useful algebraic laws, $r^* = r^+|\epsilon$ and $r^+ = rr^* = r^*r$ relate the Kleene closure and positive closure.

2. *Zero or one instance.* The unary postfix operator ? means "zero or one occurrence." That is, r? is equivalent to $r|\epsilon$, or put another way, $L(r?) = L(r) \cup \{\epsilon\}$. The ? operator has the same precedence and associativity as $*$ and $+$.

3. *Character classes.* A regular expression $a_1|a_2|\cdots|a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2\cdots a_n]$. More importantly, when $a_1, a_2, \ldots, a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n, that is, just the first and last separated by a hyphen. Thus, [**abc**] is shorthand for **a**|**b**|**c**, and [**a-z**] is shorthand for **a**|**b**|$\cdots$|**z**.

Example 3.7 : Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$
\begin{array}{rcl}
letter\_ & \rightarrow & \texttt{[A-Za-z\_]} \\
digit & \rightarrow & \texttt{[0-9]} \\
id & \rightarrow & letter\_ \; (\; letter\_ \; | \; digit \;)^*
\end{array}
$$

The regular definition of Example 3.6 can also be simplified:

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^+ \\
number & \rightarrow & digits \; (\texttt{.} \; digits)? \; (\; \texttt{E} \; \texttt{[+-]}? \; digits \;)?
\end{array}
$$

$\square$

3.3.6 Exercises for Section 3.3

Exercise 3.3.1 : Consult the language reference manuals to determine (*i*) the sets of characters that form the input alphabet (excluding those that may only appear in character strings or comments), (*ii*) the lexical form of numerical constants, and (*iii*) the lexical form of identifiers, for each of the following languages: (a) C (b) C++ (c) C# (d) Fortran (e) Java (f) Lisp (g) SQL.

! Exercise 3.3.2 : Describe the languages denoted by the following regular expressions:

a) $\mathbf{a(a|b)^*a}$.

b) $((\epsilon|\mathbf{a})\mathbf{b}^*)^*$.

c) $(\mathbf{a|b})^*\mathbf{a(a|b)(a|b)}$.

d) $\mathbf{a^*ba^*ba^*ba^*}$.

!! e) $(\mathbf{aa|bb})^*((\mathbf{ab|ba})(\mathbf{aa|bb})^*(\mathbf{ab|ba})(\mathbf{aa|bb})^*)^*$.

Exercise 3.3.3 : In a string of length n, how many of the following are there?

a) Prefixes.

b) Suffixes.

c) Proper prefixes.

! d) Substrings.

! e) Subsequences.

Exercise 3.3.4 : Most languages are *case sensitive*, so keywords can be written only one way, and the regular expressions describing their lexemes are very simple. However, some languages, like SQL, are *case insensitive*, so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword `SELECT` can also be written `select`, `Select`, or `sElEcT`, for instance. Show how to write a regular expression for a keyword in a case-insensitive language. Illustrate the idea by writing the expression for "select" in SQL.

! Exercise 3.3.5 : Write regular definitions for the following languages:

a) All strings of lowercase letters that contain the five vowels in order.

b) All strings of lowercase letters in which the letters are in ascending lexicographic order.

c) Comments, consisting of a string surrounded by `/*` and `*/`, without an intervening `*/`, unless it is inside double-quotes (`"`).

!! d) All strings of digits with no repeated digits. *Hint*: Try this problem first
      with a few digits, such as $\{0, 1, 2\}$.

!! e) All strings of digits with at most one repeated digit.

!! f) All strings of *a*'s and *b*'s with an even number of *a*'s and an odd number
      of *b*'s.

   g) The set of Chess moves, in the informal notation, such as `p-k4` or `kbp×qn`.

!! h) All strings of *a*'s and *b*'s that do not contain the substring *abb*.

   i) All strings of *a*'s and *b*'s that do not contain the subsequence *abb*.

Exercise 3.3.6: Write character classes for the following sets of characters:

   a) The first ten letters (up to "j") in either upper or lower case.

   b) The lowercase consonants.

   c) The "digits" in a hexadecimal number (choose either upper or lower case
      for the "digits" above 9).

   d) The characters that can appear at the end of a legitimate English sentence
      (e.g., exclamation point).

   The following exercises, up to and including Exercise 3.3.10, discuss the
extended regular-expression notation from Lex (the lexical-analyzer generator
that we shall discuss extensively in Section 3.5). The extended notation is listed
in Fig. 3.8.

Exercise 3.3.7: Note that these regular expressions give all of the following
symbols (*operator characters*) a special meaning:

   \ " . ^ $ [ ] * + ? { } | /

Their special meaning must be turned off if they are needed to represent them-
selves in a character string. We can do so by quoting the character within a
string of length one or more; e.g., the regular expression `"**"` matches the string
**. We can also get the literal meaning of an operator character by preceding
it by a backslash. Thus, the regular expression `\*\*` also matches the string
**. Write a regular expression that matches the string `"\`.

Exercise 3.3.8: In Lex, a *complemented character class* represents any char-
acter except the ones listed in the character class. We denote a complemented
class by using ^ as the first character; this symbol (caret) is not itself part of
the class being complemented, unless it is listed within the class itself. Thus,
`[^A-Za-z]` matches any character that is not an uppercase or lowercase letter,
and `[^\^]` represents any character but the caret (or newline, since newline
cannot be in any character class). Show that for every regular expression with
complemented character classes, there is an equivalent regular expression with-
out complemented character classes.

```
Here is a photo of <B>my house</B>:
<P><IMG SRC = "house.gif"><BR>
See <A HREF = "morePix.html">More Pictures</A> if you
liked that one.<P>
```

into appropriate lexemes. Which lexemes should get associated lexical values, and what should those values be?

3.2 Input Buffering

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. The box on "Tricky Problems When Recognizing Tokens" in Section 3.1 gave an extreme example, but there are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id**. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large lookaheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

3.2.1 Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in Fig. 3.3.
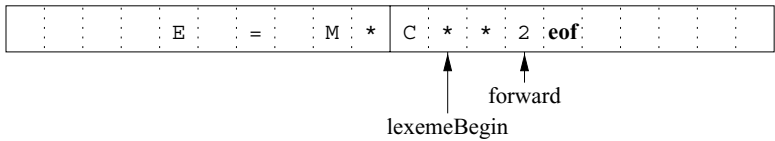

Figure 3.3: Using a pair of input buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by **eof**,

marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

1. Pointer `lexemeBegin`, marks the beginning of the current lexeme, whose extent we are attempting to determine.

2. Pointer `forward` scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, `forward` is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, `lexemeBegin` is set to the character immediately after the lexeme just found. In Fig. 3.3, we see `forward` has passed the end of the next lexeme, `**` (the Fortran exponentiation operator), and must be retracted one position to its left.

Advancing `forward` requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move `forward` to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

3.2.2 Sentinels

If we use the scheme of Section 3.2.1 as described, we must check, each time we advance `forward`, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a *sentinel* character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**.

Figure 3.4 shows the same arrangement as Fig. 3.3, but with the sentinels added. Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of a buffer means that the input is at an end. Figure 3.5 summarizes the algorithm for advancing `forward`. Notice how the first test, which can be part of a multiway branch based on the character pointed to by `forward`, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

3.3 Specification of Tokens

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in spec-

---

Can We Run Out of Buffer Space?

In most modern languages, lexemes are short, and one or two characters of lookahead is sufficient. Thus a buffer size N in the thousands is ample, and the double-buffer scheme of Section 3.2.1 works without problem. However, there are some risks. For example, if character strings can be very long, extending over many lines, then we could face the possibility that a lexeme is longer than N. To avoid problems with long character strings, we can treat them as a concatenation of components, one from each line over which the string is written. For instance, in Java it is conventional to represent long strings by writing a piece on each line and concatenating pieces with a + operator at the end of each piece.

A more difficult problem occurs when arbitrarily long lookahead may be needed. For example, some languages like PL/I do not treat keywords as *reserved*; that is, you can use identifiers with the same name as a keyword like DECLARE. If the lexical analyzer is presented with text of a PL/I program that begins DECLARE (ARG1, ARG2,... it cannot be sure whether DECLARE is a keyword, and ARG1 and so on are variables being declared, or whether DECLARE is a procedure name with its arguments. For this reason, modern languages tend to reserve their keywords. However, if not, one can treat a keyword like DECLARE as an ambiguous identifier, and let the parser resolve the issue, perhaps in conjunction with symbol-table lookup.

---

ifying those types of patterns that we actually need for tokens. In this section we shall study the formal notation for regular expressions, and in Section 3.5 we shall see how these expressions are used in a lexical-analyzer generator. Then, Section 3.7 shows how to build the lexical analyzer by converting regular expressions to automata that perform the recognition of the specified tokens.

3.3.1 Strings and Languages

An *alphabet* is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set $\{0, 1\}$ is the *binary alphabet*. ASCII is an important example of an alphabet; it is used in many software systems. Uni-
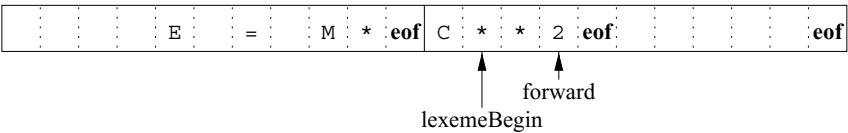

Figure 3.4: Sentinels at the end of each buffer

```
switch ( *forward++ ) {
      case eof:
            if (forward is at end of first buffer ) {
                  reload second buffer;
                  forward = beginning of second buffer;
            }
            else if (forward is at end of second buffer ) {
                  reload first buffer;
                  forward = beginning of first buffer;
            }
            else /* eof within a buffer marks the end of input */
                  terminate lexical analysis;
            break;
      Cases for the other characters
}
```

Figure 3.5: Lookahead code with sentinels

Implementing Multiway Branches

We might imagine that the switch in Fig. 3.5 requires many steps to execute, and that placing the case **eof** first is not a wise choice. Actually, it doesn't matter in what order we list the cases for each character. In practice, a multiway branch depending on the input character is made in one step by jumping to an address found in an array of addresses, indexed by characters.

---

code, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.

A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written $|s|$, is the number of occurrences of symbols in s. For example, banana is a string of length six. The *empty string*, denoted ϵ, is the string of length zero.

A *language* is any countable set of strings over some fixed alphabet. This definition is very broad. Abstract languages like $\emptyset$, the *empty set*, or $\{\epsilon\}$, the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed C programs and the set of all grammatically correct English sentences, although the latter two languages are difficult to specify exactly. Note that the definition of "language" does not require that any meaning be ascribed to the strings in the language. Methods for defining the "meaning" of strings are discussed in Chapter 5.

Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string s is any string obtained by removing zero or more symbols from the end of s. For example, `ban`, `banana`, and ϵ are prefixes of `banana`.

2. A *suffix* of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, `nana`, `banana`, and ϵ are suffixes of `banana`.

3. A *substring* of s is obtained by deleting any prefix and any suffix from s. For instance, `banana`, `nan`, and ϵ are substrings of `banana`.

4. The *proper* prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ϵ or not equal to s itself.

5. A *subsequence* of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, `baan` is a subsequence of `banana`.

If x and y are strings, then the *concatenation* of x and y, denoted xy, is the string formed by appending y to x. For example, if $x = $ `dog` and $y = $ `house`, then $xy = $ `doghouse`. The empty string is the identity under concatenation; that is, for any string s, $\epsilon s = s\epsilon = s$.

If we think of concatenation as a product, we can define the "exponentiation" of strings as follows. Define s^0 to be ϵ, and for all $i > 0$, define s^i to be $s^{i-1}s$. Since $\epsilon s = s$, it follows that $s^1 = s$. Then $s^2 = ss$, $s^3 = sss$, and so on.

3.3.2 Operations on Languages

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 3.6. Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. The (*Kleene*) *closure* of a language L, denoted L^*, is the set of strings you get by concatenating L zero or more times. Note that L^0, the "concatenation of L zero times," is defined to be $\{\epsilon\}$, and inductively, L^i is $L^{i-1}L$. Finally, the positive closure, denoted L^+, is the same as the Kleene closure, but without the term L^0. That is, ϵ will not be in L^+ unless it is in L itself.

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s$ is in L or s is in $M\}$
Concatenation of L and M	$LM = \{st \mid s$ is in L and t is in $M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Figure 3.6: Definitions of operations on languages

Example 3.3 : Let L be the set of letters $\{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$ and let D be the set of digits $\{\texttt{0}, \texttt{1}, \ldots \texttt{9}\}$. We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators of Fig. 3.6:

1. $L \cup D$ is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.

2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.

3. L^4 is the set of all 4-letter strings.

4. L^* is the set of all strings of letters, including ϵ, the empty string.

5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.

6. D^+ is the set of all strings of one or more digits.

□

3.3.3 Regular Expressions

Suppose we wanted to describe the set of valid C identifiers. It is almost exactly the language described in item (5) above; the only difference is that the underscore is included among the letters.

In Example 3.3, we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure. This process is so useful that a notation called *regular expressions* has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if *letter_* is established to stand for any letter or the underscore, and *digit* is

established to stand for any digit, then we could describe the language of C identifiers by:

$$letter\_ \ (\ letter\_\ |\ digit\ )^*$$

The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and the juxtaposition of *letter_* with the remainder of the expression signifies concatenation.

The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language $L(r)$, which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.

2. If a is a symbol in Σ, then **a** is a regular expression, and $L(\mathbf{a}) = \{a\}$, that is, the language with one string, of length one, with a in its one position. Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.[1]

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages $L(r)$ and $L(s)$, respectively.

1. $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.

2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.

3. $(r)^*$ is a regular expression denoting $\big(L(r)\big)^*$.

4. (r) is a regular expression denoting $L(r)$. This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

a) The unary operator $*$ has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

---

[1] However, when talking about specific characters from the ASCII character set, we shall generally use teletype font for both the character and its regular expression.

c) | has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression
$(\mathbf{a})|((\mathbf{b})^*(\mathbf{c}))$ by $\mathbf{a}|\mathbf{b}^*\mathbf{c}$. Both expressions denote the set of strings that are either
a single a or are zero or more b's followed by one c.

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.

2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two
   over the alphabet Σ. Another regular expression for the same language is
   $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.

3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that
   is, $\{\epsilon, a, aa, aaa, \dots\}$.

4. $(\mathbf{a}|\mathbf{b})^*$ denotes the set of all strings consisting of zero or more instances
   of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$.
   Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.

5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \dots\}$, that is, the string a
   and all strings consisting of zero or more a's and ending in b.

$\Box$

A language that can be defined by a regular expression is called a *regular
set*. If two regular expressions r and s denote the same regular set, we say they
are *equivalent* and write $r = s$. For instance, $(\mathbf{a}|\mathbf{b}) = (\mathbf{b}|\mathbf{a})$. There are a number
of algebraic laws for regular expressions; each law asserts that expressions of
two different forms are equivalent. Figure 3.7 shows some of the algebraic laws
that hold for arbitrary regular expressions r, s, and t.

LAW	DESCRIPTION
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over |
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Figure 3.7: Algebraic laws for regular expressions

3.3.4 Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$$
\begin{aligned}
d_1 &\rightarrow r_1 \\
d_2 &\rightarrow r_2 \\
&\cdots \\
d_n &\rightarrow r_n
\end{aligned}
$$

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and

2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

By restricting r_i to Σ and the previously defined d's, we avoid recursive definitions, and we can construct a regular expression over Σ alone, for each r_i. We do so by first replacing uses of d_1 in r_2 (which cannot use any of the d's except for d_1), then replacing uses of d_1 and d_2 in r_3 by r_1 and (the substituted) r_2, and so on. Finally, in r_n we replace each d_i, for $i = 1, 2, \ldots, n-1$, by the substituted version of r_i, each of which has only symbols of Σ.

Example 3.5 : C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$
\begin{aligned}
letter\_ &\rightarrow \quad \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
digit &\rightarrow \quad \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
id &\rightarrow \quad letter\_ \; (\; letter\_ \mid digit \;)^*
\end{aligned}
$$

□

Example 3.6 : Unsigned numbers (integer or floating point) are strings such as $\texttt{5280}$, $\texttt{0.01234}$, $\texttt{6.336E4}$, or $\texttt{1.89E-4}$. The regular definition

$$
\begin{aligned}
digit &\rightarrow \quad \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
digits &\rightarrow \quad digit \; digit^* \\
optionalFraction &\rightarrow \quad \texttt{.} \; digits \mid \epsilon \\
optionalExponent &\rightarrow \quad (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; digits \;) \mid \epsilon \\
number &\rightarrow \quad digits \; optionalFraction \; optionalExponent
\end{aligned}
$$

is a precise specification for this set of strings. That is, an *optionalFraction* is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string). An *optionalExponent*, if not missing, is the letter E followed by an optional + or − sign, followed by one or more digits. Note that at least one digit must follow the dot, so *number* does not match $\texttt{1.}$, but does match $\texttt{1.0}$. □

3.3.5 Extensions of Regular Expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns. Here we mention a few notational extensions that were first incorporated into Unix utilities such as Lex that are particularly useful in the specification lexical analyzers. The references to this chapter contain a discussion of some regular-expression variants in use today.

1. *One or more instances.* The unary, postfix operator $^+$ represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then $(r)^+$ denotes the language $(L(r))^+$. The operator $^+$ has the same precedence and associativity as the operator $*$. Two useful algebraic laws, $r^* = r^+|\epsilon$ and $r^+ = rr^* = r^*r$ relate the Kleene closure and positive closure.

2. *Zero or one instance.* The unary postfix operator ? means "zero or one occurrence." That is, r? is equivalent to $r|\epsilon$, or put another way, $L(r?) = L(r) \cup \{\epsilon\}$. The ? operator has the same precedence and associativity as $*$ and $+$.

3. *Character classes.* A regular expression $a_1|a_2|\cdots|a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2\cdots a_n]$. More importantly, when $a_1, a_2, \ldots, a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n, that is, just the first and last separated by a hyphen. Thus, [**abc**] is shorthand for **a**|**b**|**c**, and [**a-z**] is shorthand for **a**|**b**|$\cdots$|**z**.

Example 3.7: Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$
\begin{aligned}
letter\_ &\rightarrow & [\texttt{A-Za-z\_}] \\
digit &\rightarrow & [\texttt{0-9}] \\
id &\rightarrow & letter\_ \ (\ letter\_ \ | \ digit\ )^*
\end{aligned}
$$

The regular definition of Example 3.6 can also be simplified:

$$
\begin{aligned}
digit &\rightarrow & [\texttt{0-9}] \\
digits &\rightarrow & digit^+ \\
number &\rightarrow & digits \ (.\ digits)? \ (\ \texttt{E}\ [\texttt{+-}]?\ digits\ )?
\end{aligned}
$$

□

3.3.6 Exercises for Section 3.3

Exercise 3.3.1 : Consult the language reference manuals to determine (*i*) the sets of characters that form the input alphabet (excluding those that may only appear in character strings or comments), (*ii*) the lexical form of numerical constants, and (*iii*) the lexical form of identifiers, for each of the following languages: (a) C (b) C++ (c) C# (d) Fortran (e) Java (f) Lisp (g) SQL.

! Exercise 3.3.2 : Describe the languages denoted by the following regular expressions:

    a) **a(a|b)*a**.

    b) **((ϵ|a)b*)***.

    c) **(a|b)*a(a|b)(a|b)**.

    d) **a*ba*ba*ba***.

!! e) **(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)***.

Exercise 3.3.3 : In a string of length n, how many of the following are there?

    a) Prefixes.

    b) Suffixes.

    c) Proper prefixes.

  ! d) Substrings.

  ! e) Subsequences.

Exercise 3.3.4 : Most languages are *case sensitive*, so keywords can be written only one way, and the regular expressions describing their lexemes are very simple. However, some languages, like SQL, are *case insensitive*, so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword `SELECT` can also be written `select`, `Select`, or `sElEcT`, for instance. Show how to write a regular expression for a keyword in a case-insensitive language. Illustrate the idea by writing the expression for "select" in SQL.

! Exercise 3.3.5 : Write regular definitions for the following languages:

    a) All strings of lowercase letters that contain the five vowels in order.

    b) All strings of lowercase letters in which the letters are in ascending lexicographic order.

    c) Comments, consisting of a string surrounded by `/*` and `*/`, without an intervening `*/`, unless it is inside double-quotes (`"`).

!! d) All strings of digits with no repeated digits. *Hint*: Try this problem first with a few digits, such as $\{0, 1, 2\}$.

!! e) All strings of digits with at most one repeated digit.

!! f) All strings of a's and b's with an even number of a's and an odd number of b's.

g) The set of Chess moves, in the informal notation, such as `p-k4` or `kbp×qn`.

!! h) All strings of a's and b's that do not contain the substring *abb*.

i) All strings of a's and b's that do not contain the subsequence *abb*.

Exercise 3.3.6 : Write character classes for the following sets of characters:

a) The first ten letters (up to "j") in either upper or lower case.

b) The lowercase consonants.

c) The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).

d) The characters that can appear at the end of a legitimate English sentence (e.g., exclamation point).

The following exercises, up to and including Exercise 3.3.10, discuss the extended regular-expression notation from Lex (the lexical-analyzer generator that we shall discuss extensively in Section 3.5). The extended notation is listed in Fig. 3.8.

Exercise 3.3.7 : Note that these regular expressions give all of the following symbols (*operator characters*) a special meaning:

```
\ " . ^ $ [ ] * + ? { } | /
```

Their special meaning must be turned off if they are needed to represent themselves in a character string. We can do so by quoting the character within a string of length one or more; e.g., the regular expression `"**"` matches the string `**`. We can also get the literal meaning of an operator character by preceding it by a backslash. Thus, the regular expression `\*\*` also matches the string `**`. Write a regular expression that matches the string `"\`.

Exercise 3.3.8 : In Lex, a *complemented character class* represents any character except the ones listed in the character class. We denote a complemented class by using ^ as the first character; this symbol (caret) is not itself part of the class being complemented, unless it is listed within the class itself. Thus, `[^A-Za-z]` matches any character that is not an uppercase or lowercase letter, and `[^\^]` represents any character but the caret (or newline, since newline cannot be in any character class). Show that for every regular expression with complemented character classes, there is an equivalent regular expression without complemented character classes.

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
$\backslash c$	character c literally	`\*`
`"s"`	string s literally	`"**"`
.	any character but newline	`a.*b`
^	beginning of a line	`^abc`
$	end of a line	`abc$`
$[s]$	any one of the characters in string s	`[abc]`
$[\hat{\ }s]$	any one character not in string s	`[^abc]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one r	`a?`
$r\{m,n\}$	between m and n occurrences of r	`a{1,5}`
$r_1 r_2$	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

Figure 3.8: `Lex` regular expressions

! Exercise 3.3.9 : The regular expression $r\{m,n\}$ matches from m to n occurrences of the pattern r. For example, `a{1,5}` matches a string of one to five a's. Show that for every regular expression containing repetition operators of this form, there is an equivalent regular expression without repetition operators.

! Exercise 3.3.10 : The operator ^ matches the left end of a line, and $ matches the right end of a line. The operator ^ is also used to introduce complemented character classes, but the context always makes it clear which meaning is intended. For example, `^[^aeiou]*$` matches any complete line that does not contain a lowercase vowel.

    a) How do you tell which meaning of ^ is intended?

    b) Can you always replace a regular expression using the ^ and $ operators by an equivalent expression that does not use either of these operators?

! Exercise 3.3.11 : The UNIX shell command `sh` uses the operators in Fig. 3.9 in filename expressions to describe sets of file names. For example, the filename expression `*.o` matches all file names ending in `.o`; `sort1.?` matches all filenames of the form `sort1.`c, where c is any character. Show how `sh` filename