Processors are being produced with the potential for very many parallel operations on the instruction level....Far greater extremes in instruction-level parallelism are on the horizon.

J. Fisher [1981], in the paper that inaugurated the term “instruction-level parallelism”

One of the surprises about IA-64 is that we hear no claims of high frequency, despite claims that an EPIC processor is less complex than a superscalar processor. It’s hard to know why this is so, but one can speculate that the overall complexity involved in focusing on CPI, as IA-64 does, makes it hard to get high megahertz.

M. Hopkins [2000], in a commentary on the IA-64 architecture, a joint development of HP and Intel designed to achieve dramatic increases in the exploitation of ILP while retaining a simple architecture, which would allow higher performance.
4.1 Basic Compiler Techniques for Exposing ILP

This chapter starts by examining the use of compiler technology to improve the performance of pipelines and simple multiple-issue processors. These techniques are key even for processors that make dynamic issue decisions but use static scheduling and are crucial for processors that use static issue or static scheduling. After applying these concepts to reducing stalls from data hazards in single issue pipelines, we examine the use of compiler-based techniques for branch prediction. Armed with this more powerful compiler technology, we examine the design and performance of multiple-issue processors using static issuing or scheduling. Sections 4.4 and 4.5 examine more advanced software and hardware techniques, designed to enable a processor to exploit more instruction-level parallelism. Putting It All Together examines the IA-64 architecture and its first implementation, Itanium. Two different static, VLIW-style processors are covered in Another View.
Basic Pipeline Scheduling and Loop Unrolling

To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction. A compiler’s ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline. Throughout this chapter we will assume the FP unit latencies shown in Figure 4.1, unless different latencies are explicitly stated.

We assume the standard 5-stage integer pipeline, so that branches have a delay of one clock cycle. We assume that the functional units are fully pipelined or replicated (as many times as the pipeline depth), so that an operation of any type can be issued on every clock cycle and there are no structural hazards.

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>

**FIGURE 4.1 Latencies of FP operations used in this chapter.** The first column shows the originating instruction type. The second column is the type of the consuming instruction. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is zero, since the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0.

In this subsection, we look at how the compiler can increase the amount of available ILP by unrolling loops. This example serves both to illustrate an important technique as well as to motivate the more powerful program transformations described later in this chapter. We will rely on an example similar to the one we used in the last chapter, adding a scalar to a vector:

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

We can see that this loop is parallel by noticing that the body of each iteration is independent. We will formalize this notion later in this chapter and describe how we can test whether loop iterations are independent at compile-time. First, let’s look at the performance of this loop, showing how we can use the parallelism to improve its performance for a MIPS pipeline with the latencies shown above.

The first step is to translate the above segment to MIPS assembly language. In the following code segment, $r1$ is initially the address of the element in the array
with the highest address, and F2 contains the scalar value, s. Register R2 is pre-computed, so that 8(R2) is the last element to operate on.

The straightforward MIPS code, not scheduled for the pipeline, looks like this:

```
Loop: L.D F0,0(R1) ;F0=array element
       ADD.D F4,F0,F2 ;add scalar in F2
       S.D F4,0(R1) ;store result
       DADDUI R1,R1,#-8 ;decrement pointer
                       ;8 bytes (per DW)
       BNE R1,R2,Loop ;branch R1!=zero
```

Let’s start by seeing how well this loop will run when it is scheduled on a simple pipeline for MIPS with the latencies from Figure 4.1.

**EXAMPLE**
Show how the loop would look on MIPS, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for both delays from floating-point operations and from the delayed branch.

**ANSWER**
Without any scheduling the loop will execute as follows:

<table>
<thead>
<tr>
<th>Clock cycle issued</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: L.D F0,0(R1)</td>
</tr>
<tr>
<td>stall</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
</tr>
<tr>
<td>stall</td>
</tr>
<tr>
<td>stall</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
</tr>
<tr>
<td>DADDUI R1,R1,#-8</td>
</tr>
<tr>
<td>stall</td>
</tr>
<tr>
<td>BNE R1,R2,Loop</td>
</tr>
<tr>
<td>stall</td>
</tr>
</tbody>
</table>

This code requires 10 clock cycles per iteration. We can schedule the loop to obtain only one stall:

```
Loop: L.D F0,0(R1)
       DADDUI R1,R1,#-8
       ADD.D F4,F0,F2
       stall
       BNE R1,R2,Loop ;delayed branch
       S.D F4,8(R1) ;altered & interchanged
                      with DADDUI
```

Execution time has been reduced from 10 clock cycles to 6. The stall after ADD.D is for the use by the S.D.
Notice that to schedule the delayed branch, the compiler had to determine that it could swap the `DADDUI` and `S.D` by changing the address to which the `S.D` stored: the address was `0(R1)` and is now `8(R1)`. This change is not trivial, since most compilers would see that the `S.D` instruction depends on the `DADDUI` and would refuse to interchange them. A smarter compiler, capable of limited symbolic optimization, could figure out the relationship and perform the interchange. The chain of dependent instructions from the `L.D` to the `ADD.D` and then to the `S.D` determines the clock cycle count for this loop. This chain must take at least 6 cycles because of dependencies and pipeline latencies.

In the above example, we complete one loop iteration and store back one array element every 6 clock cycles, but the actual work of operating on the array element takes just 3 (the load, add, and store) of those 6 clock cycles. The remaining 3 clock cycles consist of loop overhead—the `DADDUI` and `BNE`—and a stall. To eliminate these 3 clock cycles we need to get more operations within the loop relative to the number of overhead instructions.

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is *loop unrolling*. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together. In this case, we can eliminate the data use stall by creating additional independent instructions within the loop body. If we simply replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop. Thus, we will want to use different registers for each iteration, increasing the required register count.

---

**Example**

Show our loop unrolled so that there are four copies of the loop body, assuming `R1` is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4. Eliminate any obviously redundant computations and do not reuse any of the registers.

**Answer**

Here is the result after merging the `DADDUI` instructions and dropping the unnecessary `BNE` operations that are duplicated during unrolling. Note that `R2` must now be set so that `32(R2)` is the starting address of the last four elements.

```plaintext
Loop:  L.D   F0, 0(R1)
       ADD.D F4, F0, F2
       S.D F4, 0(R1) ;drop DADDUI & BNE
       L.D F6, -8(R1)
       ADD.D F8, F6, F2
       S.D F8, -8(R1) ;drop DADDUI & BNE
       L.D F10, -16(R1)
```
ADD.D F12, F10, F2
S.D F12, -16(R1) ; drop DADDUI & BNE
L.D F14, -24(R1)
ADD.D F16, F14, F2
S.D F16, -24(R1)
DADDUI R1, R1, # -32
BNE R1, R2, Loop

We have eliminated three branches and three decrements of R1. The addresses on the loads and stores have been compensated to allow the DADDUI instructions on R1 to be merged. This optimization may seem trivial, but it is not; it requires symbolic substitution and simplification. We will see more general forms of these optimizations that eliminate dependent computations in Section 4.4.

Without scheduling, every operation in the unrolled loop is followed by a dependent operation and thus will cause a stall. This loop will run in 28 clock cycles—each L.D has 1 stall, each ADD.D 2, the DADDUI 1, the branch 1, plus 14 instruction issue cycles—or 7 clock cycles for each of the four elements. Although this unrolled version is currently slower than the scheduled version of the original loop, this will change when we schedule the unrolled loop. Loop unrolling is normally done early in the compilation process, so that redundant computations can be exposed and eliminated by the optimizer.

In real programs we do not usually know the upper bound on the loop. Suppose it is \( n \), and we would like to unroll the loop to make \( k \) copies of the body. Instead of a single unrolled loop, we generate a pair of consecutive loops. The first executes \((n \mod k)\) times and has a body that is the original loop. The second is the unrolled body surrounded by an outer loop that iterates \((n/k)\) times. For large values of \( n \), most of the execution time will be spent in the unrolled loop body.

In the above Example, unrolling improves the performance of this loop by eliminating overhead instructions, although it increases code size substantially. How will the unrolled loop perform when it is scheduled for the pipeline described earlier?

**EXAMPLE**
Show the unrolled loop in the previous example after it has been scheduled for the pipeline with the latencies shown in Figure 4.1 on page 222.

**ANSWER**
Loop: L.D F0, 0(R1)
L.D F6, -8(R1)
L.D F10, -16(R1)
L.D F14, -24(R1)
ADD.D F4, F0, F2
The execution time of the unrolled loop has dropped to a total of 14 clock cycles, or 3.5 clock cycles per element, compared with 7 cycles per element before scheduling and 6 cycles when scheduled but not unrolled.

The gain from scheduling on the unrolled loop is even larger than on the original loop. This increase arises because unrolling the loop exposes more computation that can be scheduled to minimize the stalls; the code above has no stalls. Scheduling the loop in this fashion necessitates realizing that the loads and stores are independent and can be interchanged.

Summary of the Loop Unrolling and Scheduling Example
Throughout this chapter we will look at a variety of hardware and software techniques that allow us to take advantage of instruction-level parallelism to fully utilize the potential of the functional units in a processor. The key to most of these techniques is to know when and how the ordering among instructions may be changed. In our example we made many such changes, which to us, as human beings, were obviously allowable. In practice, this process must be performed in a methodical fashion either by a compiler or by hardware. To obtain the final unrolled code we had to make the following decisions and transformations:

1. Determine that it was legal to move the `s.d` after the `daddui` and `bne`, and find the amount to adjust the `s.d` offset.

2. Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.

3. Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations.

4. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.

5. Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This transformation requires analyzing the memory addresses and finding that they do not refer to the same address.
6. Schedule the code, preserving any dependences needed to yield the same result as the original code.

The key requirement underlying all of these transformations is an understanding of how an instruction depends on another and how the instructions can be changed or reordered given the dependences. Before examining how these techniques work for higher issue rate pipelines, let us examine how the loop unrolling and scheduling techniques affect data dependences.

**EXAMPLE**  
Show how the process of optimizing the loop overhead by unrolling the loop actually eliminates data dependences. In this example and those used in the remainder of this chapter, we use nondelayed branches for simplicity; it is easy to extend the examples to use delayed branches.

**ANSWER**  
Here is the unrolled but unoptimized code with the extra **DADDUI** instructions, but without the branches. (Eliminating the branches is another type of transformation, since it involves control rather than data.) The arrows show the data dependences that are within the unrolled body and involve the **DADDUI** instructions. The underlined registers are the dependent uses.

```
Loop:   L.D  F0,0(R1)
       ADD.D F4,F0,F2
       S.D  F4,0(R1)
       DADDUI R1,R1,#-8 ;drop BNE
       L.D  F6,0(R1)
       ADD.D F8,F6,F2
       S.D  F8,0(R1)
       DADDUI R1,R1,#-8 ;drop BNE
       L.D  F10,0(R1)
       ADD.D F12,F10,F2
       S.D  F12,0(R1)
       DADDUI R1,R1,#-8 ;drop BNE
       L.D  F14,0(R1)
       ADD.D F16,F14,F2
       S.D  F16,0(R1)
       DADDUI R1,R1,#-8
       BNE  R1,R2,LOOP
```

As the arrows show, the **DADDUI** instructions form a dependent chain that involves the **DADDUI**, **L.D**, and **S.D** instructions. This chain forces the body to execute in order, as well as making the **DADDUI** instructions necessary, which increases the instruction count. The compiler removes this dependence by symbolically computing the intermediate values of **R1** and fold-
ing the computation into the offset of the \texttt{L.D} and \texttt{S.D} instructions and by changing the final \texttt{DADDUI} into a decrement by 32. This transformation makes the three \texttt{DADDUI} unnecessary, and the compiler can remove them. There are other types of dependences in this code, as the next few example show.

\textbf{Example} Unroll our example loop, eliminating the excess loop overhead, but using the same registers in each loop copy. Indicate both the data and name dependences within the body. Show how renaming eliminates name dependences that reduce parallelism.

\textbf{Answer} Here's the loop unrolled but with the same registers in use for each copy. The data dependences are shown with gray arrows and the name dependences with black arrows. As in earlier examples, the direction of the arrow indicates the ordering that must be preserved for correct execution of the code:

\begin{verbatim}
Loop: L.D    F0, 0 (R1)
       ADD.D F4, F0, F2
       S.D    F4, 0 (R1) ;drop DADDUI & BNE
       L.D    F0, -8 (R1)
       ADD.D F4, F0, F2
       S.D    F4, -8 (R1) ;drop DADDUI & BNE
       L.D    F0, -16 (R1)
       ADD.D F4, F0, F2
       S.D    F4, -16 (R1)
       L.D    F0, -24 (R1)
       ADD.D F4, F0, F2
       S.D    F4, -24 (R1)
       DADDUI R1, R1, # -32
       BNE R1, R2, LOOP
\end{verbatim}

The name dependences force the instructions in the loop to be almost completely ordered, allowing only the order of the \texttt{L.D} following each \texttt{S.D} to be interchanged. When the registers used for each copy of the loop
body are renamed only the true dependences within each body remain:

```
Loop:  L.D  F0, 0 (R1)
      ADD.D F4, F0, F2
      S.D  F4, 0 (R1)    ;drop DADDUI & BNE
      L.D  F6, -8 (R1)
      ADD.D F8, F6, F2
      S.D  F8, -8 (R1)    ;drop DADDUI & BNE
      L.D  F10, -16 (R1)
      ADD.D F12, F10, F2
      S.D  F12, -16 (R1)
      L.D  F14, -24 (R1)
      ADD.D F16, F14, F2
      S.D  F16, -24 (R1)
      DADDUI R1, R1, # -32
      BNE  R1, R2, LOOP
```

With the renaming, the copies of each loop body become independent
and can be overlapped or executed in parallel. This renaming process can
be performed either by the compiler or in hardware, as we saw in the last
chapter.

There are three different types of limits to the gains that can be achieved by
loop unrolling: a decrease in the amount of overhead amortized with each unroll,
code size limitations, and compiler limitations. Let’s consider the question of loop
overhead first. When we unrolled the loop four times, it generated sufficient parallelism
among the instructions that the loop could be scheduled with no stall cycles. In fact, in fourteen clock cycles, only two cycles were loop overhead: the
DSUBI, which maintains the index value, and the BNE, which terminates the
loop. If the loop is unrolled eight times, the overhead is reduced from 1/2 cycle per
original iteration to 1/4. One of the exercises asks you to compute the theoretically
optimal number of times to unroll this loop for a random number of iterations.

A second limit to unrolling is the growth in code size that results. For larger
loops, the code size growth may be a concern either in the embedded space where
memory may be at a premium or if the larger code size causes a decrease in the instruction cache miss rate. We return to the issue of code size when we consider
more aggressive techniques for uncovering instruction level parallelism in Section
4.4.

Another factor often more important than code size is the potential shortfall in
registers that is created by aggressive unrolling and scheduling. This secondary
affect that results from instruction scheduling in large code segments is called register pressure. It arises because scheduling code to increase ILP causes the number
of live values to increase. After aggressive instruction scheduling, it not be possi-
ble to allocate all the live values to registers. The transformed code, while theoretically faster, may lose some or all of its advantage, because it generates a shortage of registers. Without unrolling, aggressive scheduling is sufficiently limited by branches so that register pressure is rarely a problem. The combination of unrolling and aggressive scheduling can, however, cause this problem. The problem becomes especially challenging in multiple issue machines that require the exposure of more independent instruction sequences whose execution can be overlapped. In general, the use of sophisticated high-level transformations, whose potential improvements are hard to measure before detailed code generation, has led to significant increases in the complexity of modern compilers.

Loop unrolling is a simple but useful method for increasing the size of straight-line code fragments that can be scheduled effectively. This transformation is useful in a variety of processors, from simple pipelines like those in MIPS to the statically scheduled superscalars we described in the last chapter, as we will see now.

Using Loop Unrolling and Pipeline Scheduling with Static Multiple Issue

We begin by looking at a simple two-issue, statically-scheduled superscalar MIPS pipeline from the last chapter, using the pipeline latencies from Figure 4.1 on page 222 and the same example code segment we used for the single issue examples above. This processor can issue two instructions per clock cycle, where one of the instructions can be a load, store, branch, or integer ALU operation, and the other can be any floating-point operation.

Recall that this pipeline did not generate a significant performance enhancement for the example above, because of the limited ILP in a given loop iteration. Let’s see how loop unrolling and pipeline scheduling can help.

**EXAMPLE** Unroll and schedule the loop used in the earlier examples and shown on page 223.

**ANSWER** To schedule this loop without any delays, we will need to unroll the loop to make five copies of the body. After unrolling, the loop will contain five each of L.D, ADD.D, and S.D; one DADDUI; and one BNE. The unrolled and scheduled code is shown in Figure 4.2. This unrolled superscalar loop now runs in 12 clock cycles per iteration, or 2.4 clock cycles per element, versus 3.5 for the scheduled and unrolled loop on the ordinary MIPS pipeline. In this Example, the performance of the superscalar MIPS is limited by the balance between integer and floating-point computation. Every floating-point instruction is issued together with an integer instruction, but there are not enough floating-point instructions to keep the floating-point pipeline full. When scheduled, the original loop ran in 6 clock cycles per iteration. We have improved on that by a factor of 2.5, more than half of which came from loop unrolling. Loop unrolling took us from 6 to 3.5 (a factor of 1.7), while superscalar execution gave us
4.2 Static Branch Prediction

In Chapter 3, we examined the use of dynamic branch predictors. Static branch predictors are sometimes used in processors where the expectation is that branch behavior is highly predictable at compile-time; static prediction can also be used to assist dynamic predictors.

In Chapter 1, we discussed an architectural feature that supports static branch predication, namely delayed branches. Delayed branches expose a pipeline hazard so that the compiler can reduce the penalty associated with the hazard. As we saw, the effectiveness of this technique partly depends on whether we correctly guess which way a branch will go. Being able to accurately predict a branch at compile time is also helpful for scheduling data hazards. Loop unrolling is on simple example of this; another example, arises from conditional selection branches. Consider the following code segment:

```
LD     R1, 0(R2)
DSUBU  R1, R1, R3
BEQZ   R1, L
OR     R4, R5, R6
DADDU  R10, R4, R3
```

a factor of 1.5 improvement.

<table>
<thead>
<tr>
<th>Integer instruction</th>
<th>FP instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D     F0, 0(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D     F6, -8(R1)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L.D     F10, -16(R1)</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>L.D     F14, -24(R1)</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>L.D     F18, -32(R1)</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>S.D     F4, 0(R1)</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>S.D     F8, -8(R1)</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>S.D     F12, -16(R1)</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>DADDUI  R1, R1, #40</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>S.D     F16, 16(R1)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>BNE     R1, R2, Loop</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>S.D     F20, 8(R1)</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>
The dependence of the `DSUBU` and `BEQZ` on the `LD` instruction means that a stall will be needed after the `LD`. Suppose we knew that this branch was almost always taken and that the value of `R7` was not needed on the fall-through path. Then we could increase the speed of the program by moving the instruction `DADD R7, R8, R9` to the position after the `LD`. Correspondingly, if we knew the branch was rarely taken and that the value of `R4` was not needed on the taken path, then we could contemplate moving the `OR` instruction after the `LD`. In addition, we can also use the information to better schedule any branch delay, since choosing how to schedule the delay depends on knowing the branch behavior. We will return to this topic in section 4.4, when we discuss global code scheduling.

To perform these optimizations, we need to predict the branch statically when we compile the program. There are several different methods to statically predict branch behavior. The simplest scheme is to predict a branch as taken. This scheme has an average misprediction rate that is equal to the untaken branch frequency, which for the SPEC programs is 34%. Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%).

A better alternative is to predict on the basis of branch direction, choosing backward-going branches to be taken and forward-going branches to be not taken. For some programs and compilation systems, the frequency of forward taken branches may be significantly less than 50%, and this scheme will do better than just predicting all branches as taken. In the SPEC programs, however, more than half of the forward-going branches are taken. Hence, predicting all branches as taken is the better approach. Even for other benchmarks or compilers, direction-based prediction is unlikely to generate an overall misprediction rate of less than 30% to 40%. An enhancement of this technique was explored by Ball and Larus; their approach uses program context information and generates more accurate predictions than a simple scheme based solely on branch direction.

A still more accurate technique is to predict branches on the basis of profile information collected from earlier runs. The key observation that makes this worthwhile is that the behavior of branches is often bimodally distributed; that is, an individual branch is often highly biased toward taken or untaken. Figure 4.3 shows the success of branch prediction using this strategy. The same input data were used for runs and for collecting the profile; other studies have shown that
changing the input so that the profile is for a different run leads to only a small change in the accuracy of profile-based prediction.

![Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which varies from 3% to 24%; we will examine the combined effect in Figure 4.4.](image)

Although we can derive the prediction accuracy of a predict-taken strategy and measure the accuracy of the profile scheme, as in Figure 4.3, the wide range of frequency of conditional branches in these programs, from 3% to 24%, means that the overall frequency of a mispredicted branch varies widely. Figure 4.4 shows the number of instructions executed between mispredicted branches for both a profile-based and a predict-taken strategy. The number varies widely, both because of the variation in accuracy and the variation in branch frequency. On average, the predict-taken strategy has 20 instructions per mispredicted branch and the profile-based strategy has 110. These averages, however, are very different for integer and FP programs, as the data in Figure 4.4 show.

Static branch behavior is useful for scheduling instructions when the branch delays are exposed by the architecture (either delayed or canceling branches), for assisting dynamic predictors (as we will see in the IA-64 architecture in section 4.7), and for determining which code paths are more frequent, which is a key step in code scheduling (see section 4.4, page 251).
Superscalar processors decide on the fly how many instructions to issue. A statically scheduled superscalar must check for any dependences between instructions in the issue packet as well as between any issue candidate and any instruction already in the pipeline. As we have seen in Section 4.1, a statically-scheduled superscalar requires significant compiler assistance to achieve good performance. In contrast, a dynamically-scheduled superscalar requires less compiler assistance, but has significant hardware costs.

An alternative to the superscalar approach is to rely on compiler technology not only to minimize the potential data hazard stalls, but to actually format the instructions in a potential issue packet so that the hardware need not check explicitly for dependences. The compiler may be required to ensure that dependences within the issue packet cannot be present or, at a minimum, indicate when a dependence may be present. Such an approach offers the potential advantage of simpler hardware while still exhibiting good performance through extensive compiler optimization.

4.3 Static Multiple Issue: the VLIW Approach

Superscalar processors decide on the fly how many instructions to issue. A statically scheduled superscalar must check for any dependences between instructions in the issue packet as well as between any issue candidate and any instruction already in the pipeline. As we have seen in Section 4.1, a statically-scheduled superscalar requires significant compiler assistance to achieve good performance. In contrast, a dynamically-scheduled superscalar requires less compiler assistance, but has significant hardware costs.

An alternative to the superscalar approach is to rely on compiler technology not only to minimize the potential data hazard stalls, but to actually format the instructions in a potential issue packet so that the hardware need not check explicitly for dependences. The compiler may be required to ensure that dependences within the issue packet cannot be present or, at a minimum, indicate when a dependence may be present. Such an approach offers the potential advantage of simpler hardware while still exhibiting good performance through extensive compiler optimization.
4.3 Static Multiple Issue: the VLIW Approach

The first multiple-issue processors that required the instruction stream to be explicitly organized to avoid dependences used wide instructions with multiple operations per instruction. For this reason, this architectural approach was named VLIW, standing for Very Long Instruction Word, and denoting that the instructions, since they contained several instructions, were very wide (64 to 128 bits, or more). The basic architectural concepts and compiler technology are the same whether multiple operations are organized into a single instruction, or whether a set of instructions in an issue packet is preconfigured by a compiler to exclude dependent operations (since the issue packet can be thought of as a very large instruction). Early VLIWs were quite rigid in their instruction formats and effectively required recompilation of programs for different versions of the hardware.

To reduce this inflexibility and enhance performance of the approach, several innovations have been incorporated into more recent architectures of this type, while still requiring the compiler to do most of the work of finding and scheduling instructions for parallel execution. This second generation of VLIW architectures is the approach being pursued for desktop and server markets.

In the remainder of this section, we look at the basic concepts in a VLIW architecture. Section 4.4 introduces additional compiler techniques that are required to achieve good performance for compiler-intensive approaches, and Section 4.5 describes hardware innovations that improve flexibility and performance of explicitly parallel approaches. Finally, Section 4.7 describes how the Intel IA-64 supports explicit parallelism.

The Basic VLIW Approach

VLIWs use multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction, or requires that the instructions in the issue packet satisfy the same constraints. Since there is not fundamental difference in the two approaches, we will just assume that multiple operations are placed in one instruction, as in the original VLIW approach. Since the burden for choosing the instructions to be issued simultaneously falls on the compiler, the hardware in a superscalar to make these issue decisions is unneeded.

Since this advantage of a VLIW increases as the maximum issue rate grows, we focus on a wider-issue processor. Indeed, for simple two issue processors, the overhead of a superscalar is probably minimal. Many designers would probably argue that a four issue processor has manageable overhead, but as we saw in the last chapter, this overhead grows with issue width.

Because VLIW approaches make sense for wider processors, we choose to focus our example on such an architecture. For example, a VLIW processor might have instructions that contain five operations, including: one integer operation (which could also be a branch), two floating-point operations, and two memory references. The instruction would have a set of fields for each functional unit—perhaps 16 to 24 bits per unit, yielding an instruction length of between 112 and 168 bits.
To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. This parallelism is uncovered by unrolling loops and scheduling the code within the single larger loop body. If the unrolling generates straighline code, then local scheduling techniques, which operate on a single basic block can be used. If finding and exploiting the parallelism requires scheduling code across branches, a substantially more complex global scheduling algorithm must be used. Global scheduling algorithms are not only more complex in structure, but they must deal with significantly more complicated tradeoffs in optimization, since moving code across branches is expensive. In the next section, we will discuss trace scheduling, one of these global scheduling techniques developed specifically for VLIWs. In Section 4.5, we will examine hardware support that allows some conditional branches to be eliminated, extending the usefulness of local scheduling and enhancing the performance of global scheduling.

For now, let’s assume we have a technique to generate long, straight-line code sequences, so that we can use local scheduling to build up VLIW instructions and instead focus on how well these processors operate.

**Example**

Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop \( x[i] = x[i] + s \) (see page 223 for the MIPS code) for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore the branch-delay slot.

**Answer**

The code is shown in Figure 4.5. The loop has been unrolled to make seven copies of the body, which eliminates all stalls (i.e., completely empty issue cycles), and runs in 9 cycles. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result, nearly twice as fast as the two-issue superscalar of Section 4.1 that used unrolled and scheduled code.

For the original VLIW model, there are both technical and logistical problems. The technical problems are the increase in code size and the limitations of lock-step operation. Two different elements combine to increase code size substantially for a VLIW. First, generating enough operations in a straight-line code fragment requires ambitiously unrolling loops (as earlier examples) thereby increasing code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding. In Figure 4.5, we saw that only about 60% of the functional units were used, so almost half of each instruction was empty. In most VLIWs, an instruction may need to be left completely empty if no operations can be scheduled.

To combat this code size increase, clever encodings are sometimes used. For example, there may be only one large immediate field for use by any functional unit. Another technique is to compress the instructions in main memory and ex-
Early VLIWs operated in lock-step; there was no hazard detection hardware at all. This structure dictated that a stall in any functional unit pipeline must cause the entire processor to stall, since all the functional units must be kept synchronized. Although a compiler may be able to schedule the deterministic functional units to prevent stalls, predicting which data accesses will encounter a cache stall and scheduling them is very difficult. Hence, caches needed to be blocking and to cause all the functional units to stall. As the issue rate and number of memory references becomes large, this synchronization restriction becomes unacceptable. In more recent processors, the functional units operate more independently, and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronized execution once instructions are issued.

Binary code compatibility has also been a major logistical problem for VLIWs. In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies. Thus, different numbers of functional units and unit latencies require different versions of the code. This requirement makes migrating between successive implementations, or between implementations with different issue widths, more difficult than it is for a superscalar design. Of course, obtaining improved performance from a new superscalar design may require recompilation. Nonetheless, the ability to run old binary files is a practical advantage for the superscalar approach.

One possible solution to this migration problem, and the problem of binary code compatibility in general, is object-code translation or emulation. This technology is developing quickly and could play a significant role in future migration.

---

**FIGURE 4.5** VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes nine cycles assuming no branch delay; normally the branch delay would also need to be scheduled. The issue rate is 23 operations in nine clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a larger number of registers than MIPS would normally use in this loop. The VLIW code sequence above requires at least eight FP registers, while the same code sequence for the base MIPS processor can use as few as two FP registers or as many as five when unrolled and scheduled. In the superscalar example in Figure 4.2, six registers were needed.
schemes. Another approach is to temper the strictness of the approach so that bi-
nary compatibility is still feasible. This later approach is used in the IA-64 archi-
tecture, as we will see in Section 4.7.

The major challenge for all multiple-issue processors is to try to exploit large
amounts of ILP. When the parallelism comes from unrolling simple loops in FP
programs, the original loop probably could have been run efficiently on a vector
processor (described in Appendix B). It is not clear that a multiple-issue proce-
sor is preferred over a vector processor for such applications; the costs are simi-
lar, and the vector processor is typically the same speed or faster. The potential
advantages of a multiple-issue processor versus a vector processor are twofold.
First, a multiple-issue processor has the potential to extract some amount of par-
allelism from less regularly structured code, and, second, it has the ability to use
a more conventional, and typically less expensive, cache-based memory system.
For these reasons multiple-issue approaches have become the primary method for
taking advantage of instruction-level parallelism, and vectors have become pri-
marily an extension to these processors.

4.4 Advanced Compiler Support for Exposing
and Exploiting ILP

In this section we discuss compiler technology for increasing the amount of par-
allelism that we can exploit in a program. We begin by defining when a loop is
parallel and how a dependence can prevent a loop from being parallel. We also
discuss techniques for eliminating some types of dependences. As we will see in
later sections, hardware support for these compiler techniques can greatly in-
crease their effectiveness. This section serves as an introduction to these tech-
niques. We do not attempt to explain the details of ILP-oriented compiler
techniques, since this would take hundreds of pages, rather than the 20 we have
allotted. Instead, we view this material as providing general background that will
enable the reader to have a basic understanding of the compiler techniques used
to exploit ILP in modern computers.

Detecting and Enhancing Loop-Level Parallelism

Loop-level parallelism is normally analyzed at the source level or close to it,
while most analysis of ILP is done once instructions have been generated by the
compiler. Loop-level analysis involves determining what dependences exist
among the operands in a loop across the iterations of that loop. For now, we will
consider only data dependences, which arise when an operand is written at some
point and read at a later point. Name dependences also exist and may be removed
by renaming techniques like those we used earlier.

The analysis of loop-level parallelism focuses on determining whether data
accesses in later iterations are dependent on data values produced in earlier iter-
ations, such a dependence is called a \textit{loop-carried dependence}. Most of the exam-
Examples we considered in Section 4.1 have no loop-carried dependences and, thus, are loop-level parallel. To see that a loop is parallel, let us first look at the source representation:

\[
\text{for } (i=1000; i>0; i=i-1) \\
    x[i] = x[i] + s;
\]

In this loop, there is a dependence between the two uses of \(x[i]\), but this dependence is within a single iteration and is not loop-carried. There is a dependence between successive uses of \(i\) in different iterations, which is loop-carried, but this dependence involves an induction variable and can be easily recognized and eliminated. We saw examples of how to eliminate dependences involving induction variables during loop unrolling in Section 4.1, and we will look at additional examples later in this section.

Because finding loop-level parallelism involves recognizing structures such as loops, array references, and induction variable computations, the compiler can do this analysis more easily at or near the source level, as opposed to the machine-code level. Let’s look at a more complex example.

**Example** Consider a loop like this one:

\[
\text{for } (i=1; i\leq100; i=i+1) \\
\quad \{ \\
\quad \quad A[i+1] = A[i] + C[i]; \quad /* \text{S1} */ \\
\quad \quad B[i+1] = B[i] + A[i+1]; \quad /* \text{S2} */ \\
\quad \}
\]

Assume that \(A\), \(B\), and \(C\) are distinct, nonoverlapping arrays. (In practice, the arrays may sometimes be the same or may overlap. Because the arrays may be passed as parameters to a procedure, which includes this loop, determining whether arrays overlap or are identical often requires sophisticated, interprocedural analysis of the program.) What are the data dependences among the statements \(\text{S1}\) and \(\text{S2}\) in the loop?

**Answer** There are two different dependences:

1. \(\text{S1}\) uses a value computed by \(\text{S1}\) in an earlier iteration, since iteration \(i\) computes \(A[i+1]\), which is read in iteration \(i+1\). The same is true of \(\text{S2}\) for \(B[i]\) and \(B[i+1]\).
2. \(\text{S2}\) uses the value, \(A[i+1]\), computed by \(\text{S1}\) in the same iteration.

These two dependences are different and have different effects. To see how they differ, let’s assume that only one of these dependences exists at a time. Because the dependence of statement \(\text{S1}\) on an earlier iteration of \(\text{S1}\), this dependence is loop-carried. This dependence forces successive iterations of this loop to execute in series.

The second dependence above (\(\text{S2}\) depending on \(\text{S1}\)) is within an it-
operation and is not loop-carried. Thus, if this were the only dependence, multiple iterations of the loop could execute in parallel, as long as each pair of statements in an iteration were kept in order. We saw this type of dependence in an example in Section 4.1, where unrolling was able to expose the parallelism.

It is also possible to have a loop-carried dependence that does not prevent parallelism, as the next example shows.

**Example** Consider a loop like this one:

```c
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];   /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
}
```

What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.

**Answer** Statement S1 uses the value assigned in the previous iteration by statement S2, so there is a loop-carried dependence between S2 and S1. Despite this loop-carried dependence, this loop can be made parallel. Unlike the earlier loop, this dependence is not circular: Neither statement depends on itself, and although S1 depends on S2, S2 does not depend on S1. A loop is parallel if it can be written without a cycle in the dependences, since the absence of a cycle means that the dependences give a partial ordering on the statements.

Although there are no circular dependences in the above loop, it must be transformed to conform to the partial ordering and expose the parallelism. Two observations are critical to this transformation:

1. There is no dependence from S1 to S2. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Since this other dependence is absent, interchanging the two statements will not affect the execution of S2.

2. On the first iteration of the loop, statement S1 depends on the value of B[1] computed prior to initiating the loop.

These two observations allow us to replace the loop above with the following code sequence:

```c
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
```
4.4 Advanced Compiler Support for Exposing and Exploiting ILP

\[
B[101] = C[100] + D[100];
\]

The dependence between the two statements is no longer loop-carried, so that iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.

Our analysis needs to begin by finding all loop-carried dependences. This dependence information is **inexact**, in the sense that it tells us that such a dependence **may** exist. Consider the following example:

```c
for (i=1; i<=100; i=i+1) {
    A[i] = B[i] + C[i]
    D[i] = A[i] * E[i]
}
```

The second reference to \(A\) in this example need not be translated to a load instruction, since we know that the value is computed and stored by the previous statement; hence, the second reference to \(A\) can simply be a reference to the register into which \(A\) was computed. Performing this optimization requires knowing that the two references are **always** to the same memory address and that there is no intervening access to the same location. Normally, data dependence analysis only tells that one reference **may** depend on another; a more complex analysis is required to determine that two references **must be** to the exact same address. In the example above, a simple version of this analysis suffices, since the two references are in the same basic block.

Often loop-carried dependences are in the form of a **recurrence**:

```c
for (i=2; i<=100; i=i+1) {
    Y[i] = Y[i-1] + Y[i];
}
```

A recurrence is when a variable is defined based on the value of that variable in an earlier iteration, often the one immediately preceding, as in the above fragment. Detecting a recurrence can be important for two reasons: Some architectures (especially vector computers) have special support for executing recurrences, and some recurrences can be the source of a reasonable amount of parallelism. To see how the latter can be true, consider this loop:

```c
for (i=6; i<=100; i=i+1) {
    Y[i] = Y[i-5] + Y[i];
}
```
On the iteration $i$, the loop references element $i - 5$. The loop is said to have a *dependence distance* of 5. Many loops with carried dependences have a dependence distance of 1. The larger the distance, the more potential parallelism can be obtained by unrolling the loop. For example, if we unroll the first loop, with a dependence distance of 1, successive statements are dependent on one another; there is still some parallelism among the individual instructions, but not much. If we unroll the loop that has a dependence distance of 5, there is a sequence of five statements that have no dependences, and thus much more ILP. Although many loops with loop-carried dependences have a dependence distance of 1, cases with larger distances do arise, and the longer distance may well provide enough parallelism to keep a processor busy.

**Finding Dependences**

Finding the dependences in a program is an important part of three tasks: (1) good scheduling of code, (2) determining which loops might contain parallelism, and (3) eliminating name dependences. The complexity of dependence analysis arises because of the presence of arrays and pointers in languages like C or C++ or pass-by-reference parameter passing in Fortran. Since scalar variable references explicitly refer to a name, they can usually be analyzed quite easily, with aliasing because of pointers and reference parameters causing some complications and uncertainty in the analysis.

How does the compiler detect dependences in general? Nearly all dependence analysis algorithms work on the assumption that array indices are affine. In simplest terms, a one-dimensional array index is affine if it can be written in the form $a \times i + b$, where $a$ and $b$ are constants, and $i$ is the loop index variable. The index of a multidimensional array is affine if the index in each dimension is affine. Sparse array accesses, which typically have the form $x[y[i]]$, are one of the major examples of nonaffine accesses.

Determining whether there is a dependence between two references to the same array in a loop is thus equivalent to determining whether two affine functions can have the same value for different indices between the bounds of the loop. For example, suppose we have stored to an array element with index value $a \times i + b$ and loaded from the same array with index value $c \times i + d$, where $i$ is the for-loop index variable that runs from $m$ to $n$. A dependence exists if two conditions hold:

1. There are two iteration indices, $j$ and $k$, both within the limits of the for loop. That is $m \leq j \leq n$, $m \leq k \leq n$.

2. The loop stores into an array element indexed by $a \times j + b$ and later fetches from that *same* array element when it is indexed by $c \times k + d$. That is, $a \times j + b = c \times k + d$. 
In general, we cannot determine whether a dependence exists at compile time. For example, the values of \( a, b, c, \) and \( d \) may not be known (they could be values in other arrays), making it impossible to tell if a dependence exists. In other cases, the dependence testing may be very expensive but decidable at compile time. For example, the accesses may depend on the iteration indices of multiple nested loops. Many programs, however, contain primarily simple indices where \( a, b, c, \) and \( d \) are all constants. For these cases, it is possible to devise reasonable compile-time tests for dependence.

As an example, a simple and sufficient test for the absence of a dependence is the *greatest common divisor*, or GCD, test. It is based on the observation that if a loop-carried dependence exists, then GCD \((c,a)\) must divide \((d - b)\). (Recall that an integer, \( x \), divides another integer, \( y \), if there is no remainder when we do the division \( y/x \) and get an integer quotient.)

### Example

Use the GCD test to determine whether dependences exist in the following loop:

```c
for (i=1; i<=100; i=i+1) {
}
```

**Answer**

Given the values \( a = 2, b = 3, c = 2, \) and \( d = 0 \), then GCD\((a,c)\) = 2, and \( d - b = -3 \). Since 2 does not divide \(-3\), no dependence is possible.

The GCD test is sufficient to guarantee that no dependence exists (you can show this in the Exercises); however, there are cases where the GCD test succeeds but no dependence exists. This can arise, for example, because the GCD test does not take the loop bounds into account.

In general, determining whether a dependence actually exists is NP-complete. In practice, however, many common cases can be analyzed precisely at low cost. Recently, approaches using a hierarchy of exact tests increasing in generality and cost have been shown to be both accurate and efficient. (A test is *exact* if it precisely determines whether a dependence exists. Although the general case is NP-complete, there exist exact tests for restricted situations that are much cheaper.)

In addition to detecting the presence of a dependence, a compiler wants to classify the type of dependence. This classification allows a compiler to recognize name dependences and eliminate them at compile time by renaming and copying.

### Example

The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate
the output dependences and antidependences by renaming.

```c
for (i=1; i<=100; i=i+1) {
    Y[i] = X[i] / c; /*S1*/
    X[i] = X[i] + c; /*S2*/
    Z[i] = Y[i] + c; /*S3*/
    Y[i] = c - Y[i]; /*S4*/
}
```

**Answer** The following dependences exist among the four statements:

1. There are true dependences from S1 to S3 and from S1 to S4 because of \(Y[i]\). These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force S3 and S4 to wait for S1 to complete.

2. There is an antidependence from S1 to S2, based on \(X[i]\).

3. There is an antidependence from S3 to S4 for \(Y[i]\).

4. There is an output dependence from S1 to S4, based on \(Y[i]\).

The following version of the loop eliminates these false (or pseudo) dependences.

```c
for (i=1; i<=100; i=i+1) {
    /* Y renamed to T to remove output dependence*/
    T[i] = X[i] / c;
    /* X renamed to X1 to remove antidependence*/
    X1[i] = X[i] + c;
    /* Y renamed to T to remove antidependence */
    Z[i] = T[i] + c;
    Y[i] = c - T[i];
}
```

After the loop the variable \(X\) has been renamed \(X1\). In code that follows the loop, the compiler can simply replace the name \(X\) by \(X1\). In this case, renaming does not require an actual copy operation but can be done by substituting names or by register allocation. In other cases, however, renaming will require copying.

Dependence analysis is a critical technology for exploiting parallelism. At the instruction level it provides information needed to interchange memory references when scheduling, as well as to determine the benefits of unrolling a loop. For detecting loop-level parallelism, dependence analysis is the basic tool. Effectively compiling programs to either vector computers or multiprocessors depends criti-
cally on this analysis. The major drawback of dependence analysis is that it applies only under a limited set of circumstances, namely among references within a single loop nest and using affine index functions. Thus, there are a wide variety of situations in which array-oriented dependence analysis cannot tell us what we might want to know, including

- when objects are referenced via pointers rather than array indices (but see discussion below);
- when array indexing is indirect through another array, which happens with many representations of sparse arrays;
- when a dependence may exist for some value of the inputs, but does not exist in actuality when the code is run since the inputs never take on those values;
- when an optimization depends on knowing more than just the possibility of a dependence, but needs to know on which write of a variable does a read of that variable depend.

To deal with the issue of analyzing programs with pointers, another type of analysis, often called points-to analysis, is required (see Wilson and Lam [1995]). The key question that we want answered from dependence analysis of pointers is whether two pointers can designate the same address. In the case of complex dynamic data structures, this problem is extremely difficult. For example, we may want to know whether two pointers can reference the same node in a list at a given point in a program, which in general is undecidable and in practice is extremely difficult to answer. We may, however, be able to answer a simpler question: can two pointers designate nodes in the same list, even if they may be separate nodes. This more restricted analysis can still be quite useful in scheduling memory accesses performed through pointers.

The basic approach used in points-to analysis relies on information from three major sources:

1. Type information, which restricts what a pointer can point to.
2. Information derived when an object is allocated or when the address of an object is taken, which can be used to restrict what a pointer can point to. For example, if p always points to an object allocated in a given source line and q never points to that object, then p and q can never point to the same object.
3. Information derived from pointer assignments. For example, if p may be assigned the value of q, then p may point to anything q points to.

There are several cases where analyzing pointers has been successfully applied and is extremely useful:

- When pointers are used to pass the address of an object as a parameter, it is pos-
sible to use points-to analysis to determine the possible set of objects referenced by a pointer. One important use is to determine if two pointer parameters may designate the same object.

- When a pointer can point to one of several types, it is sometimes possible to determine that the type of the data object a pointer designates at different parts of the program.

- It is often possible to separate out pointers that may only point to a local object versus a global one.

There are two different types of limitations that affect our ability to do accurate dependence analysis for large programs. The first type of limitation arises from restrictions in the analysis algorithms. Often, we are limited by the lack of applicability of the analysis rather than a shortcoming in dependence analysis per se. For example, dependence analysis for pointers is essentially impossible for programs that use pointers in arbitrary fashion—for example, by doing arithmetic on pointers.

The second limitation is the need to analyze behavior across procedure boundaries to get accurate information. For example, if a procedure accepts two parameters that are pointers, determining whether the values could be the same requires analyzing across procedure boundaries. This type of analysis, called *interprocedural analysis*, is much more difficult and complex than analysis within a single procedure. Unlike the case of analyzing array indices within a single loop nest, points-to analysis usually requires an interprocedural analysis. The reason for this is simple. Suppose we are analyzing a program segment with two pointers; if the analysis does not know anything about the two pointers at the start of the program segment, it must be conservative and assume the worst case. The worst case is that the two pointers may designate the same object, but they are not guaranteed to designate the same object. This worst case is likely to propagate through the analysis producing useless information. In practice, getting fully accurate interprocedural information is usually too expensive for real programs. Instead, compilers usually use approximations in interprocedural analysis. The result is that the information may be too inaccurate to be useful.

Modern programming languages that use strong typing, such as Java, make the analysis of dependences easier. At the same time the extensive use of procedures to structure programs, as well as abstract data types, makes the analysis more difficult. Nonetheless, we expect that continued advances in analysis algorithms combined with the increasing importance of pointer dependency analysis will mean that there is continued progress on this important problem.

Eliminating Dependent Computations

Compilers can reduce the impact of dependent computations so as to achieve more ILP. The key technique is to eliminate or reduce a dependent computation
by back substitution, which increases the amount of parallelism and sometimes increases the amount of computation required. These techniques can be applied both within a basic block and within loops, and we describe them differently.

Within a basic block, algebraic simplifications of expressions and an optimization called *copy propagation*, which eliminates operations that copy values, can be used to simplify sequences like the following:

```
DADDUI R1, R2, #4
DADDUI R1, R1, #4
```
to:
```
DADDUI R1, R2, #8
```
assuming this is the only use of \( R1 \). In fact, the techniques we used to reduce multiple increments of array indices during loop unrolling and to move the increments across memory addresses in Section 4.1 are examples of this type of optimization.

In these examples, computations are actually eliminated, but it also possible that we may want to increase the parallelism of the code, possibly even increasing the number of operations. Such optimizations are called *tree height reduction*, since they reduce the height of the tree structure representing a computation, making it wider but shorter. Consider the following code sequence:

```
ADD R1, R2, R3
ADD R4, R1, R6
ADD R8, R4, R7
```
Notice that this sequence requires at least three execution cycles, since all the instructions depend on the immediate predecessor. By taking advantage of associativity, we can transform the code and rewrite it as:

```
ADD R1, R2, R3
ADD R4, R6, R7
ADD R8, R1, R4
```
This sequence can be computed in two execution cycles. When loop unrolling is used, opportunities for these types of optimizations occur frequently.

Although arithmetic with unlimited range and precision is associative, computer arithmetic is not associative, either for integer arithmetic, because of limited range, or floating point arithmetic, because of both range and precision. Thus, using these restructuring techniques can sometimes lead to erroneous behavior, although such occurrences are rare. For this reason, most compilers require that optimizations that rely on associativity be explicitly enabled.

When loops are unrolled this sort of algebraic optimization is important to reduce the impact of dependences arising from recurrences. *Recurrences* are expressions whose value on one iteration is given by a function that depends on the previous iterations. When a loop with a recurrence is unrolled, we may be able to algebraically optimize the unrolled loop, so that the recurrence need only be eval-
ated once per unrolled iteration. One common type of recurrence arises from an explicit program statements, such as:

\[ \text{sum} = \text{sum} + x; \]

Assume we unroll a loop with this recurrence five times, if we let the value of \( x \) on these five iterations be given by \( x_1, x_2, x_3, x_4, \) and \( x_5 \), then we can write the value of \( \text{sum} \) at the end of each unroll as:

\[ \text{sum} = \text{sum} + x_1 + x_2 + x_3 + x_4 + x_5; \]

If unoptimized this expression requires five dependent operations, but it can be rewritten as:

\[ \text{sum} = ((\text{sum} + x_1) + (x_2 + x_3)) + (x_4 + x_5); \]

which can be evaluated in only three dependent operations.

Recurrences also arise from implicit calculations, such as those associated with array indexing. Each array index translates to an address that is computed based on the loop index variable. Again, with unrolling and algebraic optimization, the dependent computations can be minimized.

**Software Pipelining: Symbolic Loop Unrolling**

We have already seen that one compiler technique, loop unrolling, is useful to uncover parallelism among instructions by creating longer sequences of straight-line code. There are two other important techniques that have been developed for this purpose: **software pipelining** and **trace scheduling**.

**Software pipelining** is a technique for reorganizing loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop. This approach is most easily understood by looking at the scheduled code for the superscalar version of MIPS, which appeared in Figure 4.2 on page 231. The scheduler in this example essentially interleaves instructions from different loop iterations, so as to separate the dependent instructions that occur within a single loop iteration. By choosing instructions from different iterations, dependent computations are separated from one another by an entire loop body, increasing the possibility that the unrolled loop can be scheduled without stalls.

A software-pipelined loop interleaves instructions from different iterations without unrolling the loop, as illustrated in Figure 4.6. This technique is the software counterpart to what Tomasulo’s algorithm does in hardware. The software-pipelined loop for the earlier example would contain one load, one add, and one store, each from a different iteration. There is also some start-up code that is needed before the loop begins as well as code to finish up after the loop is completed. We will ignore these in this discussion, for simplicity; the topic is addressed in the Exercises.
4.4 Advanced Compiler Support for Exposing and Exploiting ILP

**EXAMPLE**

Show a software-pipelined version of this loop, which increments all the elements of an array whose starting address is in R1 by the contents of F2:

```
Loop:  L.D   F0,0(R1)
       ADD.D F4,F0,F2
       S.D   F4,0(R1)
       DADDUI R1,R1,#-8
       BNE   R1,R2,Loop
```

You may omit the start-up and clean-up code.

**ANSWER**

Software pipelining symbolically unrolls the loop and then selects instructions from each iteration. Since the unrolling is symbolic, the loop overhead instructions (the DADDUI and BNE) need not be replicated. Here’s the body of the unrolled loop without overhead instructions, highlighting the instructions taken from each iteration:
Iteration i:  
L.D F0,0(R1)  
ADD.D F4,F0,F2  
S.D F4,0(R1)

Iteration i+1:  
L.D F0,0(R1)  
ADD.D F4,F0,F2  
S.D F4,0(R1),F4

Iteration i+2:  
L.D F0,0(R1)  
ADD.D F4,F0,F2  
S.D F4,0(R1)

The selected instructions from different iterations are then put together in the loop with the loop control instructions:

Loop:  
S.D F4,16(R1); stores into M[i]  
ADD.D F4,F0,F2; adds to M[i-1]  
L.D F0,0(R1); loads M[i-2]  
DADDUI R1,R1,#-8  
BNE R1,R2,Loop

This loop can be run at a rate of 5 cycles per result, ignoring the start-up and clean-up portions, and assuming that DADDUI is scheduled after the ADD.D and the L.D instruction, with an adjusted offset, is placed in the branch delay slot. Because the load and store are separated by offsets of 16 (two iterations), the loop should run for two fewer iterations. (We address this and the start-up and clean-up portions in Exercise 4.18.) Notice that the reuse of registers (e.g., F4, F0, and R1) requires the hardware to avoid the WAR hazards that would occur in the loop. This hazard should not be a problem in this case, since no data-dependent stalls should occur.

By looking at the unrolled version we can see what the start-up code and finish code will need to be. For start-up, we will need to execute any instructions that correspond to iteration 1 and 2 that will not be executed. These instructions are the L.D for iterations 1 and 2 and the ADD.D for iteration 1. For the finish code, we need to execute any instructions that will not be executed in the final two iterations. These include the ADD.D for the last iteration and the S.D for the last two iterations.

Register management in software-pipelined loops can be tricky. The example above is not too hard since the registers that are written on one loop iteration are read on the next. In other cases, we may need to increase the number of iterations between when we issue an instruction and when the result is used. This increase is required when there are a small number of instructions in the loop body and the latencies are large. In such cases, a combination of software pipelining and loop unrolling is needed. An example of this is shown in the Exercises.
Software pipelining can be thought of as *symbolic* loop unrolling. Indeed, some of the algorithms for software pipelining use loop-unrolling algorithms to figure out how to software pipeline the loop. The major advantage of software pipelining over straight loop unrolling is that software pipelining consumes less code space. Software pipelining and loop unrolling, in addition to yielding a better scheduled inner loop, each reduce a different type of overhead. Loop unrolling reduces the overhead of the loop—the branch and counter-update code. Software pipelining reduces the time when the loop is not running at peak speed to once per loop at the beginning and end. If we unroll a loop that does 100 iterations a constant number of times, say 4, we pay the overhead $100/4 = 25$ times—every time the inner unrolled loop is initiated. Figure 4.7 shows this behavior graphically. Because these techniques attack two different types of overhead, the best performance can come from doing both. In practice, compilation using software pipelining is quite difficult for several reasons: many loops require significant transformation before they can be software pipelined, the tradeoffs in terms of overhead versus efficiency of the software-Pipelined loop are complex, and the issue of register management creates additional complexities. To help deal with the last two of these issues, the IA-64 added extensive hardware support for software pipelining. Although this hardware can make it more efficient to apply software pipelining, it does not eliminate the need for complex compiler support, or for the need to make difficult decisions about the best way to compile a loop.

**Global Code Scheduling**

In section 4.1 we examined the use of loop unrolling and code scheduling to improve ILP. The techniques in section 4.1 work well when the loop body is straightline code, since the resulting unrolled loop looks like a single basic block. Similarly, software pipelining works well when the body is single basic block, since it is easier to find the repeatable schedule. When the body of an unrolled loop contains internal control flow, however, scheduling the code is much more complex. In general, effective scheduling of a loop body with internal control flow will require moving instructions across branches, which is global code scheduling. In this section, we first examine the challenge and limitations of global code scheduling. In section 4.5 we examine hardware support for eliminating control flow within an inner loop; then, we examine two compiler techniques that can be used when eliminating the control flow is not a viable approach.

Global code scheduling aims to compact a code fragment with internal control structure into the shortest possible sequence that preserves the data and control dependences. The data dependences force a partial order on operations, while the control dependences dictate instructions across which code cannot be easily moved. Data dependences are overcome by unrolling and, in the case of memory operations, using dependence analysis to determine if two references refer to the same address. Finding the shortest possible sequence for a piece of code means
finding the shortest sequence for the *critical path*, which is the longest sequence of dependent instructions.

Control dependences arising from loop branches are reduced by unrolling. Global code scheduling can reduce the effect of control dependences arising from conditional nonloop branches by moving code. Since moving code across branches will often affect the frequency of execution of such code, effectively using global code motion requires estimates of the relative frequency of different paths. Although global code motion cannot guarantee faster code, if the frequency information is accurate, the compiler can determine whether such code movement is likely to lead to faster code.
Global code motion is important since many inner loops contain conditional statements. Figure 4.8 shows a typical code fragment, which may be thought of as an iteration of an unrolled loop and highlights the more common control flow.

Effectively scheduling this code could require that we move the assignments to B and C to earlier in the execution sequence, before the test of A. Such global code motion must satisfy a set of constraints to be legal. In addition, the movement of the code associated with B, unlike that associated with C, is speculative: it will speed the computation up only when the path containing the code would be taken.

To perform the movement of B, we must ensure that neither the data flow nor the exception behavior is changed. Compilers avoid changing the exception behavior by not moving certain classes of instructions, such as memory references, that can cause exceptions. In section 4.5, we will see how hardware support allow for more opportunities for speculative code motion as well as remove control dependences. Although such enhanced support for speculation can make it possible to explore more opportunities, the difficulty of choosing how to best compile the code remains complex.

How can the compiler ensure that the assignments to B and C can be moved without affecting the data flow? To see what’s involved, let’s look at a typical code generation sequence for the flowchart in Figure 4.8. Assuming that the addresses for A, B, C are in R1, R2, and R3, respectively, here is such a sequence:
Let's first consider the problem of moving the assignment to B to before the BNEZ instruction. Call the last instruction to assign to B before the if statement, i. If B is referenced before it is assigned either in code segment X or after the if-statement, call the referencing instruction j. If there is such an instruction j, then moving the assignment to B will change the data flow of the program. In particular, moving the assignment to B will cause j to become data-dependent on the moved version of the assignment to B rather than on i on which j originally depended. One could imagine more clever schemes to allow B to be moved even when the value is used: for example, in the first case, we could make a shadow copy of B before the if statement and use that shadow copy in X. Such schemes are usually avoided, both because they are complex to implement and because they will slow down the program if the trace selected is not optimal and the operations end up requiring additional instructions to execute.

Moving the assignment to C up to before the first branch requires two steps. First, the assignment is moved over the join point of the else part into the portion corresponding to the then part. This movement makes the instructions for C control-dependent on the branch and means that they will not execute if the else path, which is the infrequent path, is chosen. Hence, instructions that were data-dependent on the assignment to C, and which execute after this code fragment, will be affected. To ensure the correct value is computed for such instructions, a copy is made of the instructions that compute and assign to C on the else path. Second, we can move C from the then part of the branch across the branch condition, if it does not affect any data flow into the branch condition. If C is moved to before the if-test, the copy of C in the else branch can usually be eliminated, since it will be redundant.

We can see from this example that global code scheduling is subject to many constraints. This observation is what led designers to provide hardware support to make such code motion easier, and section 4.5 explores such support in detail.
Global code scheduling also requires complex tradeoffs to make code motion decisions. For example, assuming that the assignment to B can be moved before the conditional branch (possibly with some compensation code on the alternative branch) will this movement make the code run faster? The answer is: possibly! Similarly, moving the copies of C into the if and else branches, makes the code initially bigger! Only if the compiler can successfully move the computation across the if-test will there be a likely benefit.

Consider the factors that the compiler would have to consider in moving the computation and assignment of B:

- What are the relative execution frequencies of the then-case and the else-case in the branch? If the then-case is much more frequent, the code motion may be beneficial. If not, it is less likely, although not impossible to consider moving the code.

- What is the cost of executing the computation and assignment to B above the branch? It may be that there are a number of empty instruction issue slots in the code above the branch and that the instructions for B can be placed into these slots that would otherwise go empty. This opportunity makes the computation of B “free” at least to first order.

- How will the movement of B change the execution time for the then-case? If B is at the start of the critical path for the then-case, moving it may be highly beneficial.

- Is B the best code fragment that can be moved above the branch? How does it compare with moving C or other statements within the then-case?

- What is the cost of the compensation code that may be necessary for the else-case? How effectively can this code be scheduled and what is its impact on execution time?

As we can see from this partial list, global code scheduling is an extremely complex problem. The tradeoffs depend on many factors and individual decisions to globally schedule instructions are highly interdependent. Even choosing which instructions to start considering as candidates for global code motion is complex!

To try to simplify this process, several different methods for global code scheduling have been developed. The two methods we briefly explore here rely on a simple principle: focus the attention of the compiler on a straightline code segment representing what is estimated to be the most frequently executed code path. Unrolling is used to generate the straightline code, but, of course, the complexity arises in how conditional branches are handled. In both cases, they are effectively straightened by choosing and scheduling the most frequent path.
Trace Scheduling: Focusing on the Critical Path

Trace scheduling is useful for processors with a large number of issues per clock, where conditional or predicated execution (see Section 4.5) is inappropriate or unsupported, and where simple loop unrolling may not be sufficient by itself to uncover enough ILP to keep the processor busy. Trace scheduling is a way to organize the global code motion process, so as to simplify the code scheduling by incurring the costs of possible code motion on the less frequent paths. Because it can generate significant overheads on the designated infrequent path, it is best used where profile information indicates significant differences in frequency between different paths and where the profile information is highly indicative of program behavior independent of the input. Of course, this limits its effective applicability to certain classes of programs.

There are two steps to trace scheduling. The first step, called trace selection, tries to find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions; this sequence is called a trace. Loop unrolling is used to generate long traces, since loop branches are taken with high probability. Additionally, by using static branch prediction, other conditional branches are also chosen as taken or not taken, so that the resultant trace is a straight-line sequence resulting from concatenating many basic blocks. If, for example, the program fragment shown in Figure 4.8 on page 253 corresponds to an inner loop with the highlighted path being much more frequent, and the loop were unwound four times, the primary trace would consist of four copies of the shaded portion of the program, as shown in Figure 4.9.

Once a trace is selected, the second process, called trace compaction, tries to squeeze the trace into a small number of wide instructions. Trace compaction is code scheduling; hence, it attempts to move operations as early as it can in a sequence (trace), packing the operations into as few wide instructions (or issue packets) as possible.

The advantage of the trace scheduling approach is that it simplifies the decisions concerning global code motion. In particular, branches are viewed as jumps into or out of the selected trace, which is assumed to the most probable path. When code is moved across such trace entry and exit points, additional bookkeeping code will often be needed on the entry or exit point. The key assumption is that the trace is so much more probable than the alternatives that the cost of the bookkeeping code need not be a deciding factor: if an instruction can be moved and make the main trace execute faster, it is moved.

Although trace scheduling has been successfully applied to scientific code with its intensive loops and accurate profile data, it remains unclear whether this approach is suitable for programs that are less simply characterized and less loop-intensive. In such programs, the significant overheads of compensation code may make trace scheduling an unattractive approach, or, at best, its effective use will be extremely complex for the compiler.
FIGURE 4.9 This trace is obtained by assuming that the program fragment in Figure 4.8 is the inner loop and unwinding it four times treating the shaded portion in Figure 4.8 as the likely path. The trace exits correspond to jumps off the frequent path, and the trace entrances correspond to returns to the trace.
Superblocks
One of the major drawbacks of trace scheduling is that the entries and exits into the middle of the trace cause significant complications requiring the compiler to generate and track the compensation code and often making it difficult to assess the cost of such code. Superblocks are formed by a process similar to that used for traces, but, are a form of extended basic blocks, which are restricted to have a single entry point but allow multiple exits.

Because superblocks have only a single entry point, compacting a superblock is easier than compacting a trace since only code motion across an exit need be considered. In our earlier example, we would form superblock that did not contain any entrances and hence, moving C would be easier. Furthermore, in loops that have a single loop exit based on a count (for example, a for-loop with no loop exit other than the loop termination condition), the resulting superblocks have only one exit as well as one entrance. Such blocks can then be scheduled more easily.

How can a superblock with only one entrance be constructed? The answer is to use tail duplication to create a separate block that corresponds to the portion of the trace after the entry. In our example above, each unrolling of the loop would create an exit from the superblock to a residual loop that handles the remaining iterations. Figure 4.10 shows the superblock structure if the code fragment from Figure 4.8 is treated as the body of an inner loop and unrolled four times. The residual loop handles any iterations that occur if the superblock is exited, which, in turn, occurs when the unpredicted path is selected. If the expected frequency of the residual loop were still high, a superblock could be created for that loop as well.

The superblock approach reduces the complexity of bookkeeping and scheduling versus the more general trace generation approach, but may enlarge code size more than a trace-based approach. Like trace scheduling, superblock scheduling may be most appropriate when other techniques (if-conversion, e.g.) fail. Even in such cases, assessing the cost of code duplication may limit the usefulness of the approach and will certainly complicate the compilation process.

Loop unrolling, software pipelining, trace scheduling, and superblock scheduling all aim at trying to increase the amount of ILP that can be exploited by a processor issuing more than one instruction on every clock cycle. The effectiveness of each of these techniques and their suitability for various architectural approaches are among the hottest topics being actively pursued by researchers and designers of high-speed processors.
FIGURE 4.10  This superblock results from unrolling the code in Figure 4.8 four times and creating a superblock.
Techniques such as loop unrolling, software pipelining, and trace scheduling can be used to increase the amount of parallelism available when the behavior of branches is fairly predictable at compile time. When the behavior of branches is not well known, compiler techniques alone may not be able to uncover much ILP. In such cases, the control dependences may severely limit the amount of parallelism that can be exploited. Similarly, potential dependences between memory reference instructions could prevent code movement that would increase available ILP. This section introduces several techniques that can help overcome such limitations.

The first is an extension of the instruction set to include conditional or predicated instructions. Such instructions can be used to eliminate branches converting a control dependence into a data dependence and potentially improving performance. Such approaches are useful with either the hardware-intensive schemes of the last chapter or the software-intensive approaches discussed in this chapter, since in both cases, predication can be used to eliminate branches.

Hardware speculation with in-order commit preserved exception behavior by detecting and raising exceptions only at commit time when the instruction was no longer speculative. To enhance the ability of the compiler to speculatively move code over branches, while still preserving the exception behavior, we consider several different methods, which either include explicit checks for exceptions or techniques to ensure that only those exceptions that should arise are generated.

Finally, the hardware speculation schemes of the last chapter provided support for reordering loads and stores, by checking for potential address conflicts at runtime. To allow the compiler to reorder loads and stores when it suspects they do not conflict, but cannot be absolutely certain, a mechanism for checking for such conflicts can be added to the hardware. This mechanism permits additional opportunities for memory reference speculation.

### Conditional or Predicated Instructions

The concept behind conditional instructions is quite simple: An instruction refers to a condition, which is evaluated as part of the instruction execution. If the condition is true, the instruction is executed normally; if the condition is false, the execution continues as if the instruction was a no-op. Many newer architectures include some form of conditional instructions. The most common example of such an instruction is conditional move, which moves a value from one register to another if the condition is true. Such an instruction can be used to completely eliminate a branch in simple code sequences.
EXAMPLE Consider the following code:

\[
\text{if (A==0) \{S=T;\}}
\]

Assuming that registers R1, R2, and R3 hold the values of A, S, and T, respectively, show the code for this statement with the branch and with the conditional move.

ANSWER The straightforward code using a branch for this statement is (remember that we are assuming normal rather than delayed branches)

\[
\begin{align*}
\text{BNEZ R1, L} \\
\text{ADDU R2, R3, R0}
\end{align*}
\]

L:

Using a conditional move that performs the move only if the third operand is equal to zero, we can implement this statement in one instruction:

\[
\text{CMOVZ R2, R3, R1}
\]

The conditional instruction allows us to convert the control dependence present in the branch-based code sequence to a data dependence. (This transformation is also used for vector computers, where it is called if-conversion.) For a pipelined processor, this moves the place where the dependence must be resolved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline where the register write occurs.

One obvious use for conditional move is to implement the absolute value function: \( A = \text{abs}(B) \), which is implemented as \( \text{if (B<0) \{A=-B;\}} \) else \{A=B;\}. This if statement can be implemented as a pair of conditional moves, or as one unconditional move (A=B) and one conditional move (A=-B).

In the example above or in the compilation of absolute value, conditional moves are used to change a control dependence into a data dependence. This enables us to eliminate the branch and possibly improve the pipeline behavior. As issue rates increase, designers are faced with one of two choices: execute multiple branches per clock cycle or find a method to eliminate branches to avoid this requirement. Handling multiple branches per clock is complex, since one branch must be control dependent on the other. The difficulty of accurately predicting two branch outcomes, updating the prediction tables, and executing the correct sequence, has so far caused most designers to avoid processors that execute multiple branches per clock. Conditional moves and predicated instructions provide a way of reducing the branch pressure. In addition, a conditional move can often eliminate a branch that is hard to predict, increasing the potential gain.

Conditional moves are the simplest form of conditional or predicated instructions, and although useful for short sequences, have limitations. In particular, using conditional move to eliminate branches that guard the execution of large blocks of code can be inefficient, since many conditional moves may need to be introduced.
To remedy the inefficiency of using conditional moves, some architectures support full predication, whereby the execution of all instructions is controlled by a predicate. When the predicate is false, the instruction becomes a no-op. Full predication allows us to simply convert large blocks of code that are branch dependent. For example, an if-then-else statement within a loop can be entirely converted to predicated execution, so that the code in the then-case executes only if the value of the condition is true, and the code in the else-case executes only if the value of the condition is false. Predication is particularly valuable with global code scheduling, since it can eliminate nonloop branches, which significantly complicate instruction scheduling.

Predicated instructions can also be used to speculatively move an instruction that is time-critical, but may cause an exception if moved before a guarding branch. Although it is possible to do this with conditional move, it is more costly, as we explore in the exercises.

**EXAMPLE**

Here is a code sequence for a two-issue superscalar that can issue a combination of one memory reference and one ALU operation, or a branch by itself, every cycle:

<table>
<thead>
<tr>
<th>First instruction slot</th>
<th>Second instruction slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1,40(R2)</td>
<td>ADD R3,R4,R5</td>
</tr>
<tr>
<td></td>
<td>ADD R6,R3,R7</td>
</tr>
<tr>
<td>BEQZ R10,L</td>
<td></td>
</tr>
<tr>
<td>LW R8,0(R10)</td>
<td></td>
</tr>
<tr>
<td>LW R9,0(R8)</td>
<td></td>
</tr>
</tbody>
</table>

This sequence wastes a memory operation slot in the second cycle and will incur a data dependence stall if the branch is not taken, since the second LW after the branch depends on the prior load. Show how the code can be improved using a predicated form of LW.

**ANSWER**

Call the predicated version load word LWC and assume the load occurs unless the third operand is 0. The LW immediately following the branch can be converted to a LWC and moved up to the second issue slot:

<table>
<thead>
<tr>
<th>First instruction slot</th>
<th>Second instruction slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1,40(R2)</td>
<td>ADD R3,R4,R5</td>
</tr>
<tr>
<td>LWC R8,20(R10),R10</td>
<td>ADD R6,R3,R7</td>
</tr>
<tr>
<td>BEQZ R10,L</td>
<td></td>
</tr>
<tr>
<td>LW R9,0(R8)</td>
<td></td>
</tr>
</tbody>
</table>
This improves the execution time by several cycles since it eliminates one instruction issue slot and reduces the pipeline stall for the last instruction in the sequence. Of course, if the compiler mispredicted the branch, the predicated instruction will have no effect and will not improve the running time. This is why the transformation is speculative.

If the sequence following the branch were short, the entire block of code might be converted to predicated execution, and the branch eliminated.

When we convert an entire code segment to predicated execution or speculatively move an instruction and make it predicted, we remove a control dependence. Correct code generation and the conditional execution of predicated instructions ensure that we maintain the data flow enforced by the branch. To ensure that the exception behavior is also maintained, a predicated instruction must not generate an exception if the predicate is false. The property of not causing exceptions is quite critical, as the Example above shows: If register R10 contains zero, the instruction LW R8,0(R10) executed unconditionally is likely to cause a protection exception, and this exception should not occur. Of course, if the condition is satisfied (i.e. R10 is not zero), the LW may still cause a legal and resumable exception (e.g., a page fault), and the hardware must take the exception when it knows that the controlling condition is true.

The major complication in implementing predicated instructions is deciding when to annul an instruction. Predicated instructions may either be annulled during instruction issue or later in the pipeline before they commit any results or raise an exception. Each choice has a disadvantage. If predicated instructions are annulled early in the pipeline, the value of the controlling condition must be known early to prevent a stall for a data hazard. Since data dependent branch conditions, which tend to be less predictable, are candidates for conversion to predicated execution, this choice can lead to more pipeline stalls. Because of this potential for data hazard stalls, no design with predicated execution (or conditional move) annuls instructions early. Instead, all existing processors annul instructions later in the pipeline, which means that annulled instructions will consume functional unit resources and potentially have a negative impact on performance. A variety of other pipeline implementation techniques, such as forwarding, interact with predicated instructions further complicating the implementation.

Predicated or conditional instructions are extremely useful for implementing short alternative control flows, for eliminating some unpredictable branches, and for reducing the overhead of global code scheduling. Nonetheless, the usefulness of conditional instructions is limited by several factors:

- Predicated instructions that are annulled (i.e., whose conditions are false) still take some processor resources. An annulled predicated instruction requires fetch resources at a minimum, and in most processors functional unit execution
time. Therefore, moving an instruction across a branch and making it conditional will slow the program down whenever the moved instruction would not have been normally executed. Likewise, predating a control dependent portion of code and eliminating a branch may slow down the processor if that code would not have been executed. An important exception to these situations occurs when the cycles used by the moved instruction when it is not performed would have been idle anyway (as in the superscalar example above). Moving an instruction across a branch or converting a code segment to predicated execution is essentially speculating on the outcome of the branch. Conditional instructions make this easier but do not eliminate the execution time taken by an incorrect guess. In simple cases, where we trade a conditional move for a branch and a move, using conditional moves or predication is almost always better. When longer code sequences are made conditional, the benefits are more limited.

Predicated instructions are most useful when the predicate can be evaluated early. If the condition evaluation and predicated instructions cannot be separated (because of data dependences in determining the condition), then a conditional instruction may result in a stall for a data hazard. With branch prediction and speculation, such stalls can be avoided, at least when the branches are predicted accurately.

The use of conditional instructions can be limited when the control flow involves more than a simple alternative sequence. For example, moving an instruction across multiple branches requires making it conditional on both branches, which requires two conditions to be specified or requires additional instructions to compute the controlling predicate. If such capabilities are not present, the overhead of if-conversion will be larger, reducing its advantage.

Conditional instructions may have some speed penalty compared with unconditional instructions. This may show up as a higher cycle count for such instructions or a slower clock rate overall. If conditional instructions are more expensive, they will need to be used judiciously.

For these reasons, many architectures have included a few simple conditional instructions (with conditional move being the most frequent), but only a few architectures include conditional versions for the majority of the instructions. The MIPS, Alpha, Power-PC, SPARC and Intel x86 (as defined in the Pentium processor) all support conditional move. The IA-64 architecture supports full predication for all instructions, as we will see section 4.7.

Compiler Speculation with Hardware Support

As we saw earlier in this chapter, many programs have branches that can be accurately predicted at compile time either from the program structure or by using a profile. In such cases, the compiler may want to speculate either to improve the
Hardware Support for Exposing More Parallelism at Compile-Time

scheduling or to increase the issue rate. Predicated instructions provide one method to speculate, but they are really more useful when control dependences can be completely eliminated by if-conversion. In many cases, we would like to move speculated instructions not only before branch, but before the condition evaluation, and predication cannot achieve this.

As pointed out earlier, to speculate ambitiously requires three capabilities:

1. the ability of the compiler to find instructions that, with the possible use of register renaming, can be speculatively moved and not affect the program data flow,

2. the ability to ignore exceptions in speculated instructions, until we know that such exceptions should really occur, and

3. the ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts.

The first of these is a compiler capability, while the last two require hardware support, which we explore next.

Hardware Support for Preserving Exception Behavior

To speculate ambitiously, we must be able to move any type of instruction and still preserve its exception behavior. The key to being able to do this is to observe that the results of a speculated sequence that is mispredicted will not be used in the final computation, and such a speculated instruction should not cause an exception.

There are four methods that have been investigated for supporting more ambitious speculation without introducing erroneous exception behavior:

1. The hardware and operating system cooperatively ignore exceptions for speculative instructions. As we will see below, this approach preserves exception behavior for correct programs, but not for incorrect ones. This approach may be viewed as unacceptable for some programs, but it has been used, under program control, as a “fast mode” in several processors.

2. Speculative instructions that never raise exceptions are used, and checks are introduced to determine when an exception should occur.

3. A set of status bits, called poison bits, are attached to the result registers written by speculated instructions when the instructions cause exceptions. The poison bits cause a fault when a normal instruction attempts to use the register.

4. A mechanism is provided to indicate that an instruction is speculative and the hardware buffers the instruction result until it is certain that the instruction is no longer speculative.
To explain these schemes, we need to distinguish between exceptions that indicate a program error and would normally cause termination, such as a memory protection violation, and those that are handled and normally resumed, such as a page fault. Exceptions that can be resumed can be accepted and processed for speculative instructions just as if they were normal instructions. If the speculative instruction should not have been executed, handling the unneeded exception may have some negative performance effects, but it cannot cause incorrect execution. The cost of these exceptions may be high, however, and some processors use hardware support to avoid taking such exceptions, just as processors with hardware speculation may take some exceptions in speculative mode, while avoiding others until an instruction is known not to be speculative.

Exceptions that indicate a program error should not occur in correct programs, and the result of a program that gets such an exception is not well defined, except perhaps when the program is running in a debugging mode. If such exceptions arise in speculated instructions, we cannot take the exception until we know that the instruction is no longer speculative.

In the simplest method for preserving exceptions, the hardware and the operating system simply handle all resumable exceptions when the exception occurs and simply return an undefined value for any exception that would cause termination. If the instruction generating the terminating exception was not speculative, then the program is in error. Note that instead of terminating the program, the program is allowed to continue, though it will almost certainly generate incorrect results. If the instruction generating the terminating exception is speculative, then the program may be correct and the speculative result will simply be unused; thus, returning an undefined value for the instruction cannot be harmful. This scheme can never cause a correct program to fail, no matter how much speculation is done. An incorrect program, which formerly might have received a terminating exception, will get an incorrect result. This is acceptable for some programs, assuming the compiler can also generate a normal version of the program, which does not speculate and can receive a terminating exception.

**Example**

Consider the following code fragment from an if-then-else statement of the form

```
if (A==0) A = B; else A = A+4;
```

where A is at 0(R3) and B is at 0(R2):

```
LD R1,0(R3) ;load A
BNEZ R1,L1 ;test A
LD R1,0(R2) ;then clause
J L2 ;skip else
L1: DADDI R1,R1,#4 ;else clause
L2: SD 0(R3),R1 ;store A
```

Assume the then clause is *almost always* executed. Compile the code using compiler-based speculation. Assume R14 is unused and available.
Here is the new code:

```assembly
LD R1, 0(R3); load A
LD R14, 0(R2); speculative load B
BEQZ R1,L3; other branch of the if
ADD DI R14, R1, #4; the else clause
L3: SD 0(R3), R14; nonspeculative store
```

The then clause is completely speculated. We introduce a temporary register to avoid destroying R1 when B is loaded; if the load is speculative R14 will be useless. After the entire code segment is executed, A will be in R14. The else clause could have also been compiled speculatively with a conditional move, but if the branch is highly predictable and low cost, this might slow the code down, since two extra instructions would always be executed as opposed to one branch.

In such a scheme, it is not necessary to know that an instruction is speculative. Indeed, it is helpful only when a program is in error and receives a terminating exception on a normal instruction; in such cases, if the instruction were not marked as speculative, the program could be terminated.

In this method for handling speculation, as in the next one, renaming will often be needed to prevent speculative instructions from destroying live values. Renaming is usually restricted to register values. Because of this restriction, the targets of stores cannot be destroyed and stores cannot be speculative. The small number of registers and the cost of spilling will act as one constraint on the amount of speculation. Of course, the major constraint remains the cost of executing speculative instructions when the compiler's branch prediction is incorrect.

A second approach to preserving exception behavior when speculating introduces speculative versions of instructions that do not generate terminating exceptions and instructions to check for such exceptions. This combination preserves the exception behavior exactly.

**Example** Show how the previous example can be coded using a speculative load (sLD) and a speculation check instruction (SPECCK) to completely preserve exception behavior. Assume R14 is unused and available.

**Answer** Here is the code that achieves this:

```assembly
LD R1, 0(R3); load A
sLD R14, 0(R2); speculative, no termination
BNEZ R1, L1; test A
SPECCK 0(R2); perform speculation check
```
Notice that the speculation check requires that we maintain a basic block for the then-case. If we had speculated only a portion of the then-case, then a basic block representing the then-case would exist in any event. More importantly, notice that checking for a possible exception requires extra code.

A third approach for preserving exception behavior tracks exceptions as they occur but postpones any terminating exception until a value is actually used, preserving the occurrence of the exception, although not in a completely precise fashion. The scheme is simple: A poison bit is added to every register and another bit is added to every instruction to indicate whether the instruction is speculative. The poison bit of the destination register is set whenever a speculative instruction results in a terminating exception; all other exceptions are handled immediately. If a speculative instruction uses a register with a poison bit turned on, the destination register of the instruction simply has its poison bit turned on. If a normal instruction attempts to use a register source with its poison bit turned on, the instruction causes a fault. In this way, any program that would have generated an exception still generates one, albeit at the first instance where a result is used by an instruction that is not speculative. Since poison bits exist only on register values and not memory values, stores are never speculative and thus trap if either operand is “poison.”

**EXAMPLE** Consider the code fragment from page 267 and show how it would be compiled with speculative instructions and poison bits. Show where an exception for the speculative memory reference would be recognized. Assume R14, is unused and available.

**ANSWER** Here is the code (an "s" proceeding the opcode indicates a speculative instruction):

```
J L2 ;skip else
L1: DADDI R14,R1,#4 ;else clause
L2: SD 0(R3),R14 ;store A
```

If the speculative sLD generates a terminating exception, the poison bit of R14 will be turned on. When the nonspeculative SW instruction occurs, it will raise an exception if the poison bit for R14 is on.

One complication that must be overcome is how the OS saves the user registers on a context switch if the poison bit is set. A special instruction is needed to save and reset the state of the poison bits to avoid this problem.
The fourth and final approach listed above relies on a hardware mechanism that operates like a reorder buffer. In such an approach, instructions are marked by the compiler as speculative and include an indicator of how many branches the instruction was speculatively moved across and what branch action (taken/not taken) the compiler assumed. This last piece of information basically tells the hardware the location of the code block where the speculated instruction originally was. In practice, most of the benefit of speculation is gained by allowing movement across a single branch, and, thus, only one bit saying whether the speculated instruction came from the taken or not taken path is required. Alternatively, the original location of the speculative instruction is marked by a sentinel, which tells the hardware that the earlier speculative instruction is no longer speculative and values may be committed.

All instructions are placed in a reorder buffer when issued and are forced to commit in order, as in a hardware speculation approach. (Notice, though that no actual speculative branch prediction or dynamic scheduling occurs.) The reorder buffer tracks when instructions are ready to commit and delays the “write back” portion of any speculative instruction. Speculative instructions are not allowed to commit until the branches they have been speculatively moved over are also ready to commit, or, alternatively, until the corresponding sentinel is reached. At that point, we know whether the speculated instruction should have been executed or not. If it should have been executed and it generated a terminating exception, then we know that the program should be terminated. If the instruction should not have been executed, then the exception can be ignored. Notice that the compiler, rather than the hardware, has the job of register renaming to ensure correct usage of the speculated result, as well as correct program execution.

Hardware Support for Memory Reference Speculation

Moving loads across stores is usually done when the compiler is certain the addresses do not conflict. As we saw with the examples in section 4.1, such transformations are critical to reducing the critical path length of a code segment. To allow the compiler to undertake such code motion, when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture. The special instruction is left at the original location of the load instruction (and acts like a guardian) and the load is moved up across one or more stores.

When a speculated load is executed, the hardware saves the address of the accessed memory location. If a subsequent store changes the location before the check instruction, then the speculation has failed. If the location has not been touched then the speculation is successful. Speculation failure can be handled in two ways. If only the load instruction was speculated, then it suffices to redo the load at the point of the check instruction (which could supply the target register in addition to the memory address). If additional instructions that depended on the load were also speculated, then a fix-up sequence that re-executes all the
speculated instructions starting with the load is needed. In this case, the check instruction specifies the address where the fix-up code is located.

In this section we have seen a variety of hardware assist mechanisms. Such mechanisms are key to achieving good support with the compiler intensive approaches of this chapter. In addition, several of them can be easily integrated in the hardware-intensive approaches of the prior chapter and provide additional benefits.

4.6 Crosscutting Issues

Hardware versus Software Speculation Mechanisms

The hardware-intensive approaches to speculation in the previous chapter and the software approaches of this chapter provide alternative approaches to exploiting ILP. Some of the tradeoffs, and the limitations, for these approaches are listed below:

- To speculate extensively, we must be able to disambiguate memory references. This capability is difficult to do at compile time for integer programs that contain pointers. In a hardware-based scheme, dynamic runtime disambiguation of memory addresses is done using the techniques we saw earlier for Tomasulo’s algorithm. This disambiguation allows us to move loads past stores at runtime. Support for speculative memory references can help overcome the conservatism of the compiler, but unless such approaches are used carefully, the overhead of the recovery mechanisms may swamp the advantages.

- Hardware-based speculation works better when control flow is unpredictable, and when hardware-based branch prediction is superior to software-based branch prediction done at compile time. These properties hold for many integer programs. For example, a good static predictor has a misprediction rate of about 16% for four major integer SPEC92 programs, and a hardware predictor has a misprediction rate of under 10%. Because speculated instructions may slow down the computation when the prediction is incorrect, this difference is significant. One result of this difference is that even statically scheduled processors normally include dynamic branch predictors.

- Hardware-based speculation maintains a completely precise exception model even for speculated instructions. Recent software-based approaches have added special support to allow this as well.

- Hardware-based speculation does not require compensation or bookkeeping code, which is needed by ambitious software speculation mechanisms.

- Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling than a purely hardware-driv-
Hardware-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations of an architecture. Although this advantage is the hardest to quantify, it may be the most important in the long run. Interestingly, this was one of the motivations for the IBM 360/91. On the other hand, more recent explicitly parallel architectures, such as IA-64, have added flexibility that reduces the hardware dependence inherent in a code sequence.

Against these advantages stands a major disadvantage: supporting speculation in hardware is complex and requires additional hardware resources. This hardware cost must be evaluated against both the complexity of a compiler for a software-based approach and the amount and usefulness of the simplifications in a processor that relies on such a compiler. We return to this topic in the concluding remarks.

Some designers have tried to combine the dynamic and compiler-based approaches to achieve the best of each. Such a combination can generate interesting and obscure interactions. For example, if conditional moves are combined with register renaming, a subtle side-effect appears. A conditional move that is annulled must still copy a value to the destination register, since it was renamed earlier in the instruction pipeline. These subtle interactions complicate the design and verification process and can also reduce performance. For example, in the Alpha 21264 this problem is overcome by mapping conditional to two instructions in the pipeline.

4.7 Putting It All Together: The Intel IA-64 Architecture and Itanium Processor

This section is an overview of the Intel IA-64 architecture and the initial implementation, the Itanium processor.

The Intel IA-64 Instruction Set Architecture

The IA-64 is a RISC-style, register-register instruction set, but with many novel features designed to support compiler-based exploitation of ILP. Our focus here is on the unique aspects of the IA-64 ISA. Most of these aspects have been discussed already in this chapter, including predication, compiler-based parallelism detection, and support for memory reference speculation.

The IA-64 Register Model
The components of the IA-64 register state are:
128 64-bit general-purpose registers, which as we will see shortly are actually 65 bits wide;

128 82-bit floating point registers, which provides two extra exponent bits over the standard 80-bit IEEE format,

64 1-bit predicate registers,

8 64-bit branch registers, which are used for indirect branches, and

a variety of registers used for system control, memory mapping, performance counters, and communication with the OS.

The integer registers are configured to help accelerate procedure calls using a register stack mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture. Registers 0-31 are always accessible and are addressed as 0-31. Registers 32-128 are used as a register stack and each procedure is allocated a set of registers (from 0 to 96) for its use. The new register stack frame is created for a called procedure by renaming the registers in hardware; a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure. The frame consists of two parts: the local area and the output area. The local area is used for local storage, while the output area is used to pass values to any called procedure. The alloc instruction specifies the size of these areas. Only the integer registers have register stack support.

On a procedure call, the CFM pointer is updated so that R32 of the called procedure points to the first register of the output area of the called procedure. This update enables the parameters of the caller to be passed into the addressable registers of the callee. The callee executes an alloc instruction to allocate both the number of required local registers, which include the output registers of the caller, and the number of output registers needed for parameter passing to a called procedure. Special load and store instructions are available for saving and restoring the register stack, and special hardware (called the register stack engine) handles overflow of the register stack.

In addition to the integer registers, there are three other sets of registers: the floating point registers, the predicate registers, and the branch registers. The floating point registers are used for floating point data, and the branch registers are used to hold branch destination addresses for indirect branches. The predication registers hold predicates, which control the execution of predicated instructions; we describe the predication mechanism later in this section.

Both the integer and floating point registers support register rotation for registers 32-128. Register rotation is designed to ease the task of allocating of registers in software pipelined loops, a problem that we discussed in Section 4.4. In addition, when combined with the use of predication, it is possible to avoid the need for unrolling and for separate prologue and epilogue code for a software pipelined loop. This capability reduces the code expansion incurred to use soft-
ware pipelining and makes the technique usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages.

Instruction Format and Support for Explicit Parallelism
The IA-64 architecture is designed to achieve the major benefits of a VLIW-approach–implicit parallelism among operations in an instruction and fixed formatting of the operation fields–while maintaining greater flexibility than a VLIW normally allows. This combination is achieved by relying on the compiler to detect ILP and schedule instructions into parallel instruction slots, but adding flexibility in the formatting of instructions and allowing the compiler to indicate when an instruction cannot be executed in parallel with its successors.

The IA-64 architecture uses two different concepts to achieve the benefits of implicit parallelism and ease of instruction decode. Implicit parallelism is achieved by placing instructions into instruction groups, while the fixed formatting of multiple instructions is achieved through the introduction of a concept called a bundle, which contains three instructions. Let’s start by defining an instruction group.

An instruction group is a sequence of consecutive instructions with no register data dependences among them (there are a few minor exceptions). All the instructions in a group could be executed in parallel, if sufficient hardware resources existed and if any dependences through memory were preserved. An instruction group can be arbitrarily long, but the compiler must explicitly indicate the boundary between one instruction group and another. This boundary is indicated by placing a stop between two instructions that belong to different groups. To understand how stops are indicated, we must first explain how instructions are placed into bundles.

IA-64 instructions are encoded in bundles, which are 128 bits wide. Each bundle consists of a five-bit template field and three instructions, each 41 bits in length. To simply the decoding and instruction issue process, the template field of a bundle specifies what types of execution unit each instruction in the bundle requires. Figure 4.11 shows the five different execution unit types and describes what instruction classes they may hold, together with some examples.

The five-bit template field within each bundle describes both the presence of any stops associated with the bundle and the execution unit type required by each instruction within the bundle. Figure 4.12 shows the possible formats that the template field encodes and the position of any stops it specifies. The bundle formats can specify only a subset of all possible combinations of instruction types and stops. To see how the bundle works, let’s consider an example.

**Example**
Unroll the array increment example, x[i] = x[i] + s (introduced on page 223), seven times (see page 236 for the unrolled code) and place the instructions into bundles, first ignoring pipeline latencies (to minimize the number
of bundles) and then scheduling the code to minimize stalls. In scheduling the code assume 1 bundle executes per clock and that any stalls cause the entire bundle to be stalled. Use the pipeline latencies from Figure 4.1 on page 222. Use MIPS instruction mnemonics for simplicity.

ANSWER The two different versions are shown in Figure 4.13. Although the latencies are different from those in Itanium, the most common bundle, MMF, must be issued by itself in Itanium, just as our example assumes.

Instruction Set Basics
Before turning to the special support for speculation, we briefly discuss the major instruction encodings and survey the instructions in each of the primary five instruction classes (A, I, M, F, and B). Each IA-64 instruction is 41 bits in length. The high-order four bits, together with the bundle bits that specify the execution unit slot, are used as the major opcode. (That is, the four-bit opcode field is reused across the execution field slots, and it is appropriate to think of the opcode as being 4 bits + the M, F, I, B, L+X designation.) The low order six-bit of every instruction are used for specifying the predicate register that guards the instruction (see the next section).

Figure 4.14 summarizes most of the major instruction formats, other than the multimedia instructions, and gives examples of the instructions encoded for each format.

Predication and Speculation Support
The IA-64 architecture provides comprehensive support for predication: nearly every instruction in the IA-64 architecture can be predicated. An instruction is
predicated by a specifying a predicate register, whose identity is placed in the lower six bits of each instruction field. Because nearly all instructions can be predicated, both if-conversion and code motion have lower overhead than they would with only limited support for conditional instructions. One consequence of full predication is that a conditional branch is simply a branch with a guarding predicate!
<table>
<thead>
<tr>
<th>Bundle template</th>
<th>Slot 0</th>
<th>Slot 1</th>
<th>Slot 2</th>
<th>Execute cycle (1 bundle / cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>9: M M I</td>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>3</td>
</tr>
<tr>
<td>14: M M F</td>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F8,F6,F2</td>
<td>4</td>
</tr>
<tr>
<td>15: M M F</td>
<td>L.D F26,-48(R1)</td>
<td>S.D F4,0(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>6</td>
</tr>
<tr>
<td>15: M M F</td>
<td>S.D F8,-8(R1)</td>
<td>S.D F12,-16(R1)</td>
<td>ADD.D F16,F14,F2</td>
<td>9</td>
</tr>
<tr>
<td>15: M M F</td>
<td>S.D F16,-24(R1)</td>
<td></td>
<td>ADD.D F20,F18,F2</td>
<td>12</td>
</tr>
<tr>
<td>15: M M F</td>
<td>S.D F20,-32(R1)</td>
<td></td>
<td>ADD.D F24,F22,F2</td>
<td>15</td>
</tr>
<tr>
<td>15: M M F</td>
<td>S.D F24,-40(R1)</td>
<td></td>
<td>ADD.D F28,F26,F2</td>
<td>18</td>
</tr>
<tr>
<td>12: M M F</td>
<td>S.D F28,-48(R1)</td>
<td>DADDUI R1,R1,#-56</td>
<td>BNE R1,R2,Loop</td>
<td>21</td>
</tr>
</tbody>
</table>

a. The code scheduled to minimize the number of bundles.

<table>
<thead>
<tr>
<th>Bundle template</th>
<th>Slot 0</th>
<th>Slot 1</th>
<th>Slot 2</th>
<th>Execute cycle (1 bundle / cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8: M M I</td>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>9: M M I</td>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>3</td>
</tr>
<tr>
<td>14: M M F</td>
<td>L.D F26,-48(R1)</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>15: M M F</td>
<td>ADD.D F12,F10,F2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>14: M M F</td>
<td>S.D F4,0(R1)</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>14: M M F</td>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F20,F18,F2</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>15: M M F</td>
<td>S.D F12,-16(R1)</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>14: M M F</td>
<td>S.D F16,-24(R1)</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>9: M M I</td>
<td>S.D F20,-32(R1)</td>
<td>S.D F24,-40(R1)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>8: M M I</td>
<td>S.D F28,-48(R1)</td>
<td>DADDUI R1,R1,#-56</td>
<td>BNE R1,R2,Loop</td>
<td>11</td>
</tr>
</tbody>
</table>

b. The code scheduled to minimize the number of cycles assuming one bundle executed per cycle.

**FIGURE 4.13** The IA-64 instructions, including bundle bits and stops, for the unrolled version of \( x[i] = x[i] + s \), when unrolled seven times and scheduled (a) to minimize the number of instruction bundles and (b) to minimize the number of cycles (assuming that a hazard stalls an entire bundle). Blank entries indicate unused slots, which are encoded as no-ops. The absence of stops indicates that some bundles could be executed in parallel. Minimizing the number of bundles yields 9 bundles versus the 11 needed to minimize the number of cycles. The scheduled version executes in just over half the number of cycles. Version (a) fills 85% of the instructions slots, while (b) fills 70%. The number of empty slots in the scheduled code and the use of bundles may lead to code sizes that are much larger than other RISC architectures.
### Instruction Format Table

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th># Formats</th>
<th>Representative Instructions</th>
<th>Extra Opcode Bits</th>
<th>GPRs/FPRs</th>
<th>Imm. Bits</th>
<th>Other/Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8</td>
<td>add, subtract, and, or</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>2-bit shift count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift left and add</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALU immediates</td>
<td>9</td>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add immediate</td>
<td>3</td>
<td>2</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add immediate</td>
<td>0</td>
<td>2</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>2 predicate register destinations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare immediate</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td>2 predicate register destinations</td>
</tr>
<tr>
<td>I</td>
<td>29</td>
<td>Shift R/L variable</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>Many multimedia instructions use this format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test bit</td>
<td>6</td>
<td>3</td>
<td>6-bit field specifier</td>
<td>2 predicate register destinations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Move to BR</td>
<td>6</td>
<td>1</td>
<td>9-bit branch predict</td>
<td>branch register specifier</td>
</tr>
<tr>
<td>M</td>
<td>46</td>
<td>Integer/FP load and store, Line prefetch</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>Speculative /non-speculative</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integer/FP load/store, and line prefetch &amp; post-increment by immediate</td>
<td>9</td>
<td>2</td>
<td>8</td>
<td>Speculative /non-speculative</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integer/FP load prefetch &amp; register postincrement</td>
<td>10</td>
<td>3</td>
<td>Speculative /non-speculative</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integer/FP speculation check</td>
<td>3</td>
<td>1</td>
<td>21 in two fields</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>9</td>
<td>PC-relative branch, counted branch</td>
<td>7</td>
<td>0</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC relative call</td>
<td>4</td>
<td>0</td>
<td>21</td>
<td>1 branch register</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>FP arithmetic</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP compare</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6-bit predicate regs</td>
</tr>
<tr>
<td>L+X</td>
<td>4</td>
<td>Move immediate long</td>
<td>2</td>
<td>1</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 4.14** A summary of some of the instruction formats of the IA-64 ISA is shown. The major opcode bits and the guarding predication register specifier add 10 bits to every instruction. The number of formats indicated for each instruction class in the second column (a total of 111) is a strict interpretation: where a different use of a field, even of the same size, is considered a different format. The number of formats that actually have different field sizes is one-third to one-half as large. Some instructions have unused bits that are reserved, we have not included those in this table. Immediate bits include the sign bit. The branch instructions include prediction bits, which are used when the predictor does not have a valid prediction. None of the many formats for the multimedia instructions are shown in this table.
Predicate registers are set using compare or test instructions. A compare instruction specifies one of ten different comparison tests and two predicate registers as destinations. The two predicate registers are written either with the result of the comparison (0 or 1) and the complement, or with some logical function that combines the two tests (such as and) and the complement. This capability allows multiple comparisons to be done in one instruction.

Speculation support in the IA-64 architecture consists of separate support for control speculation, which deals with deferring exception for speculated instructions, and memory reference speculation, which supports speculation of load instructions.

Deferred exception handling for speculative instructions is supported by providing the equivalent of poison bits. For the GPRs, these bits are called NaTs for Not a Thing, and this extra bit makes the GPRs effectively 65 bits wide. For the FP registers this capability is obtained using a special value, NaTVal, for Not A Thing Value; this value is encoded using a significand of 0 and an exponent outside of the IEEE range. Only speculative load instructions generate such values, but all instructions that do not affect memory will cause a NaT or NatVal to be propagated to the result register. (There are both speculative and nonspeculative loads; the latter can only raise immediate exceptions and cannot defer them.) Floating point exceptions are not handled through this mechanism, but use floating point status registers to record exceptions.

A deferred exception can be resolved in two different ways. First, if a non-speculative instruction, such as a store, receives a NaT or NaTVal as a source operand, it generates an immediate and unrecoverable exception. Alternatively, a \texttt{chk.s} instruction can be used to detect the presence of NaT or NatVal and branch to a routine designed by the compiler to recover from the speculative operation. Such a recovery approach makes more sense for memory reference speculation.

The inability to store the contents of instructions with NaT or NatVal set would make it impossible for the OS to save the state of the processor. Thus, IA-64 includes special instructions to save and restore registers that do not cause an exception for a NaT or NaTVal and also save and restore the NaT bits.

Memory reference support in the IA-64 uses the a concept called advanced loads. An \textit{advanced load} is a load that has been speculatively moved above store instructions on which it is potentially dependent. To speculatively perform a load, the \texttt{ld.a} (for advanced load) instruction is used. Executing this instruction creates an entry in a special table, called the \textit{ALAT}. The ALAT stores both the register destination of the load and the address of the accessed memory location. When a store is executed, an associative look-up against the active ALAT entries is performed. If there is an ALAT entry with the same memory address as the store, the ALAT entry is marked as invalid.

Before any nonspeculative instruction (i.e., a store) uses the value generated by an advanced load or a value derived from the result of an advanced load, an explicit check is required. The check specifies the destination register of the advanced load. If the ALAT for that register is still valid, the speculation was legal.
and the only effect of the check is to clear the ALAT entry. If the check fails, the action taken depends on which of two different types of checks was employed. The first type of check is an instruction \texttt{ld.c}, which simply causes the data to be reloaded from memory at that point. An \texttt{ld.c} instruction is used when only the load is advanced. The alternative form of a check, \texttt{chk.a}, specifies the address of a fix-up routine that is used to re-execute the load \textit{and any other} speculated code that depended on the value of the load.

The Itanium Processor

The Itanium processor is the first implementation of the IA-64 architecture. It became available in the middle of 2001 with an 800 MHz clock. The processor core is capable of up to six issues per clock, with up to three branches and two memory references. The memory hierarchy consists of a three-level cache. The first level uses split instruction and data caches; floating point data is not placed in the first level cache. The second and third levels are unified caches, with the third level being an off-chip 4MB cache placed in the same container as the Itanium die.

Functional Units and Instruction Issue

There are nine functional units in the Itanium processor: 2 I-units, 2 M-units, 3 B-units, and 2 F-units. All the functional units are pipelined. Figure 4.15 gives the pipeline latencies for some typical instructions. In addition, when a result is bypassed from one unit to another, there is usually at least one additional cycle of delay.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer load</td>
<td>1</td>
</tr>
<tr>
<td>Floating point load</td>
<td>9</td>
</tr>
<tr>
<td>Correctly predicted taken branch</td>
<td>0-3</td>
</tr>
<tr>
<td>Mispredicted branch</td>
<td>9</td>
</tr>
<tr>
<td>Integer ALU operations</td>
<td>0</td>
</tr>
<tr>
<td>FP arithmetic</td>
<td>4</td>
</tr>
</tbody>
</table>

FIGURE 4.15 The latency of some typical instructions on Itanium. The latency is defined as the smallest number of intervening instructions between two dependent instructions. Integer load latency assumes a hit in the first-level cache. FP loads always bypass the primary cache, so the latency is until a hit in the second-level cache. There are some minor restrictions for the some of the functional units, but these primarily involve the execution of infrequent instructions.
Itanium has an instruction issue window that contains up to two bundles at any given time. With this window size, Itanium can issue up to six instructions in a clock. In the worst case, if a bundle is split when it is issued, the hardware could see as few as four instructions: one from the first bundle to be executed and three from the second bundle. Instructions are allocated to functional units based on the bundle bits, ignoring the presence of no-ops or predicated instructions with untrue predicates. In addition, when issue to a functional unit is blocked, because the next instruction to be issued needs an already committed unit, the resulting bundle is split. A split bundle still occupies one of the two bundle slots, even if it has only one instruction remaining. In addition, there are several Itanium-dependent restrictions that cause a bundle to be split and issue a stop. For example, an MMF bundle, which contains two memory type instructions and a floating point type instruction, always generates a split before this bundle and after this bundle. This issue limitation means that a sequence of MMF bundles (like those in our earlier example shown in 4.13 on page 276) can execute at most three instructions per clock, even if no data dependences are present and no cache misses occur.

The Itanium processor uses a 10-stage pipeline divided into four major parts:

- Front-end (stages IPG, Fetch, and Rotate): prefetches up to 32 bytes per clock (2 bundles) into a prefetch buffer, which can hold up to eight bundles (24 instructions). Branch prediction is done using a multilevel adaptive predictor like that in the P6 microarchitecture we saw in Chapter 3.

- Instruction delivery (stages EXP and REN): distributes up to six instructions to the nine functional units. Implements registers renaming for both rotation and register stacking.

- Operand delivery (WLD and REG): accesses the register file, performs register bypassing, accesses and updates a register scoreboard, and checks predicate dependencies. The scoreboard is used to detect when individual instructions can proceed, so that a stall of one instruction in a bundle need not cause the entire bundle to stall. (As we saw in Figure 4.13 on page 276, stalling the entire bundle leads to poor performance unless the instructions are carefully scheduled.)

- Execution (EXE, DET, and WRB): executes instructions through ALUs and load/store units, detects exceptions and posts NaTs, retires instructions and performs write-back.

Remarkably, the Itanium has many of the features more commonly associated with the dynamically-scheduled pipelines described in the last chapter: strong emphasis on branch prediction, register renaming, scoreboard, a deep pipeline with many stages before execution (to handle instruction alignment, renaming, etc.), and several stages following execution to handle exception detection. It is somewhat surprising that an approach whose goal is to rely on compiler technol-
ogy and simpler hardware seems to be at least as complex as the dynamically scheduled processors we saw in the last chapter!

Itanium Performance

Figure 4.16 shows the performance of an 800 MHz Itanium versus a 1 GHz Alpha 21264 and a 2 GHz Pentium 4 for SPECint. The Itanium is only about 60% of the performance of the Pentium 4 and 68% of the performance of the Alpha 21264. What is perhaps even more surprising is that even if we normalize for clock rate, the Itanium is still only about 85% of the performance of the Alpha 21264, which is an older design in an older technology with about 20% less power consumption, despite its higher clock rate!

FIGURE 4.16 The SPECint benchmark set shows that the Itanium is considerably slower than either the Alpha 21264 or the Pentium 4. The Itanium system is a Hewlett Packard server rx4610 with an 800MHz Itanium and a 4 MB off-chip, level 3 cache. The Alpha system is a 1 GHz Compaq Alphaserver GS320 with only an on-chip L2 cache. The Pentium 4 system is a Compaq Precision 330 workstation with a 2 GHz part with a 256KB on-chip L2 cache. The overall SPECint_base 2000 number is computed as the geometric mean of the individual ratios.
The SPECfp benchmarks reveal a different story, as we can see in Figure 4.17. Overall, the Itanium processor is 1.08 times faster than the Pentium 4 and about 1.20 times faster than the Alpha 21264, at a clock rate that is only 40% to 80% as high. For floating point applications, the Itanium looks like a very competitive processor. As we saw in Chapter 3, floating point benchmarks exploit higher degrees of ILP well and also can make effective use of an aggressive memory system, including a large L3 cache. Both of these factors probably play a role.

There are two unusual aspects of the SPECfp performance measurements. First, the Itanium gains its performance advantage over the Pentium 4 primarily on the strength of its performance on one benchmark: “art”, where it is over four times faster than the Pentium 4. If the benchmark “art” was excluded, the Pentium 4 would outperform the Itanium for SPECfp. The other unusual aspect of
this performance data is that the Alpha processor shows a large gap of almost 30% between tuned and base performance for SPECfp. This compares to a gap between base and peak for the Itanium system of 0% and for the Pentium 4 system of 3%. Looking at the benchmark specific flags for the Alpha system, which primarily describe loop unrolling optimizations, it appears that this difference is due to compiler immaturity for the Alpha system. If the base performance could be brought to 95% of the peak performance, the Alpha system would have the highest SPECfp rating among these three processors.

As we mentioned in the last chapter, power may be most difficult hurdle in future processors and in achieving their performance goals. The limitations on power seem to be serious independent of how ILP is exploited, whether through pipelining and faster clock rates or through wider issue. The SPECfp data confirms this view. Although the Itanium achieves better floating point performance than either the Alpha 21264 or the Pentium 4, its floating point performance per watt is no better that of the Alpha 21264 and only 56% of that of the Pentium 4!

4.8 Another View: ILP in the Embedded and Mobile Markets

The Trimedia and Crusoe chips represent interesting approaches to applying the VLIW concepts in the embedded space. The Trimedia CPU is perhaps the closest current processor to a “classic” VLIW processor; it also supports a mechanism for compressing instructions while they are in main memory and the instruction cache and decompressing them during instruction fetch. This approach addresses the code size disadvantages of a VLIW processor, which would be especially troublesome in the embedded space. In contrast, the Crusoe processor uses software translation from the x86 architecture to a VLIW processor, achieving lower power consumption than typical x86 processors, which is key for Crusoe’s target market—mobile applications.

The Trimedia TM32 Architecture

The Trimedia TM32 CPU is a classic VLIW architecture: every instruction contains five operations and the processor is completely statically scheduled. In particular, the compiler is responsible for explicitly including no-ops both within an instruction—when an operation field cannot be used—and between dependent instructions. The processor does not detect hazards, which if present will lead to incorrect execution. To reduce the cost of explicit no-ops in code size, the Trimedia processor compresses the code stream until the instructions are fetched from the instruction cache when they are expanded.
A Trimedia instruction consists of five operation slots, each able to specify an operation to one functional unit or an immediate field. Each individual operation in an instruction is predicated with a single register value, which if 0 causes that operation in the instruction to be canceled. The compiler must ensure that when multiple branches are included in an instruction, at most predicate is true. Loads can be freely speculated in the Trimedia architecture, since they do not generate exceptions. (There is no support for paged virtual memory.)

The mapping between instruction slots and units is limited, both for instruction encoding reasons and to simply instruction dispatch. As Figure 4.18 shows, there are 23 functional units of 11 different types. An instruction can specify any combination that will fit within the restrictions on the five fields.

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Unit Latency</th>
<th>Operation Slots</th>
<th>Typical operations performed by functional unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>0</td>
<td>yes yes yes yes yes</td>
<td>Integer add/subtract(compare), logicals</td>
</tr>
<tr>
<td>DMem</td>
<td>2</td>
<td>yes</td>
<td>Loads and stores</td>
</tr>
<tr>
<td>DMemSpec</td>
<td>2</td>
<td>yes</td>
<td>Cache invalidate, prefetch, allocate</td>
</tr>
<tr>
<td>Shifter</td>
<td>0</td>
<td>yes</td>
<td>Shifts and rotates</td>
</tr>
<tr>
<td>DSPALU</td>
<td>1</td>
<td>yes</td>
<td>Simple DSP arithmetic operations</td>
</tr>
<tr>
<td>DSPMul</td>
<td>2</td>
<td>yes</td>
<td>DSP operations with multiplication</td>
</tr>
<tr>
<td>Branch</td>
<td>3</td>
<td>yes yes yes</td>
<td>Branches and jumps</td>
</tr>
<tr>
<td>FALU</td>
<td>2</td>
<td>yes</td>
<td>FP add, subtract</td>
</tr>
<tr>
<td>IFMul</td>
<td>2</td>
<td>yes</td>
<td>Integer and FP multiply</td>
</tr>
<tr>
<td>FComp</td>
<td>0</td>
<td>yes</td>
<td>FP compare</td>
</tr>
<tr>
<td>FTough</td>
<td>16</td>
<td>yes</td>
<td>FP divide, square root</td>
</tr>
</tbody>
</table>

**FIGURE 4.18** There are 23 functional units of 11 different types in the Trimedia CPU. This table shows the type of operations executed by each functional unit and the instruction slots available for specifying a particular functional unit. The number of instructions slots available for specifying a unit is equal to the number of copies of that unit. Hence, there are five ALU units and two FALU units.

To see how this VLIW processor operates, let’s look at an example.

**EXAMPLE** First compile the loop for the following C code into MIPS instructions, and then show what it might look like if the Trimedia processor’s operations fields were the same as MIPS instructions. (In fact, the Trimedia operation types are very close to MIPS instructions in capability.) Assume the func-
a. The MIPS code before unrolling.

Loop: LD R11, R0 (R4)  # R11 = a[i]
LD R12, R0 (R5)  # R12 = b[i]
DADDU R17, R11, R12  # R17 = a[i]+b[i]
SD 0 (R6), R17,  # c[i] = a[i]+b[i]
DADDIU R4, R4, 8  # R4 = next a[] address
DADDIU R5, R5, 8  # R5 = next b[] address
DADDIU R6, R6, 8  # R6 = next c[] address
BNE R4, R7, Loop  # if not last go to Loop

b. The MIPS code after unrolling four times and optimizing the code. For simplicity, we have assumed that n is a multiple of four.

FIGURE 4.19 The MIPS code for the integer vector sum shown in part a before unrolling and in part b after unrolling four times. These code sequences assume that the starting addresses of a, b, and c are in registers R4, R5, and R6, and that R7 contains the address of a[n].

Loop: LD R11, 0 (R4)  # load a[i]
LD R12, R0 (R5)  # load b[i]
DADDU R17, R11, R12  # load b[i]
SD 0 (R6), R17,  # c[i] = a[i]+b[i]
LD R14, 8 (R4)  # load a[i]
LD R15, 8 (R5)  # load b[i]
DADDU R18, R14, R15  # a[i]+b[i]
SD 8 (R6), R18  # c[i] = a[i]+b[i]
LD R19, 16 (R4)  # load a[i]
LD R20, 16 (R5)  # load b[i]
DADDU R21, R19, R20  # a[i]+b[i]
SD 16 (R6), R21  # c[i] = a[i]+b[i]
LD R22, 24 (R4)  # load a[i]
LD R23, 24 (R5)  # load b[i]
DADDU R24, R22, R23  # a[i]+b[i]
SD 24 (R6), R24  # c[i] = a[i]+b[i]
DADDIU R4, R4, 32  # R4 = next a[] address
DADDIU R5, R5, 32  # R5 = next b[] address
DADDIU R6, R6, 32  # R6 = next c[] address
BNE R4, R7, Loop  # if not last go to Loop
tional unit capacities and latencies shown in Figure 4.18.

```c
void sum (int a[], int b[], int c[], int n)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i]+b[i];
}
```

Unroll the loop so there are up to four copies of the body, if needed.

**ANSWER** Figure 4.19 shows the MIPS code before and after unrolling. Figure 4.20 shows the code for the Trimedia processor is shown in. (We assume that R30 contains the address of the first instruction in the sequence.) A standard MIPS processor needs 20 32-bit instructions for the unrolled loop and the Trimedia processor takes 8 instructions, meaning that 1/2 of the VLIW operation slots are full. The importance of compressing the code stream in the Trimedia CPU is clear from this example. As Figure 2.37 showed, even after compression, Trimedia code is two to three times larger than MIPS code.

![Figure 4.20](image)

**FIGURE 4.20** The Trimedia code for a simple loop summing two vectors to generate a third makes good use of multiple memory ports but still contains a large fraction of idle slots. Loops with more computation within the body would make better use of the available operation slots. The `DADDUI` and `SETEQ` operations in the second and third instruction (first slot) serve to compute the loop termination test. The `DADDUI` duplicates a later add, so that the computation can be done early enough to schedule the branch and fill the 3 branch delay slots. The loop branch uses the `JMPF` instruction that tests a register (R25) and branches to an address specified by another register (R30);

Figure 4.21 shows the performance and code size of the TM1300, a 166 MHz implementation of the TM-32 architecture, and the NEC VR5000, a 250 MHz
version of the MIPS-32 architecture using the EEMBC consumer benchmarks or
the measurements. The performance, which is plotted with columns on the left
axis, and code size, which is plotted with lines on the right axis, are both shown
relative to the NEC VR4122, a low-end embedded processor implementing the
MIPS instruction set. Two different performance measurements are shown for the
TM1300. The “out-of-box” measurement allows no changes to the source; the
optimized version allows changes, including hand-coding. In the case of the
TM1300 only source code medications and pragmas, which supply directions to
the compiler, are used. The optimized TM1300 result is almost five times faster
overall than the out-of-the-box result when the performance is summarized by
the geometric mean. The out-of-the-box result for the TM1300 is 1.6 times faster
than the VR5000.

FIGURE 4.21 The performance and the code size for the EEMBC consumer benchmarks run on the Trimedia
TM1300 and the NEC VR5000 and shown relative to the performance and code size for the low-end NEC VR4122.
The columns and the left axis show the performance of the processors normalized to the out-of-the-box performance of the
NEC VR4122. The TM1300 has a clock speed of 166 MHz and results are shown for a version with no source code changes
(the “out-of-the-box” version) and with a set of changes at the source level (the “optimized” version) consisting of code
changes and pragmas, which is then compiled. The lines and the right axis show the code size relative to the out-of-the-box
NEC VR4122 using the Green Hills Compiler. These measurements all come from the EEMBC website: http://www.eem-
bc.org/benchmark/benchmain.asp.
One cost that is paid for this performance gain is a significant increase in code size. The code size of the out-of-the-box version of the benchmarks on the TM1300 is twice as large overall as the code size on the VR5000. For the TM1300 optimized version the code size is four times larger than the VR5000 version. Imagine how much larger the code size might be, if the code compression techniques were not used in the TM-32 architecture.

This tradeoff between code size and performance illustrates a fundamental difference in the design objectives of the MIPS and TM-32 architectures. The MIPS architecture is a general-purpose architecture with some extensions for the embedded market. The TM-32 architecture is an architecture designed for specific classes of embedded applications. The much larger code size of the TM-32 would simply make it unsuitable for many market segments and its specialized instruction set might not have significant performance benefits. On the other hand, its high performance for certain important functions, especially those in media processing, may allow it to be used in place of special-purpose chips designed for a single function, such as JPEG compression or image conversion. By replacing several special-purpose dedicated chips with a single programmable processor, system cost might be reduced.

The Transmeta Crusoe Processor

The Crusoe processor is a VLIW processor designed for the low-power marketplace, especially mobile PCs and mobile Internet appliances, but, what makes it most unusual is that it achieves instruction set compatibility with the x86 instruction set through a software system that translates from the x86 instruction set to the VLIW instruction set implemented by Crusoe.

The Crusoe processor: Instruction Set Basics

The Crusoe processor is a reasonably straightforward VLIW with in-order execution. Instructions come in two sizes: 64 bits (containing two operations) and 128 bits (containing four operations).

There are five different types of operation slots:

1. ALU operations: typical RISC ALU operations with three integer register operands, each specifying one of 64 integer registers.
2. Compute: this slot may specify any integer ALU operation (there are two integer ALUs in the datapath), a floating point operation (using the 32 floating point registers), or a multimedia operation.
3. Memory: a load or store operation.
5. Immediate: a 32-bit immediate used by another operation in this instruction.
There are two different 128-bit instruction formats, characterized by what operation slots they have:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Compute</th>
<th>ALU</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Compute</td>
<td>ALU</td>
<td>Branch</td>
</tr>
</tbody>
</table>

The Crusoe processor uses a simple in-order, six-stage pipeline for integer instructions—two fetch stages, decode, register read, executive, and register writeback—and a ten-stage pipeline for floating point, which has four extra execute stages.

The Crusoe processor: software translation and hardware support

The software responsible for implementing the x86 instruction set uses a variety of techniques to establish a balance between execution speed and translation time. Initially, and for lowest overhead execution, the x86 code can be interpreted on an instruction by instruction basis. If a code segment is executed several times, it can be translated into an equivalent Crusoe code sequence, and the translation can be cached. The unit of translation is at least a basic block, since we know that if any instruction is executed in the block, they will all be executed. Translating an entire block both improves the translated code quality and reduces the translation overhead, since the translator need only be called once per basic block. Even a quick translation of a basic block can produce acceptable results, since simple code scheduling can be done during the translation.

One of the major challenges of a software-based implementation of an instruction set is maintaining the exception behavior of the original ISA while achieving good performance. In particular, achieving good performance often means reordering operations that correspond to instructions in the original program, which means that operations will be executed in a different order than in a strict sequential interpretation. This reordering is crucial to obtaining good performance when the target is a VLIW. Hence, just as other VLIW processors have found it useful to have support for speculative reordering, such support is important in Crusoe.

The Crusoe support for speculative reordering consists of four major parts: a shadowed register file, a program-controlled store buffer, memory alias detection hardware with speculative loads, and a conditional move instruction (called select) that is used to do if-conversion on x86 code sequences.

The shadowed register file and the program-controlled store buffer allow operations to be executed in a different order while ensuring that permanent state is not committed until no exceptions are possible. 48 of the integer registers and 16 of the floating point registers are shadowed. The shadow registers are used to hold the precise state and are updated only when a translated sequence that may
correspond to several x86 instructions has been executed without an exception. To indicate that the shadow registers should be updated a commit is executed, which has no overhead since every instruction has a bit used to indicate that a commit should be executed at the end of the instruction. If an exception occurs, the primary register set (called the working registers) can be restored from the shadow registers using a rollback operation. This mechanism allows out-of-order completion of register writes without sacrificing the precise exception model of the x86.

One of the most novel features of the Crusoe processor is the program-controlled write buffer. Stores generally cause irrevocable state update. Thus, in the dynamically-scheduled pipelines of Chapter 3, stores are committed in-order. Similarly, in the IA-64 architecture, stores are not speculated, since the state update cannot be undone. The Crusoe architecture provides a novel solution to this scheme: it includes the ability to control when the write buffer is allowed to update the memory. A gate instruction causes all stores to be held in the buffer, until a commit is executed. A rollback will cause the buffer to be flushed. This feature allows for speculative store execution without violating the exception model.

By using special speculative loads and stores (similar to the ld.s and chk.s mechanisms in IA-64) together with the rollback capability, the software translator can speculatively reorder loads and stores. The ld.p instruction indicates a speculative load, whose effective address is stored in a special cache. A special store, stam, indicates a store that a load was moved across; if the ld.p and stam touch the same address, then the speculative load was incorrect. A rollback is initiated and the code sequence is re-executed starting at the x86 instruction that followed the last gate.

This combination can also be used to do speculative data reuse in the case where a store intervenes between two loads that the compiler believes are to the same address. By making the first load a ld.p and the store a stam, the translator can then reuse the value of the first load, knowing that if the store was to the same address as the load, it would cause a trap. The resulting trap can then re-execute the sequence using a more conservative interpretation.

The Crusoe processor: performance measures
Since the aim of the Crusoe processor is to achieve competitive performance at low power, benchmarks that measure both performance and power are critical. Because Crusoe depends on realistic behavior to tune the code translation process, it will not perform in a predictive manner when benchmarked using simple, but unrealistic scripts. Unfortunately, existing standard benchmarks use simple scripts that do not necessarily reflect actual user behavior (for benchmarks such as Microsoft Office) in terms of both repetition and timing. To remedy this factor, Transmeta has proposed a new set of benchmark scripts. Unfortunately, these scripts have not been released and endorsed by either a group of vendors or an independent entity.
Instead of including such results, Figure 4.22 summarizes the results of benchmarks whose behavior is well known (both are multimedia benchmarks). Since the execution time is constrained by real-time constraints the execution times are identical, and we compare only the power required.

<table>
<thead>
<tr>
<th>Workload description</th>
<th>Power consumption for the workload (Watts)</th>
<th>Relative consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mobile Pentium III @ 500 MHz, 1.6V</td>
<td>TM 3200 @ 400MHz</td>
</tr>
<tr>
<td>MP3 playback</td>
<td>0.672</td>
<td>0.214</td>
</tr>
<tr>
<td>DVD playback</td>
<td>1.13</td>
<td>0.479</td>
</tr>
</tbody>
</table>

**FIGURE 4.22** The energy performance of the processor and memory interface modules using two multimedia benchmarks is shown for the Mobile Pentium III and the Transmeta 3200. Both these chips are available in more recent versions that have additional power management features.

Although processor power differences can certainly affect battery life, with new processor designed to reduce energy consumption, the processor is often a minor contributor to overall energy usage. Figure shows power measurements for a typical laptop based on a Mobile Pentium III. As you can see small differences in processor power consumption are unlikely to make a large difference in overall power usage.

<table>
<thead>
<tr>
<th>Major system</th>
<th>Component</th>
<th>Power (W)</th>
<th>Percent of total power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Low power Pentium III</td>
<td>0.8</td>
<td>8%</td>
</tr>
<tr>
<td></td>
<td>Processor interface/ memory controller</td>
<td>0.65</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>0.1</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Graphics</td>
<td>0.5</td>
<td>5%</td>
</tr>
<tr>
<td>I/O</td>
<td>Hard drive</td>
<td>0.65</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td>DVD drive</td>
<td>2.51</td>
<td>24%</td>
</tr>
<tr>
<td></td>
<td>Audio</td>
<td>0.5</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Control and other</td>
<td>1.3</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td>TFT display</td>
<td>2.8</td>
<td>27%</td>
</tr>
<tr>
<td>Power</td>
<td>Power supply</td>
<td>0.72</td>
<td>7%</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>10.43</td>
<td>100%</td>
</tr>
</tbody>
</table>

**FIGURE 4.23** Power distribution inside a laptop doing DVD payback shows that the processor subsystem consumes only 20% of the power! The I/O subsystem consumes an astonishing 74% of the power, with the display and DVD drive alone responsible for more than 50% of the total system power. The lesson for laptop users is clear: do not use your disk drive and keep your display off! This data was measured by Intel and is available on their web site.
4.9 Fallacies and Pitfalls

Fallacy: There is a simple approach to multiple issue processors that yields high performance without a significant investment in silicon area or design complexity.

This is a fallacy in the sense that many designers have believed it and committed significant effort to trying to find this “silver bullet” approach. Although it is possible to build relatively simple multiple-issue processors, as issue rates increase the gap between peak and sustained performance grows quickly. This gap has forced designers to explore sophisticated techniques for maintaining performance (dynamic scheduling, hardware and software support for speculation, more flexible issue, sophisticated instruction prefetch, and branch prediction). As Figure 4.24--which includes data on Itanium, Pentium III and 4, and Alpha 21264--shows, the result is uniformly high transistor counts, as well as high power consumption. See if you match the characteristics to the processor without reading the answer in the caption!

<table>
<thead>
<tr>
<th>Issue rate: Total / Memory / Integer / FP / Branch</th>
<th>Max. clock rate available (in mid 2001)</th>
<th>Transistors with / without caches</th>
<th>On-chip caches: 1st level second level</th>
<th>Power (Watts)</th>
<th>SPECbase CPU2000 INT / FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>4/2/4/2/1</td>
<td>1 GHz</td>
<td>15 M / 6 M</td>
<td>64KB + 64KB</td>
<td>107</td>
<td>561 / 643</td>
</tr>
<tr>
<td>3/2/2/1/1</td>
<td>2 GHz</td>
<td>42 M / 23M</td>
<td>12K entries + 8KB 256 KB</td>
<td>67</td>
<td>636 / 648</td>
</tr>
<tr>
<td>3/2/2/1/1</td>
<td>1 GHz</td>
<td>28 M / 9.5 M</td>
<td>16KB + 16KB 256 KB</td>
<td>36</td>
<td>454 / 329</td>
</tr>
<tr>
<td>6/2/2/2/3</td>
<td>0.8 GHz</td>
<td>25 M / 17 M</td>
<td>16K + 16K 96 KB</td>
<td>130</td>
<td>379 / 714</td>
</tr>
</tbody>
</table>

FIGURE 4.24 The key characteristics of four recent multiple issue microprocessors show significant dramatic variety. These vary from a dynamically-scheduled speculative processor to a statically schedule multiple issue processor to a VLIW. They range in die size from just over 100 mm2 to almost 300 mm2 and in power from 26 W to just under 100W, although the integrated circuit processes differ significantly. The SPEC numbers are the highest official numbers reported as of August 2001, and the clock rate of that system is shown. Can you guess what these four processors are?


In addition to the hardware complexity, it has become clear that compiling for processors with significant amounts of ILP has become extremely complex. Not only must the compiler support a wide set of sophisticated transformation, but tuning the compiler to achieve good performance across a wide set of benchmarks appears to be very difficult.
Obtaining good performance is also affected by design decisions at the system level, and such choices can be complex. For example, for the first machine in Figure 4.24 the highest SPECInt number comes from a 1 GHz part, but the highest SPECFP number comes from a system with a 833 MHz part!

The EPIC approach is based on the application of massive resources. These resources include more load-store, computational, and branch units, as well as larger, lower-latency caches than would be required for a superscalar processor. Thus, IA-64 gambles that, in the future, power will not be the critical limitation, and that massive resources, along with the machinery to exploit them, will not penalize performance with their adverse effect on clock speed, path length, or CPI factors.

M. Hopkins [2000], in a commentary on the EPIC approach and the IA-64 architecture

The relative merits of software-intensive and hardware-intensive approaches to exploiting ILP continue to be debated. Over time, it appears that advantageous elements from the “enemy camp” are slowly being incorporated into each approach. Examples include:

<table>
<thead>
<tr>
<th>“Software” techniques in hardware-centric approaches</th>
<th>“Hardware” techniques in software-intensive approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for conditional instructions.</td>
<td>Scoreboard scheduling of instructions.</td>
</tr>
<tr>
<td>Prefetch instructions and other cache “hints”.</td>
<td>Dynamic branch prediction.</td>
</tr>
<tr>
<td>Branch prediction hints.</td>
<td>Rollback or trap-and-fixup support for speculation.</td>
</tr>
<tr>
<td>Special support for speculative (non-excepting) loads.</td>
<td>Hardware for checking speculated load correctness.</td>
</tr>
</tbody>
</table>

Initially, the software-intensive and hardware-intensive approaches were quite different, and the ability to manage the complexity of the hardware-intensive approaches was in doubt. The development of several high performance dynamic speculation processors, which have high clock rates, has eased this concern. The complexity of the IA-64 architecture and the Itanium design has indicated to many designers that it is unlikely that a software-intensive approach will produce processors that are much faster, much smaller (in transistor count or die size), much simpler, or much more power efficient. Similarly, the development of compilers for these processors has proved challenging. Although it is likely that both future compilers for IA-64 and future implementations will be more effective, the IA-64 architecture does not appear to represent a significant breakthrough in
scaling ILP or in avoiding the problems of complexity and power consumption in high performance processors.

As both approaches have proven to have advantages, each has tended to incorporate techniques from the other. It remains unclear whether the two approaches will continue to move toward the middle, or whether a new architectural approach that truly combines the best of each will be developed.

The alternative to trying to continue to push uniprocessors to exploit ILP is to look toward multiprocessors, the topic of Chapter 6. Looking toward multiprocessors to take advantage of parallelism overcomes a fundamental problem in ILP processors: building a cost-effective memory system. A multiprocessor memory system is inherently multiported and, as we will see, can even be distributed in a larger processor.

Using multiprocessors to exploit parallelism encounters two difficulties. First, it is likely that the software model will need to change. Second, multiprocessors may have difficulty in exploiting fine-grained, low-level parallelism. Although it appears clear that using a large number of processors requires new programming approaches, using a smaller number of processors efficiently could be based on compiler or language approaches, or might even be used for multiple independent processes. Exploiting the type of fine-grained parallelism that a compiler can easily uncover can be quite difficult in a multiprocessor, since the processors are relatively far apart. Simultaneous multithreading (see Chapter 6) may be the intermediate step between ILP and true multiprocessing.

In 2000, IBM announced the first commercial single-chip general-purpose multiprocessor, the IBM Power4, which contains two Power3 processors and an integrated second level cache, for a total transistor count of 174 million! Because the Power4 chip also contains a memory interface, a third-level cache interface, and a direct multiprocessor interconnect, IBM used the Power4 to build an 8-processor module using four Power4 chips. The module has a total size of about 64 in² and is capable of a peak performance of 32 billion floating point operations per second! The challenge for multiprocessors appears to be the same as for ILP-intensive uniprocessors: translate this enormous peak performance into delivered performance on real applications. In the case of the IBM design, the intended market is large-scale servers, where the available application parallelism may make a multiprocessor attractive.

The embedded world actually delivered multiple processors on a die first! The TI TMS320C80 provides four DSPs and a RISC processor, which acts as controller, on a single die. Likewise, several embedded versions of MIPS processors use multiple processors per die. The obvious parallelism in embedded applications and the lack of stringent software compatibility requirements may allow the embedded world to embrace on-chip multiprocessing faster than the desktop environment. We will return to this discussion in Chapter 6.
4.11 Historical Perspective and References

This section describes the historical development of multiple issue approaches, which began with static multiple issue and proceeds to the most recent work leading to IA-64. Similarly, we look at the long history of compiler technology in this area.

The Development of Multiple-Issue Processors

Most of the early multiple-issue processors followed a LIW or VLIW design approach. Charlesworth [1981] reports on the Floating Point Systems AP-120B, one of the first wide-instruction processors containing multiple operations per instruction. Floating Point Systems applied the concept of software pipelining in both a compiler and by hand-writing assembly language libraries to use the processor efficiently. Since the processor was an attached processor, many of the difficulties of implementing multiple issue in general-purpose processors, for example, virtual memory and exception handling, could be ignored.

The Stanford MIPS processor had the ability to place two operations in a single instruction, though this capability was dropped in commercial variants of the architecture, primarily for performance reasons. Along with his colleagues at Yale, Fisher [1983] proposed creating a processor with a very wide instruction (512 bits), and named this type of processor a VLIW. Code was generated for the processor using trace scheduling, which Fisher [1981] had developed originally for generating horizontal microcode. The implementation of trace scheduling for the Yale processor is described by Fisher et al. [1984] and by Ellis [1986]. The Multiflow processor (see Colwell et al. [1987]) was based on the concepts developed at Yale, although many important refinements were made to increase the practicality of the approach. Among these was a controllable store buffer that provided support for a form of speculation. Although more than 100 Multiflow processors were sold, a variety of problems, including the difficulties of introducing a new instruction set from a small company and the competition provided from commercial RISC microprocessors that changed the economics in the mini-computer market, led to failure of Multiflow as a company.

Around the same time as Multiflow, Cydrome was founded to build a VLIW-style processor (see Rau et al. [1989]), which was also unsuccessful commercially. Dehnert, Hsu, and Bratt [1989] explain the architecture and performance of the Cydrome Cydra 5, a processor with a wide-instruction word that provides dynamic register renaming and additional support for software pipelining. The Cydra 5 is a unique blend of hardware and software, including conditional instructions and register rotation, aimed at extracting ILP. Cydrome relied on more hardware than the Multiflow processor and achieved competitive performance primarily on vector-style codes. In the end, Cydrome suffered from problems
similar to those of Multiflow and was not a commercial success. Both Multiflow and Cydrome, though unsuccessful as commercial entities, produced a number of people with extensive experience in exploiting ILP as well as advanced compiler technology; many of those people have gone on to incorporate their experience and the pieces of the technology in newer processors. Fisher and Rau [1993] edited a comprehensive collection of papers covering the hardware and software of these two important processors.

Rau had also developed a scheduling technique called polycyclic scheduling, which is a basis for most software pipelining schemes (see Rau, Glaeser, and Picard [1982]). Rau’s work built on earlier work by Davidson and his colleagues on the design of optimal hardware schedulers for pipelined processors. Other historical LIW processors have included the Apollo DN 10000 and the Intel i860, both of which could dual issue FP and integer operations.

One of the interesting approaches used in early VLIW processors, such as the AP-120B and i860, was the idea of a pipeline organization that requires operations to be “pushed through” a functional unit and the results to be caught at the end of the pipeline. In such processors, operations advance only when another operation pushes them from behind (in sequence). Furthermore, an instruction specifies the destination for an instruction issued earlier that will be pushed out of the pipeline when this new operation is pushed in. Such an approach has the advantage that it does not specify a result destination when an operation first issues but only when the result register is actually written. This separation eliminates the need to detect WAW and WAR hazards in the hardware. The disadvantage is that it increases code size since no-ops may be needed to push results out when there is a dependence on an operation that is still in the pipeline and no other operations of that type are immediately needed. Instead of the “push-and-catch” approach used in these two processors, almost all designers have chosen to use self-draining pipelines that specify the destination in the issuing instruction and in which an issued instruction will complete without further action. The advantages in code density and simplifications in code generation seem to outweigh the advantages of the more unusual structure.

Compiler Technology and Hardware-Support for Scheduling

Loop-level parallelism and dependence analysis was developed primarily by D. Kuck and his colleagues at the University of Illinois in the 1970s. They also coined the commonly used terminology of antidependence and output dependence and developed several standard dependence tests, including the GCD and Banerjee tests. The latter test was named after Uptal Banerjee and comes in a variety of flavors. Recent work on dependence analysis has focused on using a variety of exact tests ending with an algorithm called Fourier-Motzkin, which is a linear programming algorithm. D. Maydan and W. Pugh both showed that the sequences of exact tests were a practical solution.
In the area of uncovering and scheduling ILP, much of the early work was connected to the development of VLIW processors, described earlier. Lam [1988] developed algorithms for software pipelining and evaluated their use on Warp, a wide-instruction-word processor designed for special-purpose applications. Weiss and J. E. Smith [1987] compare software pipelining versus loop unrolling as techniques for scheduling code on a pipelined processor. Rau [1994] developed modulo scheduling to deal with the issues of software pipelining loops and simultaneously handling register allocation.

Support for speculative code scheduling was explored in a variety of contexts, including several processors that provided a mode in which exceptions were ignored, allowing more aggressive scheduling of loads (e.g., the MIPS TFP processor, see [Hsu 1994]). Several groups explored ideas for more aggressive hardware support for speculative code scheduling. For example, Smith, Horowitz, and Lam [1992] created a concept called boosting that contains a hardware facility for supporting speculation but provides a checking and recovery mechanism, similar to those in IA-64 and Crusoe. The sentinel scheduling idea, which is also similar to the speculate-and-check approach used in both Crusoe and the IA-64 architectures, was developed jointly by researchers at U. of Illinois and HP Laboratories (see [Mahlke et al. 1992]).

In the early 1990s, Wen-Mei Hwu and his colleagues at the University of Illinois developed a compiler framework, called IMPACT (see [Chang, et. al. 1991]), for exploring the interaction between multiple-issue architectures and compiler technology. This project led to several important ideas, including: superblock scheduling (see [Hwu et. al. 1993]), extensive use of profiling for guiding a variety of optimizations (e.g., procedure inlining), and the use of a special buffer (similar to the ALAT or program-controlled store buffer) for compile-aided memory conflict detection (see [Gallagher, et. al. 1994]). They also explored the performance trade-offs between partial and full support for predication in [Mahlke, et. al. 1995].

The early RISC processors all had delayed branches, a scheme inspired from microprogramming, and several studies on compile-time branch prediction were inspired by delayed branch mechanisms. McFarling and Hennessy [1986] did a quantitative comparison of a variety of compile-time and runtime branch prediction schemes. Fisher and Freudenberger [1992] evaluated a range of compile-time branch prediction schemes using the metric of distance between mispredictions.

EPIC and the IA-64 Development

The roots of the EPIC approach lie in earlier attempts to build LIW and VLIW machines—especially those at Cydrome and Multiflow—and in a long history of compiler work that continued after these companies failed at HP, the University of Illinois, and elsewhere. Insights gained from that work led designers at HP to propose a VLIW-style, 64-bit architecture to follow on to the HP-PA RISC archi-
Intel was looking for a new architecture to replace the x86 (now called IA-32) architecture and to provide 64-bit capability. In 1995, they formed a partnership to design a new architecture, IA-64, and build processors based on it. Itanium is the first such processor. A description of the IA-64 architecture is available online at: http://devresource.hp.com/devresource/Docs/Refs/IA64ISA/. A description of the highlights of the Itanium processor is available at: http://www.intel.com/design/itanium/microarch_ovw/index.htm.

References


HWU, W. W., MAHLKE, S. A., CHEN, W. Y., CHANG, P. P., WARTER, N. J., BRINGMANN, R. A.,
4.11 Historical Perspective and References


E X E R C I S E S

Most of these exercises are still good. What we need are exercises that explore the concepts of predication and speculative scheduling.

New addition: consider the simple loop in Section 4.1. Assume the number of iterations is unknown, but large. Find the theoretically optimal number of unrollings using the timing of the loop in Section 4.1

Hint: recall that you will need two loops: one unrolled and one not!

4.1 [15] <4.1> List all the dependences (output, anti, and true) in the following code fragment. Indicate whether the true dependences are loop-carried or not. Show why the loop is not parallel.

```c
for (i=2;i<100;i=i+1) {
```
a[i] = b[i] + a[i]; /* S1 */
c[i-1] = a[i] + d[i]; /* S2 */
a[i-1] = 2 * b[i]; /* S3 */
b[i+1] = 2 * b[i]; /* S4 */

4.2 [15] <4.1> Here is an unusual loop. First, list the dependences and then rewrite the loop so that it is parallel.

for (i=1; i<100; i=i+1) {
    a[i] = b[i] + c[i]; /* S1 */
    b[i] = a[i] + d[i]; /* S2 */
    a[i+1] = a[i] + e[i]; /* S3 */
}

4.3 [15] <4.1> Assuming the pipeline latencies from Figure 4.1, unroll the following loop as many times as necessary to schedule it without any delays, collapsing the loop overhead instructions. Assume a one-cycle delayed branch. Show the schedule. The loop computes Y[i] = a × X[i] + Y[i], the key step in a Gaussian elimination.

loop: L.D F0,0(R1)
      MUL.D F0,F0,F2
      L.D F4,0(R2)
      ADD.D F0,F0,F4
      S.D F0,0(R2)
      DADDUI R1,R1,#-8
      DADDUI R1,R1,#-8
      BNE R1,R3,loop

4.4 [15] <4.1> Assume the pipeline latencies from Figure 4.1 and a one-cycle delayed branch. Unroll the following loop a sufficient number of times to schedule it without any delays. Show the schedule after eliminating any redundant overhead instructions. The loop is a dot product (assuming F2 is initially 0) and contains a recurrence. Despite the fact that the loop is not parallel, it can be scheduled with no delays.

loop: L.D F0,0(R1)
      L.D F4,0(R2)
      MUL.D F0,F0,F4
      ADD.D F2,F0,F2
      DADDUI R1,R1,#-8
      DADDUI R1,R1,#-8
      BNE R1,R3,loop

4.5 [20/22/22/22/25/25/25/20/22/22] <4.1,4.2,4.3> In this Exercise, we will look at how a common vector loop runs on a variety of pipelined versions of MIPS. The loop is the so-called SAXPY loop (discussed extensively in Appendix B) and the central operation in Gaussian elimination. The loop implements the vector operation Y = a × X + Y for a vector of length 100. Here is the MIPS code for the loop:

foo: L.D F2,0(R1) ; load X(i)
     MUL.D F4,F2,F0 ; multiply a*X(i)
     L.D F6,0(R2) ; load Y(i)
     ADD.D F6,F4,F6 ; add a*X(i) + Y(i)
     S.D F6,0(R2) ; store Y(i)
     ADDI R1,R1,#8 ; increment X index
     ADDI R2,R2,#8 ; increment Y index
     SGTI R3,R1,done ; test if done
     BEQZ R3,foo ; loop if not done
For (a)–(e), assume that the integer operations issue and complete in one clock cycle (including loads) and that their results are fully bypassed. Ignore the branch delay. You will use the FP latencies shown in Figure 4.1 on page 222. Assume that the FP unit is fully pipelined.

a. [20] <4.1> For this problem use the standard single-issue MIPS pipeline with the pipeline latencies from Figure 4.1. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) on the first iteration of the loop. How many clock cycles does each loop iteration take?

b. [22] <4.1> Unroll the MIPS code for SAXPY to make four copies of the body and schedule it for the standard MIPS integer pipeline and a fully pipelined FPU with the FP latencies of Figure 4.1 on page 222. When unwinding, you should optimize the code as we did in section 4.1. Significant reordering of the code will be needed to maximize performance.

c. [25] <4.1,4.3> Assume a superscalar architecture that can issue any two independent operations in a clock cycle (including two integer operations). Unwind the MIPS code for SAXPY to make four copies of the body and schedule it assuming the FP latencies of Figure 4.1 on page 222. Assume one fully pipelined copy of each functional unit (e.g., FP adder, FP multiplier) and two integer functional units with latency to use of 0. How many clock cycles will each iteration on the original code take? When unwinding, you should optimize the code as in section 4.1. What is the speedup versus the original code?

d. [25] <4.3> To further boost clock rates, a number of processors have added additional pipelining to units traditionally assigned only one stage in the pipeline, resulting in pipelines with depths of 10-14 cycles for the integer pipeline. Suppose we designed a deeply pipelined MIPS processor that had twice the clock rate of our standard MIPS pipeline and could issue any two unrelated instructions in the same time that the normal MIPS pipeline issued one operation. If the second instruction is dependent on the first, only the first will issue. Unroll the MIPS SAXPY code to make four copies of the loop body and schedule it for this deeply pipelined processor, assuming the FP latencies of Figure 4.1 on page 222. Also assume the load to use latency is 1 cycle, but other integer unit latencies are 0 cycles. How many clock cycles does each loop iteration take? Remember that these clock cycles are half as long as those on a standard MIPS pipeline or a superscalar MIPS.

e. [20] <4.3> Start with the SAXPY code and the processor used in Figure 4.5. Unroll the SAXPY loop to make four copies of the body, performing simple optimizations (as in section 4.1). Assume all integer unit latencies are 0 cycles and the FP latencies are given in Figure 4.1. Fill in a table like Figure 4.28 for the unrolled loop. How many clock cycles does each loop iteration take?

4.6 [15] <4.4> Here is a simple code fragment:

```
for (i=2;i<=100;i+=2)
    a[i] = a[50*i+1];
```

To use the GCD test, this loop must first be “normalized”—written so that the index starts at 1 and increments by 1 on every iteration. Write a normalized version of the loop (change the indices as needed), then use the GCD test to see if there is a dependence.
4.7 [15] <4.1.4.4> Here is another loop:

\[
\text{for } (i=2, i<=100; i+=2) \\
\text{ } \\
a[i] = a[i-1];
\]

Normalize the loop and use the GCD test to detect a dependence. Is there a loop-carried, true dependence in this loop?

4.8 [25] <4.4> Show that if for two array elements \(A(a \times i + b)\) and \(A(c \times i + d)\) there is a true dependence, then \(\text{GCD}(c,a)\) divides \((d - b)\).

4.9 [15] <4.4> Rewrite the software pipelining loop shown in the Example on page 249, so that it can be run by simply decrementing \(R1\) by 16 before the loop starts. After rewriting the loop, show the start-up and finish-up code. \textit{Hint:} To get the loop to run properly when \(R1\) is decremented, the \(S.D\) should store the result of the original first iteration. You can achieve this by adjusting load-store offsets.

4.10 [20] <4.4> Consider the loop that we software pipelined on page 249. Suppose the latency of the \texttt{ADD.D} was five cycles. The software pipelined loop now has a stall. Show how this loop can be written using both software pipelining and loop unrolling to eliminate any stalls. The loop should be unrolled as few times as possible (once is enough). You need not show loop start-up or clean-up.

4.11 [15] <4.5> Perform the same transformation (moving up the branch) as the example on page 262, but using only conditional move. Be careful that your loads, which are no longer control dependent, cannot raise an exception if they should not have been executed!

4.12 [Discussion] <4.3-4.5> Discuss the advantages and disadvantages of a superscalar implementation and a VLIW approach in the context of MIPS. What levels of ILP favor each approach? What other concerns would you consider in choosing which type of processor to build? How does speculation affect the results?