
2.1	Instruction-Level Parallelism: Concepts and Challenges	66
2.2	Basic Compiler Techniques for Exposing ILP	74
2.3	Reducing Branch Costs with Prediction	80
2.4	Overcoming Data Hazards with Dynamic Scheduling	89
2.5	Dynamic Scheduling: Examples and the Algorithm	97
2.6	Hardware-Based Speculation	104
2.7	Exploiting ILP Using Multiple Issue and Static Scheduling	114
2.8	Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation	118
2.9	Advanced Techniques for Instruction Delivery and Speculation	121
2.10	Putting It All Together: The Intel Pentium 4	131
2.11	Fallacies and Pitfalls	138
2.12	Concluding Remarks	140
2.13	Historical Perspective and References	141
	Case Studies with Exercises by Robert P. Colwell	142

2

Instruction-Level Parallelism and Its Exploitation

"Who's first?"

"America."

"Who's second?"

"Sir, there is no second."

Dialog between two observers of the sailing race later named "The America's Cup" and run every few years—the inspiration for John Cocke's naming of the IBM research processor as "America." This processor was the precursor to the RS/6000 series and the first superscalar microprocessor.

2.1

Instruction-Level Parallelism: Concepts and Challenges

All processors since about 1985 use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called *instruction-level parallelism* (ILP), since the instructions can be evaluated in parallel. In this chapter and Appendix G, we look at a wide range of techniques for extending the basic pipelining concepts by increasing the amount of parallelism exploited among instructions.

This chapter is at a considerably more advanced level than the material on basic pipelining in Appendix A. If you are not familiar with the ideas in Appendix A, you should review that appendix before venturing into this chapter.

We start this chapter by looking at the limitation imposed by data and control hazards and then turn to the topic of increasing the ability of the compiler and the processor to exploit parallelism. These sections introduce a large number of concepts, which we build on throughout this chapter and the next. While some of the more basic material in this chapter could be understood without all of the ideas in the first two sections, this basic material is important to later sections of this chapter as well as to Chapter 3.

There are two largely separable approaches to exploiting ILP: an approach that relies on hardware to help discover and exploit the parallelism dynamically, and an approach that relies on software technology to find parallelism, statically at compile time. Processors using the dynamic, hardware-based approach, including the Intel Pentium series, dominate in the market; those using the static approach, including the Intel Itanium, have more limited uses in scientific or application-specific environments.

In the past few years, many of the techniques developed for one approach have been exploited within a design relying primarily on the other. This chapter introduces the basic concepts and both approaches. The next chapter focuses on the critical issue of limitations on exploiting ILP.

In this section, we discuss features of both programs and processors that limit the amount of parallelism that can be exploited among instructions, as well as the critical mapping between program structure and hardware structure, which is key to understanding whether a program property will actually limit performance and under what circumstances.

The value of the CPI (cycles per instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

$$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}$$

The *ideal pipeline CPI* is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right-hand side, we minimize the overall pipeline CPI or, alternatively, increase the IPC (instructions per clock). The equation above allows us to characterize various techniques by what component of the overall CPI a technique reduces. Figure 2.1 shows the

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	A.2
Delayed branches and simple branch scheduling	Control hazard stalls	A.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	A.7
Dynamic scheduling with renaming	Data hazard stalls and stalls from antidependences and output dependences	2.4
Branch prediction	Control stalls	2.3
Issuing multiple instructions per cycle	Ideal CPI	2.7, 2.8
Hardware speculation	Data hazard and control hazard stalls	2.6
Dynamic memory disambiguation	Data hazard stalls with memory	2.4, 2.6
Loop unrolling	Control hazard stalls	2.2
Basic compiler pipeline scheduling	Data hazard stalls	A.2, 2.2
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	G.2, G.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	G.4, G.5

Figure 2.1 The major techniques examined in Appendix A, Chapter 2, or Appendix G are shown together with the component of the CPI equation that the technique affects.

techniques we examine in this chapter and in Appendix G, as well as the topics covered in the introductory material in Appendix A. In this chapter we will see that the techniques we introduce to decrease the ideal pipeline CPI can increase the importance of dealing with hazards.

What Is Instruction-Level Parallelism?

All the techniques in this chapter exploit parallelism among instructions. The amount of parallelism available within a *basic block*—a straight-line code sequence with no branches in except to the entry and no branches out except at the exit—is quite small. For typical MIPS programs, the average dynamic branch frequency is often between 15% and 25%, meaning that between three and six instructions execute between a pair of branches. Since these instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be less than the average basic block size. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is often called *loop-level parallelism*. Here is a simple example of a loop, which adds two 1000-element arrays, that is completely parallel:


```

for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];

```

Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.

There are a number of techniques we will examine for converting such loop-level parallelism into instruction-level parallelism. Basically, such techniques work by unrolling the loop either statically by the compiler (as in the next section) or dynamically by the hardware (as in Sections 2.5 and 2.6).

An important alternative method for exploiting loop-level parallelism is the use of vector instructions (see Appendix F). A vector instruction exploits data-level parallelism by operating on data items in parallel. For example, the above code sequence could execute in four instructions on some vector processors: two instructions to load the vectors *x* and *y* from memory, one instruction to add the two vectors, and an instruction to store back the result vector. Of course, these instructions would be pipelined and have relatively long latencies, but these latencies may be overlapped.

Although the development of the vector ideas preceded many of the techniques for exploiting ILP, processors that exploit ILP have almost completely replaced vector-based processors in the general-purpose processor market. Vector instruction sets, however, have seen a renaissance, at least for use in graphics, digital signal processing, and multimedia applications.

Data Dependences and Hazards

Determining how one instruction depends on another is critical to determining how much parallelism exists in a program and how that parallelism can be exploited. In particular, to exploit instruction-level parallelism we must determine which instructions can be executed in parallel. If two instructions are *parallel*, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls, assuming the pipeline has sufficient resources (and hence no structural hazards exist). If two instructions are dependent, they are not parallel and must be executed in order, although they may often be partially overlapped. The key in both cases is to determine whether an instruction is dependent on another instruction.

Data Dependences

There are three different types of dependences: *data dependences* (also called true data dependences), *name dependences*, and *control dependences*. An instruction *j* is *data dependent* on instruction *i* if either of the following holds:

- instruction *i* produces a result that may be used by instruction *j*, or
- instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i*.

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program. Note that a dependence within a single instruction (such as `ADDD R1,R1,R1`) is not considered a dependence.

For example, consider the following MIPS code sequence that increments a vector of values in memory (starting at `0(R1)`, and with the last element at `8(R2)`), by a scalar in register `F2`. (For simplicity, throughout this chapter, our examples ignore the effects of delayed branches.)

```

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
           BNE    R1,R2,LOOP   ;branch R1!=R2

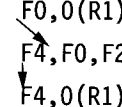
```

The data dependences in this code sequence involve both floating-point data:

```

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

```




and integer data:

```

DADDIU R1,R1,-8    ;decrement pointer
              ↓      ;8 bytes (per DW)
BNE     R1,R2,Loop  ;branch R1!=R2

```



Both of the above dependent sequences, as shown by the arrows, have each instruction depending on the previous one. The arrows here and in following examples show the order that must be preserved for correct execution. The arrow points from an instruction that must precede the instruction that the arrowhead points to.

If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions. (See Appendix A for a brief description of data hazards, which we will define precisely in a few pages.) Executing the instructions simultaneously will cause a processor with pipeline interlocks (and a pipeline depth longer than the distance between the instructions in cycles) to detect a hazard and stall, thereby reducing or eliminating the overlap. In a processor without interlocks that relies on compiler scheduling, the compiler cannot schedule dependent instructions in such a way that they completely overlap, since the program will not execute correctly. The presence of a data

dependence in an instruction sequence reflects a data dependence in the source code from which the instruction sequence was generated. The effect of the original data dependence must be preserved.

Dependences are a property of *programs*. Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the *pipeline organization*. This difference is critical to understanding how instruction-level parallelism can be exploited.

A data dependence conveys three things: (1) the possibility of a hazard, (2) the order in which results must be calculated, and (3) an upper bound on how much parallelism can possibly be exploited. Such limits are explored in Chapter 3.

Since a data dependence can limit the amount of instruction-level parallelism we can exploit, a major focus of this chapter is overcoming these limitations. A dependence can be overcome in two different ways: maintaining the dependence but avoiding a hazard, and eliminating a dependence by transforming the code. Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware.

A data value may flow between instructions either through registers or through memory locations. When the data flow occurs in a register, detecting the dependence is straightforward since the register names are fixed in the instructions, although it gets more complicated when branches intervene and correctness concerns force a compiler or hardware to be conservative.

Dependences that flow through memory locations are more difficult to detect, since two addresses may refer to the same location but look different: For example, 100(R4) and 20(R6) may be identical memory addresses. In addition, the effective address of a load or store may change from one execution of the instruction to another (so that 20(R4) and 20(R4) may be different), further complicating the detection of a dependence.

In this chapter, we examine hardware for detecting data dependences that involve memory locations, but we will see that these techniques also have limitations. The compiler techniques for detecting such dependences are critical in uncovering loop-level parallelism, as we will see in Appendix G.

Name Dependences

The second type of dependence is a *name dependence*. A name dependence occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name. There are two types of name dependences between an instruction *i* that *precedes* instruction *j* in program order:

1. An *antidependence* between instruction *i* and instruction *j* occurs when instruction *j* writes a register or memory location that instruction *i* reads. The original ordering must be preserved to ensure that *i* reads the correct value. In the example on page 69, there is an antidependence between S.D and DADDIU on register R1.

2. An *output dependence* occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j .

Both antidependences and output dependences are name dependences, as opposed to true data dependences, since there is no value being transmitted between the instructions. Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

This renaming can be more easily done for register operands, where it is called *register renaming*. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

Data Hazards

A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence. Because of the dependence, we must preserve what is called *program order*, that is, the order that the instructions would execute in if executed sequentially one at a time as determined by the original source program. The goal of both our software and hardware techniques is to exploit parallelism by preserving program order *only where it affects the outcome of the program*. Detecting and avoiding hazards ensures that necessary program order is preserved.

Data hazards, which are informally described in Appendix A, may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions i and j , with i preceding j in program order. The possible data hazards are

- **RAW (read after write)**— j tries to read a source before i writes it, so j incorrectly gets the *old* value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that j receives the value from i .
- **WAW (write after write)**— j tries to write an operand before it is written by i . The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

- **WAR (write after read)**— j tries to write a destination before it is read by i , so i incorrectly gets the *new* value. This hazard arises from an antidependence. WAR hazards cannot occur in most static issue pipelines—even deeper pipelines or floating-point pipelines—because all reads are early (in ID) and all writes are late (in WB). (See Appendix A to convince yourself.) A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline *and* other instructions that read a source late in the pipeline, or when instructions are reordered, as we will see in this chapter.

Note that the RAR (*read after read*) case is not a hazard.

Control Dependences

The last type of dependence is a *control dependence*. A control dependence determines the ordering of an instruction, i , with respect to a branch instruction so that the instruction i is executed in correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the “then” part of an if statement on the branch. For example, in the code segment

```
if p1 {
    S1;
};
if p2 {
    S2;
}
```

$S1$ is control dependent on $p1$, and $S2$ is control dependent on $p2$ but not on $p1$.

In general, there are two constraints imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved *before* the branch so that its execution *is no longer controlled* by the branch. For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.
2. An instruction that is not control dependent on a branch cannot be moved *after* the branch so that its execution *is controlled* by the branch. For example, we cannot take a statement before the if statement and move it into the then portion.

When processors preserve strict program order, they ensure that control dependences are also preserved. We may be willing to execute instructions that should not have been executed, however, thereby violating the control dependences, *if* we can do so without affecting the correctness of the program. Control dependence is not the critical property that must be preserved. Instead, the

two properties critical to program correctness—and normally preserved by maintaining both data and control dependence—are the *exception behavior* and the *data flow*.

Preserving the exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program. Often this is relaxed to mean that the reordering of instruction execution must not cause any new exceptions in the program. A simple example shows how maintaining the control and data dependences can prevent such situations. Consider this code sequence:

```

                                DADDU    R2,R3,R4
                                BEQZ     R2,L1
                                LW        R1,0(R2)
L1:

```

In this case, it is easy to see that if we do not maintain the data dependence involving R2, we can change the result of the program. Less obvious is the fact that if we ignore the control dependence and move the load instruction before the branch, the load instruction may cause a memory protection exception. Notice that *no data dependence* prevents us from interchanging the BEQZ and the LW; it is only the control dependence. To allow us to reorder these instructions (and still preserve the data dependence), we would like to just ignore the exception when the branch is taken. In Section 2.6, we will look at a hardware technique, *speculation*, which allows us to overcome this exception problem. Appendix G looks at software techniques for supporting speculation.

The second property preserved by maintenance of data dependences and control dependences is the data flow. The *data flow* is the actual flow of data values among instructions that produce results and those that consume them. Branches make the data flow dynamic, since they allow the source of data for a given instruction to come from many points. Put another way, it is insufficient to just maintain data dependences because an instruction may be data dependent on more than one predecessor. Program order is what determines which predecessor will actually deliver a data value to an instruction. Program order is ensured by maintaining the control dependences.

For example, consider the following code fragment:

```

                                DADDU    R1,R2,R3
                                BEQZ     R4,L
                                DSUBU    R1,R5,R6
L:                                ...
                                OR        R7,R1,R8

```

In this example, the value of R1 used by the OR instruction depends on whether the branch is taken or not. Data dependence alone is not sufficient to preserve correctness. The OR instruction is data dependent on both the DADDU and DSUBU instructions, but preserving that order alone is insufficient for correct execution.

Instead, when the instructions execute, the data flow must be preserved: If the branch is not taken, then the value of R1 computed by the DSUBU should be used by the OR, and if the branch is taken, the value of R1 computed by the DADDU should be used by the OR. By preserving the control dependence of the OR on the branch, we prevent an illegal change to the data flow. For similar reasons, the DSUBU instruction cannot be moved above the branch. Speculation, which helps with the exception problem, will also allow us to lessen the impact of the control dependence while still maintaining the data flow, as we will see in Section 2.6.

Sometimes we can determine that violating the control dependence cannot affect either the exception behavior or the data flow. Consider the following code sequence:

	DADDU	R1, R2, R3
	BEQZ	R12, skip
	DSUBU	R4, R5, R6
	DADDU	R5, R4, R9
skip:	OR	R7, R8, R9

Suppose we knew that the register destination of the DSUBU instruction (R4) was unused after the instruction labeled skip. (The property of whether a value will be used by an upcoming instruction is called *liveness*.) If R4 were unused, then changing the value of R4 just before the branch would not affect the data flow since R4 would be *dead* (rather than live) in the code region after skip. Thus, if R4 were dead and the existing DSUBU instruction could not generate an exception (other than those from which the processor resumes the same process), we could move the DSUBU instruction before the branch, since the data flow cannot be affected by this change.

If the branch is taken, the DSUBU instruction will execute and will be useless, but it will not affect the program results. This type of code scheduling is also a form of speculation, often called software speculation, since the compiler is betting on the branch outcome; in this case, the bet is that the branch is usually not taken. More ambitious compiler speculation mechanisms are discussed in Appendix G. Normally, it will be clear when we say speculation or speculative whether the mechanism is a hardware or software mechanism; when it is not clear, it is best to say “hardware speculation” or “software speculation.”

Control dependence is preserved by implementing control hazard detection that causes control stalls. Control stalls can be eliminated or reduced by a variety of hardware and software techniques, which we examine in Section 2.3.

2.2

Basic Compiler Techniques for Exposing ILP

This section examines the use of simple compiler technology to enhance a processor’s ability to exploit ILP. These techniques are crucial for processors that use static issue or static scheduling. Armed with this compiler technology, we will shortly examine the design and performance of processors using static issu-

ing. Appendix G will investigate more sophisticated compiler and associated hardware schemes designed to enable a processor to exploit more instruction-level parallelism.

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. Figure 2.2 shows the FP unit latencies we assume in this chapter, unless different latencies are explicitly stated. We assume the standard five-stage integer pipeline, so that branches have a delay of 1 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.

In this subsection, we look at how the compiler can increase the amount of available ILP by transforming loops. This example serves both to illustrate an important technique as well as to motivate the more powerful program transformations described in Appendix G. We will rely on the following code segment, which adds 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 in Appendix G 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.

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Figure 2.2 Latencies of FP operations used in this chapter. 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 0, 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.

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 address of 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!=R2

```

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 2.2.

Example Show how the loop would look on MIPS, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for delays from floating-point operations, but remember that we are ignoring delayed branches.

Answer Without any scheduling, the loop will execute as follows, taking 9 cycles:

		<u>Clock cycle issued</u>
Loop:	L.D F0,0(R1)	1
	<i>stall</i>	2
	ADD.D F4,F0,F2	3
	<i>stall</i>	4
	<i>stall</i>	5
	S.D F4,0(R1)	6
	DADDUI R1,R1,#-8	7
	<i>stall</i>	8
	BNE R1,R2,Loop	9

We can schedule the loop to obtain only two stalls and reduce the time to 7 cycles:

```

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

```

The stalls after ADD.D are for use by the S.D.

In the previous example, we complete one loop iteration and store back one array element every 7 clock cycles, but the actual work of operating on the array element takes just 3 (the load, add, and store) of those 7 clock cycles. The remaining 4 clock cycles consist of loop overhead—the DADDUI and BNE—and two stalls. To eliminate these 4 clock cycles we need to get more operations 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 stalls 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 number of registers.

Example Show our loop unrolled so that there are four copies of the loop body, assuming $R1 - R2$ (that is, the size of the array) 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.

```

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. Symbolic substitution and simplification

will rearrange expressions so as to allow constants to be collapsed, allowing an expression such as “ $((i + 1) + 1)$ ” to be rewritten as “ $(i + (1 + 1))$ ” and then simplified to “ $(i + 2)$.” We will see more general forms of these optimizations that eliminate dependent computations in Appendix G.

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 27 clock cycles—each LD has 1 stall, each ADDD 2, the DADDUI 1, plus 14 instruction issue cycles—or 6.75 clock cycles for each of the four elements, but it can be scheduled to improve performance significantly. 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 $(n \bmod k)$ 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 previous 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 2.2.

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
ADD.D	F8,F6,F2
ADD.D	F12,F10,F2
ADD.D	F16,F14,F2
S.D	F4,0(R1)
S.D	F8,-8(R1)
DADDUI	R1,R1,#-32
S.D	F12,16(R1)
S.D	F16,8(R1)
BNE	R1,R2,Loop

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 9 cycles per element before any unrolling or scheduling and 7 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

Throughout this chapter and Appendix G, 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:

- Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
- Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations.
- Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
- 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.
- 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 one instruction depends on another and how the instructions can be changed or reordered given the dependences.

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 14 clock cycles, only 2 cycles were loop overhead: the DADDUI, 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.

A second limit to unrolling is the growth in code size that results. For larger loops, the code size growth may be a concern particularly if it causes an increase in the instruction cache miss rate.

Another factor often more important than code size is the potential shortfall in registers that is created by aggressive unrolling and scheduling. This secondary effect 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 may not be possible 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 processors 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 we have examined so far to the multiple-issue superscalars and VLIWs explored later in this chapter.

2.3

Reducing Branch Costs with Prediction

Because of the need to enforce control dependences through branch hazards and stalls, branches will hurt pipeline performance. Loop unrolling is one way to reduce the number of branch hazards; we can also reduce the performance losses of branches by predicting how they will behave.

The behavior of branches can be predicted both statically at compile time and dynamically by the hardware at execution time. 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.

Static Branch Prediction

In Appendix A, we discuss an architectural feature that supports static branch prediction, namely, delayed branches. Being able to accurately predict a branch at compile time is also helpful for scheduling data hazards. Loop unrolling is another example of a technique for improving code scheduling that depends on predicting branches.

To reorder code around branches so that it runs faster, 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 for the SPEC programs ranges from not very accurate (59%) to highly accurate (9%).

A 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 2.3 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.

The effectiveness of any branch prediction scheme depends both on the accuracy of the scheme and the frequency of conditional branches, which vary in SPEC from 3% to 24%. The fact that the misprediction rate for the integer programs is higher and that such programs typically have a higher branch frequency is a major limitation for static branch prediction. In the next section, we consider dynamic branch predictors, which most recent processors have employed.

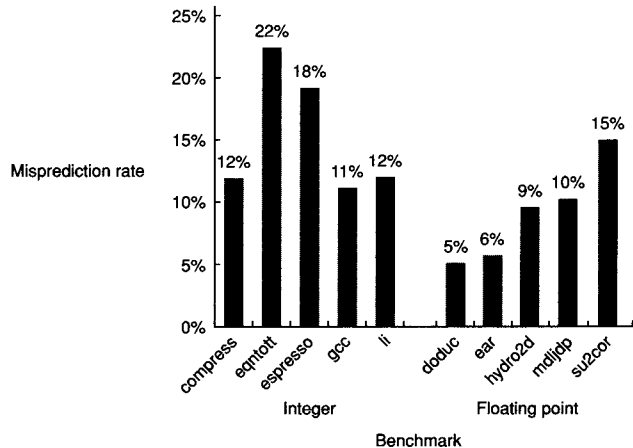


Figure 2.3 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which vary from 3% to 24%.

Dynamic Branch Prediction and Branch-Prediction Buffers

The simplest dynamic branch-prediction scheme is a *branch-prediction buffer* or *branch history table*. A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. This scheme is the simplest sort of buffer; it has no tags and is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs.

With such a buffer, we don't know, in fact, if the prediction is correct—it may have been put there by another branch that has the same low-order address bits. But this doesn't matter. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.

This buffer is effectively a cache where every access is a hit, and, as we will see, the performance of the buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches. Before we analyze the performance, it is useful to make a small, but important, improvement in the accuracy of the branch-prediction scheme.

This simple 1-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, since the misprediction causes the prediction bit to be flipped.

To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must miss twice before it is changed. Figure 2.4 shows the finite-state processor for a 2-bit prediction scheme.

A branch-prediction buffer can be implemented as a small, special “cache” accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue. As Figure 2.4 shows, if the prediction turns out to be wrong, the prediction bits are changed.

What kind of accuracy can be expected from a branch-prediction buffer using 2 bits per entry on real applications? Figure 2.5 shows that for the SPEC89 benchmarks a branch-prediction buffer with 4096 entries results in a prediction accuracy ranging from over 99% to 82%, or a *misprediction rate* of 1% to 18%. A 4K entry buffer, like that used for these results, is considered small by 2005 standards, and a larger buffer could produce somewhat better results.

As we try to exploit more ILP, the accuracy of our branch prediction becomes critical. As we can see in Figure 2.5, the accuracy of the predictors for integer programs, which typically also have higher branch frequencies, is lower than for the loop-intensive scientific programs. We can attack this problem in two ways: by increasing the size of the buffer and by increasing the accuracy of the scheme we use for each prediction. A buffer with 4K entries, however, as Figure 2.6 shows, performs quite comparably to an infinite buffer, at least for benchmarks like those in SPEC. The data in Figure 2.6 make it clear that the hit rate of the

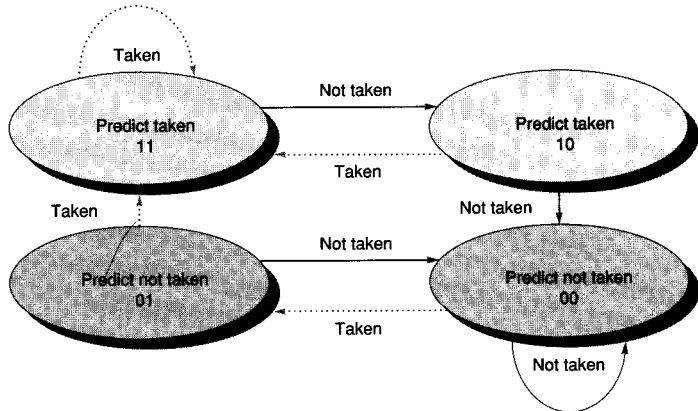


Figure 2.4 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an n -bit saturating counter for each entry in the prediction buffer. With an n -bit counter, the counter can take on values between 0 and $2^n - 1$: When the counter is greater than or equal to one-half of its maximum value (2^{n-1}), the branch is predicted as taken; otherwise, it is predicted untaken. Studies of n -bit predictors have shown that the 2-bit predictors do almost as well, and thus most systems rely on 2-bit branch predictors rather than the more general n -bit predictors.

buffer is not the major limiting factor. As we mentioned above, simply increasing the number of bits per predictor without changing the predictor structure also has little impact. Instead, we need to look at how we might increase the accuracy of each predictor.

Correlating Branch Predictors

The 2-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch. It may be possible to improve the prediction accuracy if we also look at the recent behavior of *other* branches rather than just the branch we are trying to predict. Consider a small code fragment from the eqntott benchmark, a member of early SPEC benchmark suites that displayed particularly bad branch prediction behavior:

```
if (aa==2)
    aa=0;
if (bb==2)
    bb=0;
if (aa!=bb) {
```

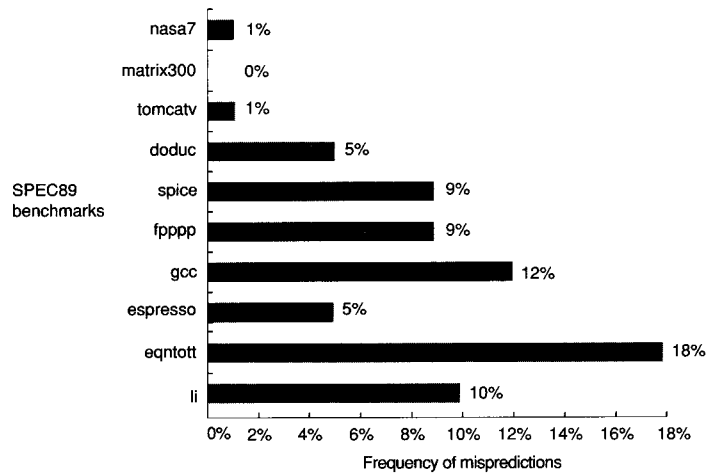



Figure 2.5 Prediction accuracy of a 4096-entry 2-bit prediction buffer for the SPEC89 benchmarks. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the FP programs (average of 4%). Omitting the FP kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch-prediction study done using the IBM Power architecture and optimized code for that system. See Pan, So, and Rameh [1992]. Although this data is for an older version of a subset of the SPEC benchmarks, the newer benchmarks are larger and would show slightly worse behavior, especially for the integer benchmarks.

Here is the MIPS code that we would typically generate for this code fragment assuming that aa and bb are assigned to registers R1 and R2:

```

                DADDIU    R3,R1,#-2
                BNEZ      R3,L1          ;branch b1    (aa!=2)
                DADD      R1,R0,R0      ;aa=0
L1:            DADDIU    R3,R2,#-2
                BNEZ      R3,L2          ;branch b2    (bb!=2)
                DADD      R2,R0,R0      ;bb=0
L2:            DSUBU      R3,R1,R2      ;R3=aa-bb
                BEQZ      R3,L3          ;branch b3    (aa==bb)

```

Let's label these branches b1, b2, and b3. The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2. Clearly, if branches b1 and b2 are both not taken (i.e., if the conditions both evaluate to true and aa and bb are both assigned 0), then b3 will be taken, since aa and bb are clearly equal. A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior.

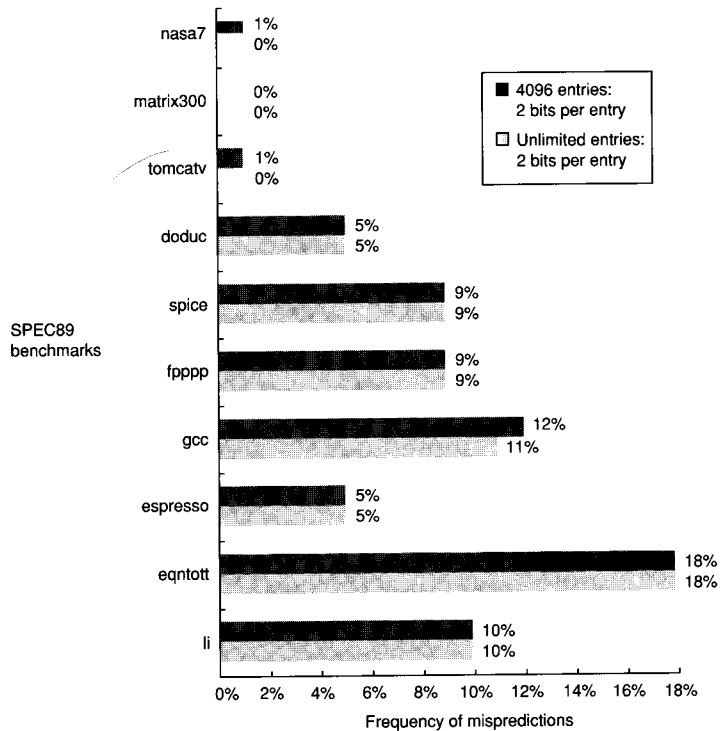


Figure 2.6 Prediction accuracy of a 4096-entry 2-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks. Although this data is for an older version of a subset of the SPEC benchmarks, the results would be comparable for newer versions with perhaps as many as 8K entries needed to match an infinite 2-bit predictor.

Branch predictors that use the behavior of other branches to make a prediction are called *correlating predictors* or *two-level predictors*. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch. For example, a (1,2) predictor uses the behavior of the last branch to choose from among a pair of 2-bit branch predictors in predicting a particular branch. In the general case an (m,n) predictor uses the behavior of the last m branches to choose from 2^m branch predictors, each of which is an n -bit predictor for a single branch. The attraction of this type of correlating branch predictor is that it can yield higher prediction rates than the 2-bit scheme and requires only a trivial amount of additional hardware.

The simplicity of the hardware comes from a simple observation: The global history of the most recent m branches can be recorded in an m -bit shift register, where each bit records whether the branch was taken or not taken. The branch-prediction buffer can then be indexed using a concatenation of the low-order bits from the branch address with the m -bit global history. For example, in a (2,2)

buffer with 64 total entries, the 4 low-order address bits of the branch (word address) and the 2 global bits representing the behavior of the two most recently executed branches form a 6-bit index that can be used to index the 64 counters.

How much better do the correlating branch predictors work when compared with the standard 2-bit scheme? To compare them fairly, we must compare predictors that use the same number of state bits. The number of bits in an (m,n) predictor is

$$2^m \times n \times \text{Number of prediction entries selected by the branch address}$$

A 2-bit predictor with no global history is simply a (0,2) predictor.

Example How many bits are in the (0,2) branch predictor with 4K entries? How many entries are in a (2,2) predictor with the same number of bits?

Answer The predictor with 4K entries has

$$2^0 \times 2 \times 4K = 8K \text{ bits}$$

How many branch-selected entries are in a (2,2) predictor that has a total of 8K bits in the prediction buffer? We know that

$$2^2 \times 2 \times \text{Number of prediction entries selected by the branch} = 8K$$

Hence, the number of prediction entries selected by the branch = 1K.

Figure 2.7 compares the misprediction rates of the earlier (0,2) predictor with 4K entries and a (2,2) predictor with 1K entries. As you can see, this correlating predictor not only outperforms a simple 2-bit predictor with the same total number of state bits, it often outperforms a 2-bit predictor with an unlimited number of entries.

Tournament Predictors: Adaptively Combining Local and Global Predictors

The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor using only local information failed on some important branches and that, by adding global information, the performance could be improved. *Tournament predictors* take this insight to the next level, by using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector. Tournament predictors can achieve both better accuracy at medium sizes (8K–32K bits) and also make use of very large numbers of prediction bits effectively. Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor (local, global, or even some mix) was most

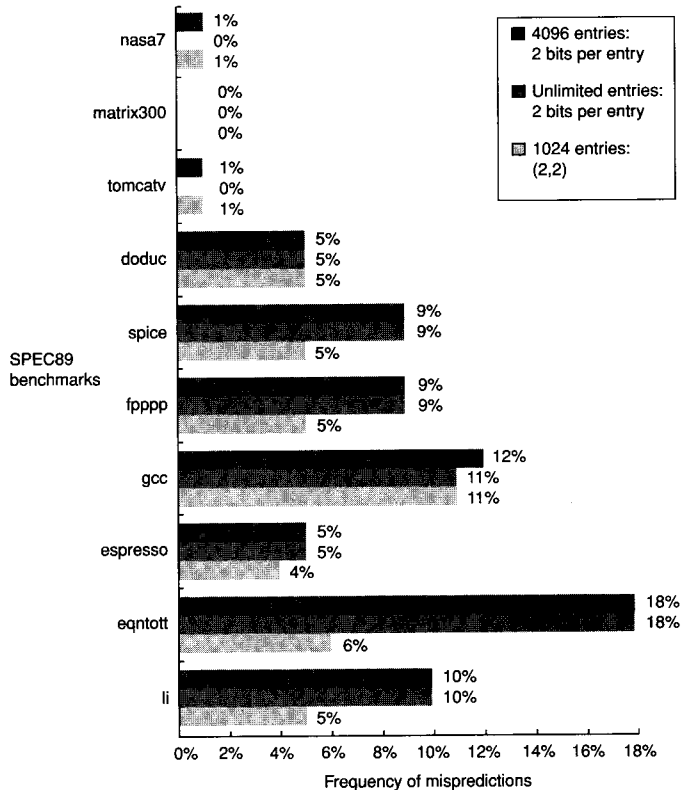


Figure 2.7 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although this data is for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

effective in recent predictions. As in a simple 2-bit predictor, the saturating counter requires two mispredictions before changing the identity of the preferred predictor.

The advantage of a tournament predictor is its ability to select the right predictor for a particular branch, which is particularly crucial for the integer benchmarks. A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks.

Figure 2.8 looks at the performance of three different predictors (a local 2-bit predictor, a correlating predictor, and a tournament predictor) for different numbers of bits using SPEC89 as the benchmark. As we saw earlier, the prediction

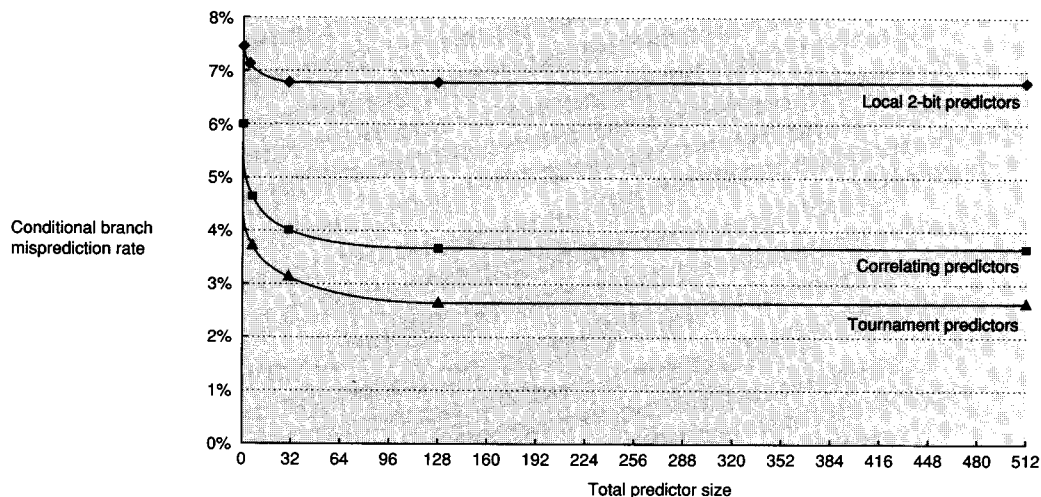


Figure 2.8 The misprediction rate for three different predictors on SPEC89 as the total number of bits is increased. The predictors are a local 2-bit predictor, a correlating predictor, which is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although this data is for an older version of SPEC, data for more recent SPEC benchmarks would show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

capability of the local predictor does not improve beyond a certain size. The correlating predictor shows a significant improvement, and the tournament predictor generates slightly better performance. For more recent versions of the SPEC, the results would be similar, but the asymptotic behavior would not be reached until slightly larger-sized predictors.

In 2005, tournament predictors using about 30K bits are the standard in processors like the Power5 and Pentium 4. The most advanced of these predictors has been on the Alpha 21264, although both the Pentium 4 and Power5 predictors are similar. The 21264's tournament predictor uses 4K 2-bit counters indexed by the local branch address to choose from among a global predictor and a local predictor. The global predictor also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor.

The local predictor consists of a two-level predictor. The top level is a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. That is, if the branch was taken 10 or more times in a row, the entry in the local history table will be all 1s. If the branch is alternately taken and untaken, the history entry consists of alternating 0s and 1s. This 10-bit history allows patterns of up to 10 branches to be discovered and predicted. The selected entry from the local history table is used to index a table of 1K entries consisting of 3-bit saturating counters, which provide the local prediction. This combination, which uses a total of 29K bits, leads to high accuracy in branch prediction.

To examine the effect on performance, we need to know the prediction accuracy as well as the branch frequency, since the importance of accurate prediction is larger in programs with higher branch frequency. For example, the integer programs in the SPEC suite have higher branch frequencies than those of the more easily predicted FP programs. For the 21264's predictor, the SPECfp95 benchmarks have less than 1 misprediction per 1000 completed instructions, and for SPECint95, there are about 11.5 mispredictions per 1000 completed instructions. This corresponds to misprediction rates of less than 0.5% for the floating-point programs and about 14% for the integer programs.

Later versions of SPEC contain programs with larger data sets and larger code, resulting in higher miss rates. Thus, the importance of branch prediction has increased. In Section 2.11, we will look at the performance of the Pentium 4 branch predictor on programs in the SPEC2000 suite and see that, despite more aggressive branch prediction, the branch-prediction miss rates for the integer programs remain significant.

2.4

Overcoming Data Hazards with Dynamic Scheduling

A simple statically scheduled pipeline fetches an instruction and issues it, unless there was a data dependence between an instruction already in the pipeline and the fetched instruction that cannot be hidden with bypassing or forwarding. (Forwarding logic reduces the effective pipeline latency so that the certain dependences do not result in hazards.) If there is a data dependence that cannot be hidden, then the hazard detection hardware stalls the pipeline starting with the instruction that uses the result. No new instructions are fetched or issued until the dependence is cleared.

In this section, we explore *dynamic scheduling*, in which the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior. Dynamic scheduling offers several advantages: It enables handling some cases when dependences are unknown at compile time (for example, because they may involve a memory reference), and it simplifies the compiler. Perhaps most importantly, it allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve. Almost as importantly, dynamic scheduling allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline. In Section 2.6, we explore hardware speculation, a technique with significant performance advantages, which builds on dynamic scheduling. As we will see, the advantages of dynamic scheduling are gained at a cost of a significant increase in hardware complexity.

Although a dynamically scheduled processor cannot change the data flow, it tries to avoid stalling when dependences are present. In contrast, static pipeline scheduling by the compiler (covered in Section 2.2) tries to minimize stalls by separating dependent instructions so that they will not lead to hazards. Of course, compiler pipeline scheduling can also be used on code destined to run on a processor with a dynamically scheduled pipeline.

Dynamic Scheduling: The Idea

A major limitation of simple pipelining techniques is that they use in-order instruction issue and execution: Instructions are issued in program order, and if an instruction is stalled in the pipeline, no later instructions can proceed. Thus, if there is a dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall will result. If there are multiple functional units, these units could lie idle. If instruction j depends on a long-running instruction i , currently in execution in the pipeline, then all instructions after j must be stalled until i is finished and j can execute. For example, consider this code:

```
DIV.D      F0,F2,F4
ADD.D      F10,F0,F8
SUB.D      F12,F8,F14
```

The SUB.D instruction cannot execute because the dependence of ADD.D on DIV.D causes the pipeline to stall; yet SUB.D is not data dependent on anything in the pipeline. This hazard creates a performance limitation that can be eliminated by not requiring instructions to execute in program order.

In the classic five-stage pipeline, both structural and data hazards could be checked during instruction decode (ID): When an instruction could execute without hazards, it was issued from ID knowing that all data hazards had been resolved.

To allow us to begin executing the SUB.D in the above example, we must separate the issue process into two parts: checking for any structural hazards and waiting for the absence of a data hazard. Thus, we still use in-order instruction issue (i.e., instructions issued in program order), but we want an instruction to begin execution as soon as its data operands are available. Such a pipeline does *out-of-order execution*, which implies *out-of-order completion*.

Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline. Consider the following MIPS floating-point code sequence:

```
DIV.D      F0,F2,F4
ADD.D      F6,F0,F8
SUB.D      F8,F10,F14
MUL.D      F6,F10,F8
```

There is an antidependence between the ADD.D and the SUB.D, and if the pipeline executes the SUB.D before the ADD.D (which is waiting for the DIV.D), it will violate the antidependence, yielding a WAR hazard. Likewise, to avoid violating output dependences, such as the write of F6 by MUL.D, WAW hazards must be handled. As we will see, both these hazards are avoided by the use of register renaming.

Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that *exactly* those exceptions that would arise if the program

were executed in strict program order *actually* do arise. Dynamically scheduled processors preserve exception behavior by ensuring that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed; we will see shortly how this property can be guaranteed.

Although exception behavior must be preserved, dynamically scheduled processors may generate *imprecise* exceptions. An exception is *imprecise* if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order. Imprecise exceptions can occur because of two possibilities:

1. The pipeline may have *already completed* instructions that are *later* in program order than the instruction causing the exception.
2. The pipeline may have *not yet completed* some instructions that are *earlier* in program order than the instruction causing the exception.

Imprecise exceptions make it difficult to restart execution after an exception. Rather than address these problems in this section, we will discuss a solution that provides precise exceptions in the context of a processor with speculation in Section 2.6. For floating-point exceptions, other solutions have been used, as discussed in Appendix J.

To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages:

1. *Issue*—Decode instructions, check for structural hazards.
2. *Read operands*—Wait until no data hazards, then read operands.

An instruction fetch stage precedes the issue stage and may fetch either into an instruction register or into a queue of pending instructions; instructions are then issued from the register or queue. The EX stage follows the read operands stage, just as in the five-stage pipeline. Execution may take multiple cycles, depending on the operation.

We distinguish when an instruction *begins execution* and when it *completes execution*; between the two times, the instruction is *in execution*. Our pipeline allows multiple instructions to be in execution at the same time, and without this capability, a major advantage of dynamic scheduling is lost. Having multiple instructions in execution at once requires multiple functional units, pipelined functional units, or both. Since these two capabilities—pipelined functional units and multiple functional units—are essentially equivalent for the purposes of pipeline control, we will assume the processor has multiple functional units.

In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order. *Scoreboarding* is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard, which developed this capability, and we discuss it in Appendix

A. Here, we focus on a more sophisticated technique, called *Tomasulo's algorithm*, that has several major enhancements over scoreboarding.

Dynamic Scheduling Using Tomasulo's Approach

The IBM 360/91 floating-point unit used a sophisticated scheme to allow out-of-order execution. This scheme, invented by Robert Tomasulo, tracks when operands for instructions are available, to minimize RAW hazards, and introduces register renaming, to minimize WAW and WAR hazards. There are many variations on this scheme in modern processors, although the key concepts of tracking instruction dependences to allow execution as soon as operands are available and renaming registers to avoid WAR and WAW hazards are common characteristics.

IBM's goal was to achieve high floating-point performance from an instruction set and from compilers designed for the entire 360 computer family, rather than from specialized compilers for the high-end processors. The 360 architecture had only four double-precision floating-point registers, which limits the effectiveness of compiler scheduling; this fact was another motivation for the Tomasulo approach. In addition, the IBM 360/91 had long memory accesses and long floating-point delays, which Tomasulo's algorithm was designed to overcome. At the end of the section, we will see that Tomasulo's algorithm can also support the overlapped execution of multiple iterations of a loop.

We explain the algorithm, which focuses on the floating-point unit and load-store unit, in the context of the MIPS instruction set. The primary difference between MIPS and the 360 is the presence of register-memory instructions in the latter architecture. Because Tomasulo's algorithm uses a load functional unit, no significant changes are needed to add register-memory addressing modes. The IBM 360/91 also had pipelined functional units, rather than multiple functional units, but we describe the algorithm as if there were multiple functional units. It is a simple conceptual extension to also pipeline those functional units.

As we will see, RAW hazards are avoided by executing an instruction only when its operands are available. WAR and WAW hazards, which arise from name dependences, are eliminated by register renaming. *Register renaming* eliminates these hazards by renaming all destination registers, including those with a pending read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of an operand.

To better understand how register renaming eliminates WAR and WAW hazards, consider the following example code sequence that includes both a potential WAR and WAW hazard:

DIV.D	F0,F2,F4
ADD.D	F6,F0,F8
S.D	F6,0(R1)
SUB.D	F8,F10,F14
MUL.D	F6,F10,F8

There is an antidependence between the ADD.D and the SUB.D and an output dependence between the ADD.D and the MUL.D, leading to two possible hazards: a WAR hazard on the use of F8 by ADD.D and a WAW hazard since the ADD.D may finish later than the MUL.D. There are also three true data dependences: between the DIV.D and the ADD.D, between the SUB.D and the MUL.D, and between the ADD.D and the S.D.

These two name dependences can both be eliminated by register renaming. For simplicity, assume the existence of two temporary registers, S and T. Using S and T, the sequence can be rewritten without any dependences as

DIV.D	F0, F2, F4
ADD.D	S, F0, F8
S.D	S, 0(R1)
SUB.D	T, F10, F14
MUL.D	F6, F10, T

In addition, any subsequent uses of F8 must be replaced by the register T. In this code segment, the renaming process can be done statically by the compiler. Finding any uses of F8 that are later in the code requires either sophisticated compiler analysis or hardware support, since there may be intervening branches between the above code segment and a later use of F8. As we will see, Tomasulo's algorithm can handle renaming across branches.

In Tomasulo's scheme, register renaming is provided by *reservation stations*, which buffer the operands of instructions waiting to issue. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register. In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming.

Since there can be more reservation stations than real registers, the technique can even eliminate hazards arising from name dependences that could not be eliminated by a compiler. As we explore the components of Tomasulo's scheme, we will return to the topic of register renaming and see exactly how the renaming occurs and how it eliminates WAR and WAW hazards.

The use of reservation stations, rather than a centralized register file, leads to two other important properties. First, hazard detection and execution control are distributed: The information held in the reservation stations at each functional unit determine when an instruction can begin execution at that unit. Second, results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers. This bypassing is done with a common result bus that allows all units waiting for an operand to be loaded simultaneously (on the 360/91 this is called the *common data bus*, or CDB). In pipelines with multiple execution units and issuing multiple instructions per clock, more than one result bus will be needed.

Figure 2.9 shows the basic structure of a Tomasulo-based processor, including both the floating-point unit and the load-store unit; none of the execution control tables are shown. Each reservation station holds an instruction that has been issued and is awaiting execution at a functional unit, and either the operand values for that instruction, if they have already been computed, or else the names of the reservation stations that will provide the operand values.

The load buffers and store buffers hold data or addresses coming from and going to memory and behave almost exactly like reservation stations, so we distinguish them only when necessary. The floating-point registers are connected by a pair of buses to the functional units and by a single bus to the store buffers. All

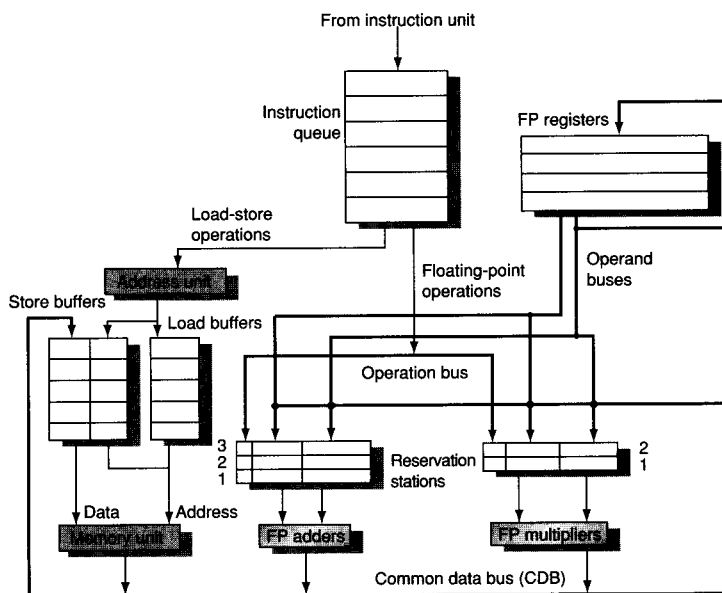


Figure 2.9 The basic structure of a MIPS floating-point unit using Tomasulo's algorithm. Instructions are sent from the instruction unit into the instruction queue from which they are issued in FIFO order. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: hold the components of the effective address until it is computed, track outstanding loads that are waiting on the memory, and hold the results of completed loads that are waiting for the CDB. Similarly, store buffers have three functions: hold the components of the effective address until it is computed, hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and hold the address and value to store until the memory unit is available. All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers. The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

results from the functional units and from memory are sent on the common data bus, which goes everywhere except to the load buffer. All reservation stations have tag fields, employed by the pipeline control.

Before we describe the details of the reservation stations and the algorithm, let's look at the steps an instruction goes through. There are only three steps, although each one can now take an arbitrary number of clock cycles:

1. *Issue*—Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to ensure the maintenance of correct data flow. If there is a matching reservation station that is empty, issue the instruction to the station with the operand values, if they are currently in the registers. If there is not an empty reservation station, then there is a structural hazard and the instruction stalls until a station or buffer is freed. If the operands are not in the registers, keep track of the functional units that will produce the operands. This step renames registers, eliminating WAR and WAW hazards. (This stage is sometimes called *dispatch* in a dynamically scheduled processor.)
2. *Execute*—If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed. When an operand becomes available, it is placed into any reservation station awaiting it. When all the operands are available, the operation can be executed at the corresponding functional unit. By delaying instruction execution until the operands are available, RAW hazards are avoided. (Some dynamically scheduled processors call this step “issue,” but we use the name “execute,” which was used in the first dynamically scheduled processor, the CDC 6600.)

Notice that several instructions could become ready in the same clock cycle for the same functional unit. Although independent functional units could begin execution in the same clock cycle for different instructions, if more than one instruction is ready for a single functional unit, the unit will have to choose among them. For the floating-point reservation stations, this choice may be made arbitrarily; loads and stores, however, present an additional complication.

Loads and stores require a two-step execution process. The first step computes the effective address when the base register is available, and the effective address is then placed in the load or store buffer. Loads in the load buffer execute as soon as the memory unit is available. Stores in the store buffer wait for the value to be stored before being sent to the memory unit. Loads and stores are maintained in program order through the effective address calculation, which will help to prevent hazards through memory, as we will see shortly.

To preserve exception behavior, no instruction is allowed to initiate execution until all branches that precede the instruction in program order have completed. This restriction guarantees that an instruction that causes an exception during execution really would have been executed. In a processor using branch prediction (as all dynamically scheduled processors do), this

means that the processor must know that the branch prediction was correct before allowing an instruction after the branch to begin execution. If the processor records the occurrence of the exception, but does not actually raise it, an instruction can start execution but not stall until it enters Write Result.

As we will see, speculation provides a more flexible and more complete method to handle exceptions, so we will delay making this enhancement and show how speculation handles this problem later.

3. *Write result*—When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. Stores are buffered in the store buffer until both the value to be stored and the store address are available, then the result is written as soon as the memory unit is free.

The data structures that detect and eliminate hazards are attached to the reservation stations, to the register file, and to the load and store buffers with slightly different information attached to different objects. These tags are essentially names for an extended set of virtual registers used for renaming. In our example, the tag field is a 4-bit quantity that denotes one of the five reservation stations or one of the five load buffers. As we will see, this produces the equivalent of 10 registers that can be designated as result registers (as opposed to the 4 double-precision registers that the 360 architecture contains). In a processor with more real registers, we would want renaming to provide an even larger set of virtual registers. The tag field describes which reservation station contains the instruction that will produce a result needed as a source operand.

Once an instruction has issued and is waiting for a source operand, it refers to the operand by the reservation station number where the instruction that will write the register has been assigned. Unused values, such as zero, indicate that the operand is already available in the registers. Because there are more reservation stations than actual register numbers, WAW and WAR hazards are eliminated by renaming results using reservation station numbers. Although in Tomasulo's scheme the reservation stations are used as the extended virtual registers, other approaches could use a register set with additional registers or a structure like the reorder buffer, which we will see in Section 2.6.

In Tomasulo's scheme, as well as the subsequent methods we look at for supporting speculation, results are broadcasted on a bus (the CDB), which is monitored by the reservation stations. The combination of the common result bus and the retrieval of results from the bus by the reservation stations implements the forwarding and bypassing mechanisms used in a statically scheduled pipeline. In doing so, however, a dynamically scheduled scheme introduces one cycle of latency between source and result, since the matching of a result and its use cannot be done until the Write Result stage. Thus, in a dynamically scheduled pipeline, the effective latency between a producing instruction and a consuming instruction is at least one cycle longer than the latency of the functional unit producing the result.

In describing the operation of this scheme, we use a terminology taken from the CDC scoreboard scheme (see Appendix A) rather than introduce new terminology, showing the terminology used by the IBM 360/91 for historical reference. It is important to remember that the tags in the Tomasulo scheme refer to the buffer or unit that will produce a result; the register names are discarded when an instruction issues to a reservation station.

Each reservation station has seven fields:

- Op—The operation to perform on source operands S1 and S2.
- Qj, Qk—The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary. (The IBM 360/91 calls these SINKunit and SOURCEunit.)
- Vj, Vk—The value of the source operands. Note that only one of the V field or the Q field is valid for each operand. For loads, the Vk field is used to hold the offset field. (These fields are called SINK and SOURCE on the IBM 360/91.)
- A—Used to hold information for the memory address calculation for a load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.
- Busy—Indicates that this reservation station and its accompanying functional unit are occupied.

The register file has a field, Qi:

- Qi—The number of the reservation station that contains the operation whose result should be stored into this register. If the value of Qi is blank (or 0), no currently active instruction is computing a result destined for this register, meaning that the value is simply the register contents.

The load and store buffers each have a field, A, which holds the result of the effective address once the first step of execution has been completed.

In the next section, we will first consider some examples that show how these mechanisms work and then examine the detailed algorithm.

2.5

Dynamic Scheduling: Examples and the Algorithm

Before we examine Tomasulo's algorithm in detail, let's consider a few examples, which will help illustrate how the algorithm works.

Example Show what the information tables look like for the following code sequence when only the first load has completed and written its result:

1.	L.D	F6,32(R2)
2.	L.D	F2,44(R3)
3.	MUL.D	F0,F2,F4
4.	SUB.D	F8,F2,F6
5.	DIV.D	F10,F0,F6
6.	ADD.D	F6,F8,F2

Answer Figure 2.10 shows the result in three tables. The numbers appended to the names add, mult, and load stand for the tag for that reservation station—Add1 is the tag for the result from the first add unit. In addition we have included an instruction status table. This table is included only to help you understand the algorithm; it is *not* actually a part of the hardware. Instead, the reservation station keeps the state of each operation that has issued.

Tomasulo's scheme offers two major advantages over earlier and simpler schemes: (1) the distribution of the hazard detection logic and (2) the elimination of stalls for WAW and WAR hazards.

The first advantage arises from the distributed reservation stations and the use of the Common Data Bus (CDB). If multiple instructions are waiting on a single result, and each instruction already has its other operand, then the instructions can be released simultaneously by the broadcast of the result on the CDB. If a centralized register file were used, the units would have to read their results from the registers when register buses are available.

The second advantage, the elimination of WAW and WAR hazards, is accomplished by renaming registers using the reservation stations, and by the process of storing operands into the reservation station as soon as they are available.

For example, the code sequence in Figure 2.10 issues both the DIV.D and the ADD.D, even though there is a WAR hazard involving F6. The hazard is eliminated in one of two ways. First, if the instruction providing the value for the DIV.D has completed, then V_k will store the result, allowing DIV.D to execute independent of the ADD.D (this is the case shown). On the other hand, if the L.D had not completed, then Q_k would point to the Load1 reservation station, and the DIV.D instruction would be independent of the ADD.D. Thus, in either case, the ADD.D can issue and begin executing. Any uses of the result of the DIV.D would point to the reservation station, allowing the ADD.D to complete and store its value into the registers without affecting the DIV.D.

We'll see an example of the elimination of a WAW hazard shortly. But let's first look at how our earlier example continues execution. In this example, and the ones that follow in this chapter, assume the following latencies: load is 1 clock cycle, add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles.

Instruction		Instruction status						
		Issue	Execute				Write Result	
L.D	F6, 32(R2)	✓	✓				✓	
L.D	F2, 44(R3)	✓	✓					
MUL.D	F0, F2, F4	✓						
SUB.D	F8, F2, F6	✓						
DIV.D	F10, F0, F6	✓						
ADD.D	F6, F8, F2	✓						

Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					45 + Regs[R3]
Add1	yes	SUB		Mem[34 + Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[34 + Regs[R2]]	Mult1		

Register status									
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Figure 2.10 Reservation stations and register tags shown when all of the instructions have issued, but only the first load instruction has completed and written its result to the CDB. The second load has completed effective address calculation, but is waiting on the memory unit. We use the array Regs[] to refer to the register file and the array Mem[] to refer to the memory. Remember that an operand is specified by either a Q field or a V field at any time. Notice that the ADD.D instruction, which has a WAR hazard at the WB stage, has issued and could complete before the DIV.D initiates.

Example Using the same code segment as in the previous example (page 97), show what the status tables look like when the MUL.D is ready to write its result.

Answer The result is shown in the three tables in Figure 2.11. Notice that ADD.D has completed since the operands of DIV.D were copied, thereby overcoming the WAR hazard. Notice that even if the load of F6 was delayed, the add into F6 could be executed without triggering a WAW hazard.

Instruction		Instruction status		
		Issue	Execute	Write Result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	√
MUL.D	F0,F2,F4	√	√	
SUB.D	F8,F2,F6	√	√	√
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√	√	√

Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL	Mem[45 + Regs[R3]]	Regs[F4]			
Mult2	yes	DIV		Mem[34 + Regs[R2]]	Mult1		

Register status									
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1					Mult2			

Figure 2.11 Multiply and divide are the only instructions not finished.

Tomasulo’s Algorithm: The Details

Figure 2.12 specifies the checks and steps that each instruction must go through. As mentioned earlier, loads and stores go through a functional unit for effective address computation before proceeding to independent load or store buffers. Loads take a second execution step to access memory and then go to Write Result to send the value from memory to the register file and/or any waiting reservation stations. Stores complete their execution in the Write Result stage, which writes the result to memory. Notice that all writes occur in Write Result, whether the destination is a register or memory. This restriction simplifies Tomasulo’s algorithm and is critical to its extension with speculation in Section 2.6.

Instruction state	Wait until	Action or bookkeeping
Issue FP operation	Station r empty	<pre> if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Q ← r; </pre>
Load or store	Buffer r empty	<pre> if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes; </pre>
Load only		RegisterStat[rt].Qi ← r;
Store only		<pre> if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; </pre>
Execute FP operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load-store step 1	RS[r].Qj = 0 & r is head of load-store queue	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 complete	Read from Mem[RS[r].A]
Write Result FP operation or load	Execution complete at r & CDB available	<pre> ∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no; </pre>
Store	Execution complete at r & RS[r].Qk = 0	<pre> Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no; </pre>

Figure 2.12 Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the destination, rs and rt are the source register numbers, imm is the sign-extended immediate field, and r is the reservation station or buffer that the instruction is assigned to. RS is the reservation station data structure. The value returned by an FP unit or by the load unit is called result. RegisterStat is the register status data structure (not the register file, which is Regs []). When an instruction is issued, the destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the reservation station that will produce the values needed as source operands. The instruction waits at the reservation station until both its operands are available, indicated by zero in the Q fields. The Q fields are set to zero either when this instruction is issued, or when an instruction on which this instruction depends completes and does its write back. When an instruction has finished execution and the CDB is available, it can do its write back. All the buffers, registers, and reservation stations whose value of Qj or Qk is the same as the completing reservation station update their values from the CDB and mark the Q fields to indicate that values have been received. Thus, the CDB can broadcast its result to many destinations in a single clock cycle, and if the waiting instructions have their operands, they can all begin execution on the next clock cycle. Loads go through two steps in Execute, and stores perform slightly differently during Write Result, where they may have to wait for the value to store. Remember that to preserve exception behavior, instructions should not be allowed to execute if a branch that is earlier in program order has not yet completed. Because any concept of program order is not maintained after the Issue stage, this restriction is usually implemented by preventing any instruction from leaving the Issue step, if there is a pending branch already in the pipeline. In Section 2.6, we will see how speculation support removes this restriction.

Tomasulo's Algorithm: A Loop-Based Example

To understand the full power of eliminating WAW and WAR hazards through dynamic renaming of registers, we must look at a loop. Consider the following simple sequence for multiplying the elements of an array by a scalar in F2:

```

Loop:    L.D      F0,0(R1)
          MUL.D   F4,F0,F2
          S.D      F4,0(R1)
          DADDIU  R1,R1,-8
          BNE     R1,R2,Loop; branches if R1≠R2

```

If we predict that branches are taken, using reservation stations will allow multiple executions of this loop to proceed at once. This advantage is gained without changing the code—in effect, the loop is unrolled dynamically by the hardware, using the reservation stations obtained by renaming to act as additional registers.

Let's assume we have issued all the instructions in two successive iterations of the loop, but none of the floating-point load-stores or operations has completed. Figure 2.13 shows reservation stations, register status tables, and load and store buffers at this point. (The integer ALU operation is ignored, and it is assumed the branch was predicted as taken.) Once the system reaches this state, two copies of the loop could be sustained with a CPI close to 1.0, provided the multiplies could complete in 4 clock cycles. With a latency of 6 cycles, additional iterations will need to be processed before the steady state can be reached. This requires more reservation stations to hold instructions that are in execution. As we will see later in this chapter, when extended with multiple instruction issue, Tomasulo's approach can sustain more than one instruction per clock.

A load and a store can safely be done out of order, provided they access different addresses. If a load and a store access the same address, then either

- the load is before the store in program order and interchanging them results in a WAR hazard, or
- the store is before the load in program order and interchanging them results in a RAW hazard.

Similarly, interchanging two stores to the same address results in a WAW hazard.

Hence, to determine if a load can be executed at a given time, the processor can check whether any uncompleted store that precedes the load in program order shares the same data memory address as the load. Similarly, a store must wait until there are no unexecuted loads or stores that are earlier in program order and share the same data memory address. We consider a method to eliminate this restriction in Section 2.9.

To detect such hazards, the processor must have computed the data memory address associated with any earlier memory operation. A simple, but not necessarily optimal, way to guarantee that the processor has all such addresses is to perform the effective address calculations in program order. (We really only need

Instruction		Instruction status			
		From iteration	Issue	Execute	Write Result
L.D	F0,0(R1)	1	✓	✓	
MUL.D	F4,F0,F2	1	✓		
S.D	F4,0(R1)	1	✓		
L.D	F0,0(R1)	2	✓	✓	
MUL.D	F4,F0,F2	2	✓		
S.D	F4,0(R1)	2	✓		

Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	yes	Load					Regs[R1] + 0
Load2	yes	Load					Regs[R1] - 8
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL		Regs[F2]	Load1		
Mult2	yes	MUL		Regs[F2]	Load2		
Store1	yes	Store	Regs[R1]			Mult1	
Store2	yes	Store	Regs[R1] - 8			Mult2	

Register status									
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Load2		Mult2						

Figure 2.13 Two active iterations of the loop with no instruction yet completed. Entries in the multiplier reservation stations indicate that the outstanding loads are the sources. The store reservation stations indicate that the multiply destination is the source of the value to store.

to keep the relative order between stores and other memory references; that is, loads can be reordered freely.)

Let's consider the situation of a load first. If we perform effective address calculation in program order, then when a load has completed effective address calculation, we can check whether there is an address conflict by examining the A field of all active store buffers. If the load address matches the address of any active entries in the store buffer, that load instruction is not sent to the load buffer until the conflicting store completes. (Some implementations bypass the value directly to the load from a pending store, reducing the delay for this RAW hazard.)

Stores operate similarly, except that the processor must check for conflicts in both the load buffers and the store buffers, since conflicting stores cannot be reordered with respect to either a load or a store.

A dynamically scheduled pipeline can yield very high performance, provided branches are predicted accurately—an issue we addressed in the last section. The major drawback of this approach is the complexity of the Tomasulo scheme, which requires a large amount of hardware. In particular, each reservation station must contain an associative buffer, which must run at high speed, as well as complex control logic. The performance can also be limited by the single CDB. Although additional CDBs can be added, each CDB must interact with each reservation station, and the associative tag-matching hardware would need to be duplicated at each station for each CDB.

In Tomasulo's scheme two different techniques are combined: the renaming of the architectural registers to a larger set of registers and the buffering of source operands from the register file. Source operand buffering resolves WAR hazards that arise when the operand is available in the registers. As we will see later, it is also possible to eliminate WAR hazards by the renaming of a register together with the buffering of a result until no outstanding references to the earlier version of the register remain. This approach will be used when we discuss hardware speculation.

Tomasulo's scheme was unused for many years after the 360/91, but was widely adopted in multiple-issue processors starting in the 1990s for several reasons:

1. It can achieve high performance without requiring the compiler to target code to a specific pipeline structure, a valuable property in the era of shrink-wrapped mass market software.
2. Although Tomasulo's algorithm was designed before caches, the presence of caches, with the inherently unpredictable delays, has become one of the major motivations for dynamic scheduling. Out-of-order execution allows the processors to continue executing instructions while awaiting the completion of a cache miss, thus hiding all or part of the cache miss penalty.
3. As processors became more aggressive in their issue capability and designers are concerned with the performance of difficult-to-schedule code (such as most nonnumeric code), techniques such as register renaming and dynamic scheduling become more important.
4. Because dynamic scheduling is a key component of speculation, it was adopted along with hardware speculation in the mid-1990s.

2.6

Hardware-Based Speculation

As we try to exploit more instruction-level parallelism, maintaining control dependences becomes an increasing burden. Branch prediction reduces the direct stalls attributable to branches, but for a processor executing multiple instructions

per clock, just predicting branches accurately may not be sufficient to generate the desired amount of instruction-level parallelism. A wide issue processor may need to execute a branch every clock cycle to maintain maximum performance. Hence, exploiting more parallelism requires that we overcome the limitation of control dependence.

Overcoming control dependence is done by speculating on the outcome of branches and executing the program as if our guesses were correct. This mechanism represents a subtle, but important, extension over branch prediction with dynamic scheduling. In particular, with speculation, we fetch, issue, and *execute* instructions, as if our branch predictions were always correct; dynamic scheduling only fetches and issues such instructions. Of course, we need mechanisms to handle the situation where the speculation is incorrect. Appendix G discusses a variety of mechanisms for supporting speculation by the compiler. In this section, we explore *hardware speculation*, which extends the ideas of dynamic scheduling.

Hardware-based speculation combines three key ideas: dynamic branch prediction to choose which instructions to execute, speculation to allow the execution of instructions before the control dependences are resolved (with the ability to undo the effects of an incorrectly speculated sequence), and dynamic scheduling to deal with the scheduling of different combinations of basic blocks. (In comparison, dynamic scheduling without speculation only partially overlaps basic blocks because it requires that a branch be resolved before actually executing any instructions in the successor basic block.)

Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions. This method of executing programs is essentially a *data flow execution*: Operations execute as soon as their operands are available.

To extend Tomasulo's algorithm to support speculation, we must separate the bypassing of results among instructions, which is needed to execute an instruction speculatively, from the actual completion of an instruction. By making this separation, we can allow an instruction to execute and to bypass its results to other instructions, without allowing the instruction to perform any updates that cannot be undone, until we know that the instruction is no longer speculative.

Using the bypassed value is like performing a speculative register read, since we do not know whether the instruction providing the source register value is providing the correct result until the instruction is no longer speculative. When an instruction is no longer speculative, we allow it to update the register file or memory; we call this additional step in the instruction execution sequence *instruction commit*.

The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit *in order* and to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits. Hence, when we add speculation, we need to separate the process of completing execution from instruction commit, since instructions may finish execution considerably before they are ready to commit. Adding this commit phase

to the instruction execution sequence requires an additional set of hardware buffers that hold the results of instructions that have finished execution but have not committed. This hardware buffer, which we call the *reorder buffer*, is also used to pass results among instructions that may be speculated.

The reorder buffer (ROB) provides additional registers in the same way as the reservation stations in Tomasulo's algorithm extend the register set. The ROB holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits. Hence, the ROB is a source of operands for instructions, just as the reservation stations provide operands in Tomasulo's algorithm. The key difference is that in Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find the result in the register file. With speculation, the register file is not updated until the instruction commits (and we know definitively that the instruction should execute); thus, the ROB supplies operands in the interval between completion of instruction execution and instruction commit. The ROB is similar to the store buffer in Tomasulo's algorithm, and we integrate the function of the store buffer into the ROB for simplicity.

Each entry in the ROB contains four fields: the instruction type, the destination field, the value field, and the ready field. The instruction type field indicates whether the instruction is a branch (and has no destination result), a store (which has a memory address destination), or a register operation (ALU operation or load, which has register destinations). The destination field supplies the register number (for loads and ALU operations) or the memory address (for stores) where the instruction result should be written. The value field is used to hold the value of the instruction result until the instruction commits. We will see an example of ROB entries shortly. Finally, the ready field indicates that the instruction has completed execution, and the value is ready.

Figure 2.14 shows the hardware structure of the processor including the ROB. The ROB subsumes the store buffers. Stores still execute in two steps, but the second step is performed by instruction commit. Although the renaming function of the reservation stations is replaced by the ROB, we still need a place to buffer operations (and operands) between the time they issue and the time they begin execution. This function is still provided by the reservation stations. Since every instruction has a position in the ROB until it commits, we tag a result using the ROB entry number rather than using the reservation station number. This tagging requires that the ROB assigned for an instruction must be tracked in the reservation station. Later in this section, we will explore an alternative implementation that uses extra registers for renaming and the ROB only to track when instructions can commit.

Here are the four steps involved in instruction execution:

1. *Issue*—Get an instruction from the instruction queue. Issue the instruction if there is an empty reservation station and an empty slot in the ROB; send the operands to the reservation station if they are available in either the registers

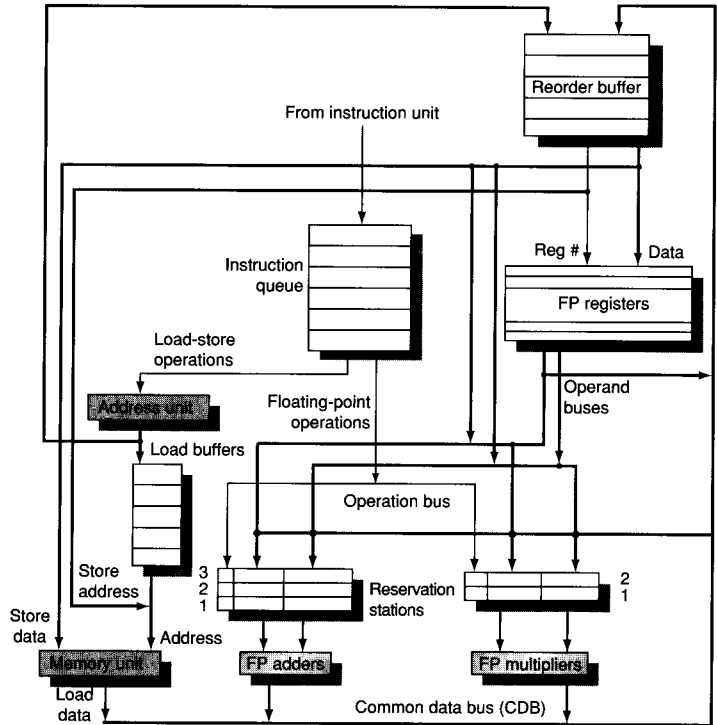


Figure 2.14 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 2.9 on page 94, which implemented Tomasulo's algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to multiple issue by making the CDB wider to allow for multiple completions per clock.

or the ROB. Update the control entries to indicate the buffers are in use. The number of the ROB entry allocated for the result is also sent to the reservation station, so that the number can be used to tag the result when it is placed on the CDB. If either all reservations are full or the ROB is full, then instruction issue is stalled until both have available entries.

2. *Execute*—If one or more of the operands is not yet available, monitor the CDB while waiting for the register to be computed. This step checks for RAW hazards. When both operands are available at a reservation station, execute the operation. Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage. Stores need only have the base register available at this step, since execution for a store at this point is only effective address calculation.

3. *Write result*—When the result is available, write it on the CDB (with the ROB tag sent when the instruction issued) and from the CDB into the ROB, as well as to any reservation stations waiting for this result. Mark the reservation station as available. Special actions are required for store instructions. If the value to be stored is available, it is written into the Value field of the ROB entry for the store. If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated. For simplicity we assume that this occurs during the Write Results stage of a store; we discuss relaxing this requirement later.
4. *Commit*—This is the final stage of completing an instruction, after which only its result remains. (Some processors call this commit phase “completion” or “graduation.”) There are three different sequences of actions at commit depending on whether the committing instruction is a branch with an incorrect prediction, a store, or any other instruction (normal commit). The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB. Committing a store is similar except that memory is updated rather than a result register. When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and execution is restarted at the correct successor of the branch. If the branch was correctly predicted, the branch is finished.

Once an instruction commits, its entry in the ROB is reclaimed and the register or memory destination is updated, eliminating the need for the ROB entry. If the ROB fills, we simply stop issuing instructions until an entry is made free. Now, let’s examine how this scheme would work with the same example we used for Tomasulo’s algorithm.

Example Assume the same latencies for the floating-point functional units as in earlier examples: add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles. Using the code segment below, the same one we used to generate Figure 2.11, show what the status tables look like when the MUL.D is ready to go to commit.

L.D	F6, 32(R2)
L.D	F2, 44(R3)
MUL.D	F0, F2, F4
SUB.D	F8, F6, F2
DIV.D	F10, F0, F6
ADD.D	F6, F8, F2

Answer Figure 2.15 shows the result in the three tables. Notice that although the SUB.D instruction has completed execution, it does not commit until the MUL.D commits. The reservation stations and register status field contain the same basic informa-

tion that they did for Tomasulo's algorithm (see page 97 for a description of those fields). The differences are that reservation station numbers are replaced with ROB entry numbers in the Qj and Qk fields, as well as in the register status fields, and we have added the Dest field to the reservation stations. The Dest field designates the ROB entry that is the destination for the result produced by this reservation station entry.

The above example illustrates the key important difference between a processor with speculation and a processor with dynamic scheduling. Compare the content of Figure 2.15 with that of Figure 2.11 on page 100, which shows the same code sequence in operation on a processor with Tomasulo's algorithm. The key difference is that, in the example above, no instruction after the earliest uncompleted instruction (MUL.D above) is allowed to complete. In contrast, in Figure 2.11 the SUB.D and ADD.D instructions have also completed.

One implication of this difference is that the processor with the ROB can dynamically execute code while maintaining a precise interrupt model. For example, if the MUL.D instruction caused an interrupt, we could simply wait until it reached the head of the ROB and take the interrupt, flushing any other pending instructions from the ROB. Because instruction commit happens in order, this yields a precise exception.

By contrast, in the example using Tomasulo's algorithm, the SUB.D and ADD.D instructions could both complete before the MUL.D raised the exception. The result is that the registers F8 and F6 (destinations of the SUB.D and ADD.D instructions) could be overwritten, and the interrupt would be imprecise.

Some users and architects have decided that imprecise floating-point exceptions are acceptable in high-performance processors, since the program will likely terminate; see Appendix G for further discussion of this topic. Other types of exceptions, such as page faults, are much more difficult to accommodate if they are imprecise, since the program must transparently resume execution after handling such an exception.

The use of a ROB with in-order instruction commit provides precise exceptions, in addition to supporting speculative execution, as the next example shows.

Example Consider the code example used earlier for Tomasulo's algorithm and shown in Figure 2.13 in execution:

```

Loop:  L.D      F0,0(R1)
        MUL.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDIU  R1,R1,#-8
        BNE     R1,R2,Loop    ;branches if R1≠R2

```

Assume that we have issued all the instructions in the loop twice. Let's also assume that the L.D and MUL.D from the first iteration have committed and all other instructions have completed execution. Normally, the store would wait in

Reorder buffer						
Entry	Busy	Instruction		State	Destination	Value
1	no	L.D	F6, 32(R2)	Commit	F6	Mem[34 + Regs[R2]]
2	no	L.D	F2, 44(R3)	Commit	F2	Mem[45 + Regs[R3]]
3	yes	MUL.D	F0, F2, F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D	F8, F2, F6	Write result	F8	#2 − #1
5	yes	DIV.D	F10, F0, F6	Execute	F10	
6	yes	ADD.D	F6, F8, F2	Write result	F6	#4 + #2

Reservation stations								
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL.D	Mem[45 + Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV.D		Mem[34 + Regs[R2]]	#3		#5	

FP register status										
Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	...	yes	yes

Figure 2.15 At the time the MUL.D is ready to commit, only the two L.D instructions have committed, although several others have completed execution. The MUL.D is at the head of the ROB, and the two L.D instructions are there only to ease understanding. The SUB.D and ADD.D instructions will not commit until the MUL.D instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The DIV.D is in execution, but has not completed solely due to its longer latency than MUL.D. The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed, but are shown for informational purposes. We do not show the entries for the load-store queue, but these entries are kept in order.

the ROB for both the effective address operand (R1 in this example) and the value (F4 in this example). Since we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

Answer Figure 2.16 shows the result in two tables.

Reorder buffer									
Entry	Busy	Instruction		State	Destination	Value			
1	no	L.D	F0,0(R1)	Commit	F0	Mem[0 + Regs[R1]]			
2	no	MUL.D	F4,F0,F2	Commit	F4	#1 × Regs[F2]			
3	yes	S.D	F4,0(R1)	Write result	0 + Regs[R1]	#2			
4	yes	DADDIU	R1,R1,#-8	Write result	R1	Regs[R1] - 8			
5	yes	BNE	R1,R2,Loop	Write result					
6	yes	L.D	F0,0(R1)	Write result	F0	Mem[#4]			
7	yes	MUL.D	F4,F0,F2	Write result	F4	#6 × Regs[F2]			
8	yes	S.D	F4,0(R1)	Write result	0 + #4	#7			
9	yes	DADDIU	R1,R1,#-8	Write result	R1	#4 - 8			
10	yes	BNE	R1,R2,Loop	Write result					

FP register status									
Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder #	6				7				
Busy	yes	no	no	no	yes	no	no	...	no

Figure 2.16 Only the L.D and MUL.D instructions have committed, although all the others have completed execution. Hence, no reservation stations are busy and none are shown. The remaining instructions will be committed as fast as possible. The first two reorder buffers are empty, but are shown for completeness.

Because neither the register values nor any memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted. Suppose that the branch BNE is not taken the first time in Figure 2.16. The instructions prior to the branch will simply commit when each reaches the head of the ROB; when the branch reaches the head of that buffer, the buffer is simply cleared and the processor begins fetching instructions from the other path.

In practice, processors that speculate try to recover as early as possible after a branch is mispredicted. This recovery can be done by clearing the ROB for all entries that appear after the mispredicted branch, allowing those that are before the branch in the ROB to continue, and restarting the fetch at the correct branch successor. In speculative processors, performance is more sensitive to the branch prediction, since the impact of a misprediction will be higher. Thus, all the aspects of handling branches—prediction accuracy, latency of misprediction detection, and misprediction recovery time—increase in importance.

Exceptions are handled by not recognizing the exception until it is ready to commit. If a speculated instruction raises an exception, the exception is recorded

in the ROB. If a branch misprediction arises and the instruction should not have been executed, the exception is flushed along with the instruction when the ROB is cleared. If the instruction reaches the head of the ROB, then we know it is no longer speculative and the exception should really be taken. We can also try to handle exceptions as soon as they arise and all earlier branches are resolved, but this is more challenging in the case of exceptions than for branch mispredict and, because it occurs less frequently, not as critical.

Figure 2.17 shows the steps of execution for an instruction, as well as the conditions that must be satisfied to proceed to the step and the actions taken. We show the case where mispredicted branches are not resolved until commit. Although speculation seems like a simple addition to dynamic scheduling, a comparison of Figure 2.17 with the comparable figure for Tomasulo's algorithm in Figure 2.12 shows that speculation adds significant complications to the control. In addition, remember that branch mispredictions are somewhat more complex as well.

There is an important difference in how stores are handled in a speculative processor versus in Tomasulo's algorithm. In Tomasulo's algorithm, a store can update memory when it reaches Write Result (which ensures that the effective address has been calculated) and the data value to store is available. In a speculative processor, a store updates memory only when it reaches the head of the ROB. This difference ensures that memory is not updated until an instruction is no longer speculative.

Figure 2.17 has one significant simplification for stores, which is unneeded in practice. Figure 2.17 requires stores to wait in the Write Result stage for the register source operand whose value is to be stored; the value is then moved from the *Vk* field of the store's reservation station to the Value field of the store's ROB entry. In reality, however, the value to be stored need not arrive until *just before* the store commits and can be placed directly into the store's ROB entry by the sourcing instruction. This is accomplished by having the hardware track when the source value to be stored is available in the store's ROB entry and searching the ROB on every instruction completion to look for dependent stores.

This addition is not complicated, but adding it has two effects: We would need to add a field to the ROB, and Figure 2.17, which is already in a small font, would be even longer! Although Figure 2.17 makes this simplification, in our examples, we will allow the store to pass through the Write Result stage and simply wait for the value to be ready when it commits.

Like Tomasulo's algorithm, we must avoid hazards through memory. WAW and WAR hazards through memory are eliminated with speculation because the actual updating of memory occurs in order, when a store is at the head of the ROB, and hence, no earlier loads or stores can still be pending. RAW hazards through memory are maintained by two restrictions:

1. not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and

Status	Wait until	Action or bookkeeping
Issue all instructions		<pre> if (RegisterStat[rs].Busy) /* in-flight instr. writes rs */ { h ← RegisterStat[rs].Reorder; if (ROB[h].Ready) /* Instr completed already */ { RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0; } else { RS[r].Qj ← h; } /* wait for instruction */ } else { RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0; }; RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd; ROB[b].Ready ← no; </pre>
FP operations and stores	Reservation station (r) and ROB (b) both available	<pre> if (RegisterStat[rt].Busy) /* in-flight instr writes rt */ { h ← RegisterStat[rt].Reorder; if (ROB[h].Ready) /* Instr completed already */ { RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0; } else { RS[r].Qk ← h; } /* wait for instruction */ } else { RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0; }; </pre>
FP operations		<pre> RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd; </pre>
Loads		<pre> RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt; </pre>
Stores		<pre> RS[r].A ← imm; </pre>
Execute FP op	(RS[r].Qj == 0) and (RS[r].Qk == 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj == 0) and there are no stores earlier in the queue	<pre> RS[r].A ← RS[r].Vj + RS[r].A; </pre>
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from Mem[RS[r].A]
Store	(RS[r].Qj == 0) and store at queue head	<pre> ROB[h].Address ← RS[r].Vj + RS[r].A; </pre>
Write result all but store	Execution done at r and CDB available	<pre> b ← RS[r].Dest; RS[r].Busy ← no; ∀x (if (RS[x].Qj == b) { RS[x].Vj ← result; RS[x].Qj ← 0; }); ∀x (if (RS[x].Qk == b) { RS[x].Vk ← result; RS[x].Qk ← 0; }); ROB[b].Value ← result; ROB[b].Ready ← yes; </pre>
Store	Execution done at r and (RS[r].Qk == 0)	<pre> ROB[h].Value ← RS[r].Vk; </pre>
Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes	<pre> d ← ROB[h].Dest; /* register dest, if exists */ if (ROB[h].Instruction == Branch) { if (branch is mispredicted) { clear ROB[h], RegisterStat; fetch branch dest; }; } else if (ROB[h].Instruction == Store) { Mem[ROB[h].Destination] ← ROB[h].Value; } else /* put the result in the register destination */ { Regs[d] ← ROB[h].Value; }; ROB[h].Busy ← no; /* free up ROB entry */ /* free up dest register if no one else writing it */ if (RegisterStat[d].Reorder == h) { RegisterStat[d].Busy ← no; }; </pre>

Figure 2.17 Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the destination, rs and rt are the sources, r is the reservation station allocated, b is the assigned ROB entry, and h is the head entry of the ROB. RS is the reservation station data structure. The value returned by a reservation station is called the result. RegisterStat is the register data structure, Regs represents the actual registers, and ROB is the reorder buffer data structure.

2. maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.

Together, these two restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data. Some speculative processors will actually bypass the value from the store to the load directly, when such a RAW hazard occurs. Another approach is to predict potential collisions using a form of value prediction; we consider this in Section 2.9.

Although this explanation of speculative execution has focused on floating point, the techniques easily extend to the integer registers and functional units, as we will see in the “Putting It All Together” section. Indeed, speculation may be more useful in integer programs, since such programs tend to have code where the branch behavior is less predictable. Additionally, these techniques can be extended to work in a multiple-issue processor by allowing multiple instructions to issue and commit every clock. In fact, speculation is probably most interesting in such processors, since less ambitious techniques can probably exploit sufficient ILP within basic blocks when assisted by a compiler.

2.7

Exploiting ILP Using Multiple Issue and Static Scheduling

The techniques of the preceding sections can be used to eliminate data and control stalls and achieve an ideal CPI of one. To improve performance further we would like to decrease the CPI to less than one. But the CPI cannot be reduced below one if we issue only one instruction every clock cycle.

The goal of the *multiple-issue processors*, discussed in the next few sections, is to allow multiple instructions to issue in a clock cycle. Multiple-issue processors come in three major flavors:

1. statically scheduled superscalar processors,
2. VLIW (very long instruction word) processors, and
3. dynamically scheduled superscalar processors.

The two types of superscalar processors issue varying numbers of instructions per clock and use in-order execution if they are statically scheduled or out-of-order execution if they are dynamically scheduled.

VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction. VLIW processors are inherently statically scheduled by the compiler. When Intel and HP created the IA-64 architecture, described in Appendix G, they also introduced the name EPIC—explicitly parallel instruction computer—for this architectural style.

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	dynamic	hardware	static	in-order execution	mostly in the embedded space: MIPS and ARM
Superscalar (dynamic)	dynamic	hardware	dynamic	some out-of-order execution, but no speculation	none at the present
Superscalar (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Pentium 4, MIPS R12K, IBM Power5
VLIW/LIW	static	primarily software	static	all hazards determined and indicated by compiler (often implicitly)	most examples are in the embedded space, such as the TI C6x
EPIC	primarily static	primarily software	mostly static	all hazards determined and indicated explicitly by the compiler	Itanium

Figure 2.18 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix G focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.

Although statically scheduled superscalars issue a varying rather than a fixed number of instructions per clock, they are actually closer in concept to VLIWs, since both approaches rely on the compiler to schedule code for the processor. Because of the diminishing advantages of a statically scheduled superscalar as the issue width grows, statically scheduled superscalars are used primarily for narrow issue widths, normally just two instructions. Beyond that width, most designers choose to implement either a VLIW or a dynamically scheduled superscalar. Because of the similarities in hardware and required compiler technology, we focus on VLIWs in this section. The insights of this section are easily extrapolated to a statically scheduled superscalar.

Figure 2.18 summarizes the basic approaches to multiple issue and their distinguishing characteristics and shows processors that use each approach.

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 no 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 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 will see in the next chapter, the growth in overhead is a major factor limiting wider-issue processors.

Let's consider a VLIW processor with 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 16–24 bits per unit, yielding an instruction length of between 80 and 120 bits. By comparison, the Intel Itanium 1 and 2 contain 6 operations per instruction packet.

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 straight-line 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 also must deal with significantly more complicated trade-offs in optimization, since moving code across branches is expensive.

In Appendix G, we will discuss *trace scheduling*, one of these global scheduling techniques developed specifically for VLIWs; we will also explore special 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, we will rely on loop unrolling to generate long, straight-line code sequences, so that we can use local scheduling to build up VLIW instructions and 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 76 for the MIPS code) for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore delayed branches.

Answer Figure 2.19 shows the code. 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 2.2 that used unrolled and scheduled code.

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

Figure 2.19 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming no branch delay; normally the branch delay would also need to be scheduled. The issue rate is 23 operations in 9 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.

For the original VLIW model, there were both technical and logistical problems that make the approach less efficient. The technical problems are the increase in code size and the limitations of lockstep 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 in 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 Appendix G, we examine software scheduling approaches, such as software pipelining, that can achieve the benefits of unrolling without as much code expansion.

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 expand them when they are read into the cache or are decoded. In Appendix G, we show other techniques, as well as document the significant code expansion seen on IA-64.

Early VLIWs operated in lockstep; 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.

The EPIC approach, of which the IA-64 architecture is the primary example, provides solutions to many of the problems encountered in early VLIW designs, including extensions for more aggressive software speculation and methods to overcome the limitation of hardware dependence while preserving binary compatibility.

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 F). It is not clear that a multiple-issue processor is preferred over a vector processor for such applications; the costs are similar, and the vector processor is typically the same speed or faster. The potential advantages of a multiple-issue processor versus a vector processor are their ability to extract some parallelism from less structured code and their ability to easily cache all forms of data. For these reasons multiple-issue approaches have become the primary method for taking advantage of instruction-level parallelism, and vectors have become primarily an extension to these processors.

2.8

Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

So far, we have seen how the individual mechanisms of dynamic scheduling, multiple issue, and speculation work. In this section, we put all three together, which yields a microarchitecture quite similar to those in modern microprocessors. For simplicity, we consider only an issue rate of two instructions per clock, but the concepts are no different from modern processors that issue three or more instructions per clock.

Let's assume we want to extend Tomasulo's algorithm to support a two-issue superscalar pipeline with a separate integer and floating-point unit, each of which can initiate an operation on every clock. We do not want to issue instructions to

the reservation stations out of order, since this could lead to a violation of the program semantics. To gain the full advantage of dynamic scheduling we will allow the pipeline to issue any combination of two instructions in a clock, using the scheduling hardware to actually assign operations to the integer and floating-point unit. Because the interaction of the integer and floating-point instructions is crucial, we also extend Tomasulo's scheme to deal with both the integer and floating-point functional units and registers, as well as incorporating speculative execution.

Two different approaches have been used to issue multiple instructions per clock in a dynamically scheduled processor, and both rely on the observation that the key is assigning a reservation station and updating the pipeline control tables. One approach is to run this step in half a clock cycle, so that two instructions can be processed in one clock cycle. A second alternative is to build the logic necessary to handle two instructions at once, including any possible dependences between the instructions. Modern superscalar processors that issue four or more instructions per clock often include both approaches: They both pipeline and widen the issue logic.

Putting together speculative dynamic scheduling with multiple issue requires overcoming one additional challenge at the back end of the pipeline: we must be able to complete and commit multiple instructions per clock. Like the challenge of issuing multiple instructions, the concepts are simple, although the implementation may be challenging in the same manner as the issue and register renaming process. We can show how the concepts fit together with an example.

Example Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

Loop:	LD	R2,0(R1)	;R2=array element
	DADDIU	R2,R2,#1	;increment R2
	SD	R2,0(R1)	;store result
	DADDIU	R1,R1,#8	;increment pointer
	BNE	R2,R3,LOOP	;branch if not last element

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.

Answer Figures 2.20 and 2.21 show the performance for a two-issue dynamically scheduled processor, without and with speculation. In this case, where a branch can be a critical performance limiter, speculation helps significantly. The third branch in

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9	19			Wait for DADDIU

Figure 2.20 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the LD following the BNE cannot start execution earlier because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch-condition evaluation allow multiple instructions to execute in the same cycle. Figure 2.21 shows this example with speculation,

the speculative processor executes in clock cycle 13, while it executes in clock cycle 19 on the nonspeculative pipeline. Because the completion rate on the non-speculative pipeline is falling behind the issue rate rapidly, the nonspeculative pipeline will stall when a few more iterations are issued. The performance of the nonspeculative processor could be improved by allowing load instructions to complete effective address calculation before a branch is decided, but unless speculative memory accesses are allowed, this improvement will gain only 1 clock per iteration.

This example clearly shows how speculation can be advantageous when there are data-dependent branches, which otherwise would limit performance. This advantage depends, however, on accurate branch prediction. Incorrect speculation will not improve performance, but will, in fact, typically harm performance.

Iteration number	Instructions		Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD	R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU	R2,R2,#1	1	5		6	7	Wait for LW
1	SD	R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	8	Commit in order
1	BNE	R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD	R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU	R2,R2,#1	4	8		9	10	Wait for LW
2	SD	R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU	R1,R1,#8	5	6		7	11	Commit in order
2	BNE	R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD	R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU	R2,R2,#1	7	11		12	13	Wait for LW
3	SD	R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU	R1,R1,#8	8	9		10	14	Executes earlier
3	BNE	R2,R3,LOOP	9	13			14	Wait for DADDIU

Figure 2.21 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation. Note that the LD following the BNE can start execution early because it is speculative.

2.9

Advanced Techniques for Instruction Delivery and Speculation

In a high-performance pipeline, especially one with multiple issue, predicting branches well is not enough; we actually have to be able to deliver a high-bandwidth instruction stream. In recent multiple-issue processors, this has meant delivering 4–8 instructions every clock cycle. We look at methods for increasing instruction delivery bandwidth first. We then turn to a set of key issues in implementing advanced speculation techniques, including the use of register renaming versus reorder buffers, the aggressiveness of speculation, and a technique called value prediction, which could further enhance ILP.

Increasing Instruction Fetch Bandwidth

A multiple issue processor will require that the average number of instructions fetched every clock cycle be at least as large as the average throughput. Of course, fetching these instructions requires wide enough paths to the instruction

cache, but the most difficult aspect is handling branches. In this section we look at two methods for dealing with branches and then discuss how modern processors integrate the instruction prediction and prefetch functions.

Branch-Target Buffers

To reduce the branch penalty for our simple five-stage pipeline, as well as for deeper pipelines, we must know whether the as-yet-undecoded instruction is a branch and, if so, what the next PC should be. If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero. A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a *branch-target buffer* or *branch-target cache*. Figure 2.22 shows a branch-target buffer.

Because a branch-target buffer predicts the next instruction address and will send it out *before* decoding the instruction, we *must* know whether the fetched instruction is predicted as a taken branch. If the PC of the fetched instruction matches a PC in the prediction buffer, then the corresponding predicted PC is used as the next PC. The hardware for this branch-target buffer is essentially identical to the hardware for a cache.

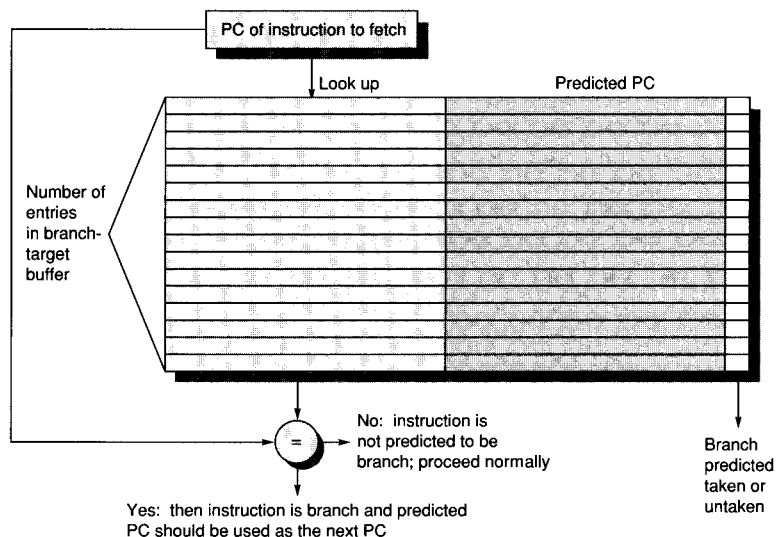


Figure 2.22 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

If a matching entry is found in the branch-target buffer, fetching begins immediately at the predicted PC. Note that unlike a branch-prediction buffer, the predictive entry must be matched to this instruction because the predicted PC will be sent out before it is known whether this instruction is even a branch. If the processor did not check whether the entry matched this PC, then the wrong PC would be sent out for instructions that were not branches, resulting in a slower processor. We only need to store the predicted-taken branches in the branch-target buffer, since an untaken branch should simply fetch the next sequential instruction, as if it were not a branch.

Figure 2.23 shows the detailed steps when using a branch-target buffer for a simple five-stage pipeline. From this we can see that there will be no branch delay if a branch-prediction entry is found in the buffer and the prediction is correct. Otherwise, there will be a penalty of at least 2 clock cycles. Dealing with the mispredictions and misses is a significant challenge, since we typically will have to halt instruction fetch while we rewrite the buffer entry. Thus, we would like to make this process fast to minimize the penalty.

To evaluate how well a branch-target buffer works, we first must determine the penalties in all possible cases. Figure 2.24 contains this information for the simple five-stage pipeline.

Example Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions from Figure 2.24. Make the following assumptions about the prediction accuracy and hit rate:

- Prediction accuracy is 90% (for instructions in the buffer).
- Hit rate in the buffer is 90% (for branches predicted taken).

Answer We compute the penalty by looking at the probability of two events: the branch is predicted taken but ends up being not taken, and the branch is taken but is not found in the buffer. Both carry a penalty of 2 cycles.

$$\begin{aligned}\text{Probability (branch in buffer, but actually not taken)} &= \text{Percent buffer hit rate} \times \text{Percent incorrect predictions} \\ &= 90\% \times 10\% = 0.09\end{aligned}$$

$$\text{Probability (branch not in buffer, but actually taken)} = 10\%$$

$$\text{Branch penalty} = (0.09 + 0.10) \times 2$$

$$\text{Branch penalty} = 0.38$$

This penalty compares with a branch penalty for delayed branches, which we evaluate in Appendix A, of about 0.5 clock cycles per branch. Remember, though, that the improvement from dynamic branch prediction will grow as the pipeline length and, hence, the branch delay grows; in addition, better predictors will yield a larger performance advantage.

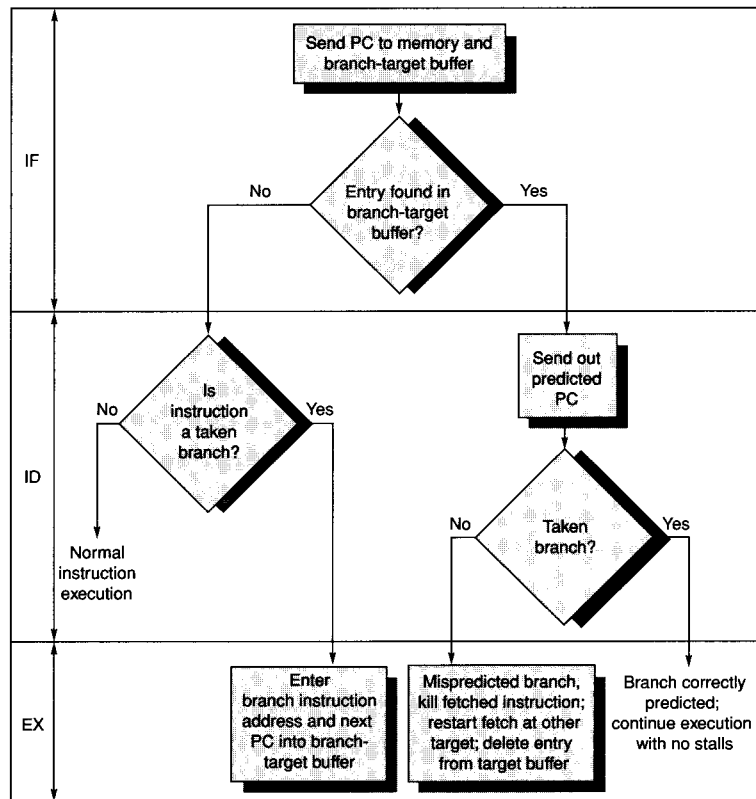


Figure 2.23 The steps involved in handling an instruction with a branch-target buffer.

Instruction in buffer	Prediction	Actual branch	Penalty cycles
yes	taken	taken	0
yes	taken	not taken	2
no		taken	2
no		not taken	0

Figure 2.24 Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer. There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer. If the branch is not correctly predicted, the penalty is equal to 1 clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and 1 clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and taken, a 2-cycle penalty is encountered, during which time the buffer is updated.

One variation on the branch-target buffer is to store one or more *target instructions* instead of, or in addition to, the predicted *target address*. This variation has two potential advantages. First, it allows the branch-target buffer access to take longer than the time between successive instruction fetches, possibly allowing a larger branch-target buffer. Second, buffering the actual target instructions allows us to perform an optimization called *branch folding*. Branch folding can be used to obtain 0-cycle unconditional branches, and sometimes 0-cycle conditional branches. Consider a branch-target buffer that buffers instructions from the predicted path and is being accessed with the address of an unconditional branch. The only function of the unconditional branch is to change the PC. Thus, when the branch-target buffer signals a hit and indicates that the branch is unconditional, the pipeline can simply substitute the instruction from the branch-target buffer in place of the instruction that is returned from the cache (which is the unconditional branch). If the processor is issuing multiple instructions per cycle, then the buffer will need to supply multiple instructions to obtain the maximum benefit. In some cases, it may be possible to eliminate the cost of a conditional branch when the condition codes are preset.

Return Address Predictors

As we try to increase the opportunity and accuracy of speculation we face the challenge of predicting indirect jumps, that is, jumps whose destination address varies at run time. Although high-level language programs will generate such jumps for indirect procedure calls, select or case statements, and FORTRAN-computed gotos, the vast majority of the indirect jumps come from procedure returns. For example, for the SPEC95 benchmarks, procedure returns account for more than 15% of the branches and the vast majority of the indirect jumps on average. For object-oriented languages like C++ and Java, procedure returns are even more frequent. Thus, focusing on procedure returns seems appropriate.

Though procedure returns can be predicted with a branch-target buffer, the accuracy of such a prediction technique can be low if the procedure is called from multiple sites and the calls from one site are not clustered in time. For example, in SPEC CPU95, an aggressive branch predictor achieves an accuracy of less than 60% for such return branches. To overcome this problem, some designs use a small buffer of return addresses operating as a stack. This structure caches the most recent return addresses: pushing a return address on the stack at a call and popping one off at a return. If the cache is sufficiently large (i.e., as large as the maximum call depth), it will predict the returns perfectly. Figure 2.25 shows the performance of such a return buffer with 0–16 elements for a number of the SPEC CPU95 benchmarks. We will use a similar return predictor when we examine the studies of ILP in Section 3.2.

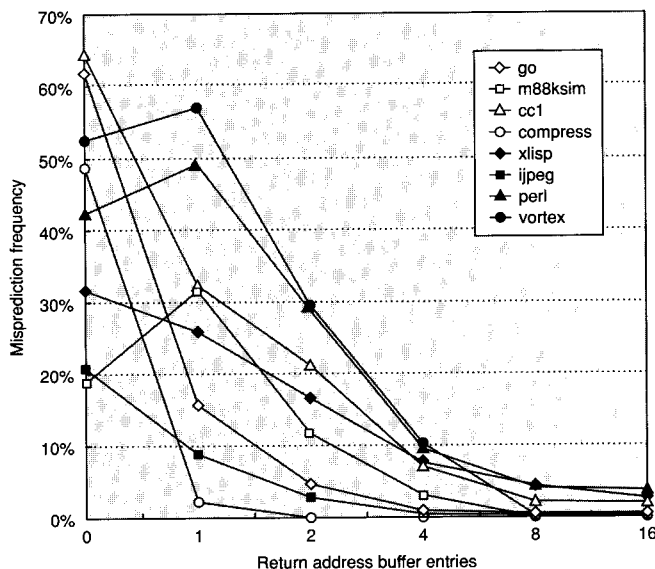


Figure 2.25 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Since call depths are typically not large, with some exceptions, a modest buffer works well. This data comes from Skadron et al. (1999), and uses a fix-up mechanism to prevent corruption of the cached return addresses.

Integrated Instruction Fetch Units

To meet the demands of multiple-issue processors, many recent designers have chosen to implement an integrated instruction fetch unit, as a separate autonomous unit that feeds instructions to the rest of the pipeline. Essentially, this amounts to recognizing that characterizing instruction fetch as a simple single pipe stage given the complexities of multiple issue is no longer valid.

Instead, recent designs have used an integrated instruction fetch unit that integrates several functions:

1. *Integrated branch prediction*—The branch predictor becomes part of the instruction fetch unit and is constantly predicting branches, so as to drive the fetch pipeline.
2. *Instruction prefetch*—To deliver multiple instructions per clock, the instruction fetch unit will likely need to fetch ahead. The unit autonomously manages the prefetching of instructions (see Chapter 5 for a discussion of techniques for doing this), integrating it with branch prediction.

3. *Instruction memory access and buffering*—When fetching multiple instructions per cycle a variety of complexities are encountered, including the difficulty that fetching multiple instructions may require accessing multiple cache lines. The instruction fetch unit encapsulates this complexity, using prefetch to try to hide the cost of crossing cache blocks. The instruction fetch unit also provides buffering, essentially acting as an on-demand unit to provide instructions to the issue stage as needed and in the quantity needed.

As designers try to increase the number of instructions executed per clock, instruction fetch will become an ever more significant bottleneck, and clever new ideas will be needed to deliver instructions at the necessary rate. One of the newer ideas, called *trace caches* and used in the Pentium 4, is discussed in Appendix C.

Speculation: Implementation Issues and Extensions

In this section we explore three issues that involve the implementation of speculation, starting with the use of register renaming, the approach that has almost totally replaced the use of a reorder buffer. We then discuss one important possible extension to speculation on control flow: an idea called value prediction.

Speculation Support: Register Renaming versus Reorder Buffers

One alternative to the use of a reorder buffer (ROB) is the explicit use of a larger physical set of registers combined with register renaming. This approach builds on the concept of renaming used in Tomasulo's algorithm and extends it. In Tomasulo's algorithm, the values of the *architecturally visible registers* (R0, . . . , R31 and F0, . . . , F31) are contained, at any point in execution, in some combination of the register set and the reservation stations. With the addition of speculation, register values may also temporarily reside in the ROB. In either case, if the processor does not issue new instructions for a period of time, all existing instructions will commit, and the register values will appear in the register file, which directly corresponds to the architecturally visible registers.

In the register-renaming approach, an extended set of physical registers is used to hold both the architecturally visible registers as well as temporary values. Thus, the extended registers replace the function of both the ROB and the reservation stations. During instruction issue, a renaming process maps the names of architectural registers to physical register numbers in the extended register set, allocating a new unused register for the destination. WAW and WAR hazards are avoided by renaming of the destination register, and speculation recovery is handled because a physical register holding an instruction destination does not become the architectural register until the instruction commits. The renaming map is a simple data structure that supplies the physical register number of the register that currently corresponds to the specified architectural register. This

structure is similar in structure and function to the register status table in Tomasulo's algorithm. When an instruction commits, the renaming table is permanently updated to indicate that a physical register corresponds to the actual architectural register, thus effectively finalizing the update to the processor state.

An advantage of the renaming approach versus the ROB approach is that instruction commit is simplified, since it requires only two simple actions: record that the mapping between an architectural register number and physical register number is no longer speculative, and free up any physical registers being used to hold the "older" value of the architectural register. In a design with reservation stations, a station is freed up when the instruction using it completes execution, and a ROB entry is freed up when the corresponding instruction commits.

With register renaming, deallocating registers is more complex, since before we free up a physical register, we must know that it no longer corresponds to an architectural register, and that no further uses of the physical register are outstanding. A physical register corresponds to an architectural register until the architectural register is rewritten, causing the renaming table to point elsewhere. That is, if no renaming entry points to a particular physical register, then it no longer corresponds to an architectural register. There may, however, still be uses of the physical register outstanding. The processor can determine whether this is the case by examining the source register specifiers of all instructions in the functional unit queues. If a given physical register does not appear as a source and it is not designated as an architectural register, it may be reclaimed and reallocated.

Alternatively, the processor can simply wait until another instruction that writes the same architectural register commits. At that point, there can be no further uses of the older value outstanding. Although this method may tie up a physical register slightly longer than necessary, it is easy to implement and hence is used in several recent superscalars.

One question you may be asking is, How do we ever know which registers are the architectural registers if they are constantly changing? Most of the time when the program is executing it does not matter. There are clearly cases, however, where another process, such as the operating system, must be able to know exactly where the contents of a certain architectural register reside. To understand how this capability is provided, assume the processor does not issue instructions for some period of time. Eventually all instructions in the pipeline will commit, and the mapping between the architecturally visible registers and physical registers will become stable. At that point, a subset of the physical registers contains the architecturally visible registers, and the value of any physical register not associated with an architectural register is unneeded. It is then easy to move the architectural registers to a fixed subset of physical registers so that the values can be communicated to another process.

Within the past few years most high-end superscalar processors, including the Pentium series, the MIPS R12000, and the Power and PowerPC processors, have chosen to use register renaming, adding from 20 to 80 extra registers. Since all results are allocated a new virtual register until they commit, these extra registers replace a primary function of the ROB and largely determine how many instructions may be in execution (between issue and commit) at one time.

How Much to Speculate

One of the significant advantages of speculation is its ability to uncover events that would otherwise stall the pipeline early, such as cache misses. This potential advantage, however, comes with a significant potential disadvantage. Speculation is not free: it takes time and energy, and the recovery of incorrect speculation further reduces performance. In addition, to support the higher instruction execution rate needed to benefit from speculation, the processor must have additional resources, which take silicon area and power. Finally, if speculation causes an exceptional event to occur, such as a cache or TLB miss, the potential for significant performance loss increases, if that event would not have occurred without speculation.

To maintain most of the advantage, while minimizing the disadvantages, most pipelines with speculation will allow only low-cost exceptional events (such as a first-level cache miss) to be handled in speculative mode. If an expensive exceptional event occurs, such as a second-level cache miss or a translation lookaside buffer (TLB) miss, the processor will wait until the instruction causing the event is no longer speculative before handling the event. Although this may slightly degrade the performance of some programs, it avoids significant performance losses in others, especially those that suffer from a high frequency of such events coupled with less-than-excellent branch prediction.

In the 1990s, the potential downsides of speculation were less obvious. As processors have evolved, the real costs of speculation have become more apparent, and the limitations of wider issue and speculation have been obvious. We return to this issue in the next chapter.

Speculating through Multiple Branches

In the examples we have considered in this chapter, it has been possible to resolve a branch before having to speculate on another. Three different situations can benefit from speculating on multiple branches simultaneously: a very high branch frequency, significant clustering of branches, and long delays in functional units. In the first two cases, achieving high performance may mean that multiple branches are speculated, and it may even mean handling more than one branch per clock. Database programs, and other less structured integer computations, often exhibit these properties, making speculation on multiple branches important. Likewise, long delays in functional units can raise the importance of speculating on multiple branches as a way to avoid stalls from the longer pipeline delays.

Speculating on multiple branches slightly complicates the process of speculation recovery, but is straightforward otherwise. A more complex technique is predicting and speculating on more than one branch per cycle. The IBM Power2 could resolve two branches per cycle but did not speculate on any other instructions. As of 2005, no processor has yet combined full speculation with resolving multiple branches per cycle.

Value Prediction

One technique for increasing the amount of ILP available in a program is value prediction. *Value prediction* attempts to predict the value that will be produced by an instruction. Obviously, since most instructions produce a different value every time they are executed (or at least a different value from a set of values), value prediction can have only limited success. There are, however, certain instructions for which it is easier to predict the resulting value—for example, loads that load from a constant pool, or that load a value that changes infrequently. In addition, when an instruction produces a value chosen from a small set of potential values, it may be possible to predict the resulting value by correlating it without an instance.

Value prediction is useful if it significantly increases the amount of available ILP. This possibility is most likely when a value is used as the source of a chain of dependent computations, such as a load. Because value prediction is used to enhance speculations and incorrect speculation has detrimental performance impact, the accuracy of the prediction is critical.

Much of the focus of research on value prediction has been on loads. We can estimate the maximum accuracy of a load value predictor by examining how often a load returns a value that matches a value returned in a recent execution of the load. The simplest case to examine is when the load returns a value that matches the value on the last execution of the load. For a range of SPEC CPU2000 benchmarks, this redundancy occurs from less than 5% of the time to almost 80% of the time. If we allow the load to match any of the most recent 16 values returned, the frequency of a potential match increases, and many benchmarks show a 80% match rate. Of course, matching 1 of 16 recent values does not tell you what value to predict, but it does mean that even with additional information it is impossible for prediction accuracy to exceed 80%.

Because of the high costs of misprediction and the likely case that misprediction rates will be significant (20% to 50%), researchers have focused on assessing which loads are more predictable and only attempting to predict those. This leads to a lower misprediction rate, but also fewer candidates for accelerating through prediction. In the limit, if we attempt to predict only those loads that always return the same value, it is likely that only 10% to 15% of the loads can be predicted. Research on value prediction continues. The results to date, however, have not been sufficiently compelling that any commercial processor has included the capability.

One simple idea that has been adopted and is related to value prediction is address aliasing prediction. *Address aliasing prediction* is a simple technique that predicts whether two stores or a load and a store refer to the same memory address. If two such references do not refer to the same address, then they may be safely interchanged. Otherwise, we must wait until the memory addresses accessed by the instructions are known. Because we need not actually predict the address values, only whether such values conflict, the prediction is both more stable and simpler. Hence, this limited form of address value speculation has been used by a few processors.

2.10

Putting It All Together: The Intel Pentium 4

The Pentium 4 is a processor with a deep pipeline supporting multiple issue with speculation. In this section, we describe the highlights of the Pentium 4 microarchitecture and examine its performance for the SPEC CPU benchmarks. The Pentium 4 also supports multithreading, a topic we discuss in the next chapter.

The Pentium 4 uses an aggressive out-of-order speculative microarchitecture, called Netburst, that is deeply pipelined with the goal of achieving high instruction throughput by combining multiple issue and high clock rates. Like the microarchitecture used in the Pentium III, a front-end decoder translates each IA-32 instruction to a series of micro-operations (uops), which are similar to typical RISC instructions. The uops are then executed by a dynamically scheduled speculative pipeline.

The Pentium 4 uses a novel *execution trace cache* to generate the uop instruction stream, as opposed to a conventional instruction cache that would hold IA-32 instructions. A *trace cache* is a type of instruction cache that holds sequences of instructions to be executed including nonadjacent instructions separated by branches; a trace cache tries to exploit the temporal sequencing of instruction execution rather than the spatial locality exploited in a normal cache; trace caches are explained in detail in Appendix C.

The Pentium 4's execution trace cache is a trace cache of uops, corresponding to the decoded IA-32 instruction stream. By filling the pipeline from the execution trace cache, the Pentium 4 avoids the need to redecode IA-32 instructions whenever the trace cache hits. Only on trace cache misses are IA-32 instructions fetched from the L2 cache and decoded to refill the execution trace cache. Up to three IA-32 instructions may be decoded and translated every cycle, generating up to six uops; when a single IA-32 instruction requires more than three uops, the uop sequence is generated from the microcode ROM.

The execution trace cache has its own branch target buffer, which predicts the outcome of uop branches. The high hit rate in the execution trace cache (for example, the trace cache miss rate for the SPEC CPUINT2000 benchmarks is less than 0.15%), means that the IA-32 instruction fetch and decode is rarely needed.

After fetching from the execution trace cache, the uops are executed by an out-of-order speculative pipeline, similar to that in Section 2.6, but using register renaming rather than a reorder buffer. Up to three uops per clock can be renamed and dispatched to the functional unit queues, and three uops can be committed each clock cycle. There are four dispatch ports, which allow a total of six uops to be dispatched to the functional units every clock cycle. The load and store units each have their own dispatch port, another port covers basic ALU operations, and a fourth handles FP and integer operations. Figure 2.26 shows a diagram of the microarchitecture.

Since the Pentium 4 microarchitecture is dynamically scheduled, uops do not follow a simple static set of pipeline stages during their execution. Instead various stages of execution (instruction fetch, decode, uop issue, rename, schedule, execute, and retire) can take varying numbers of clock cycles. In the Pentium III,

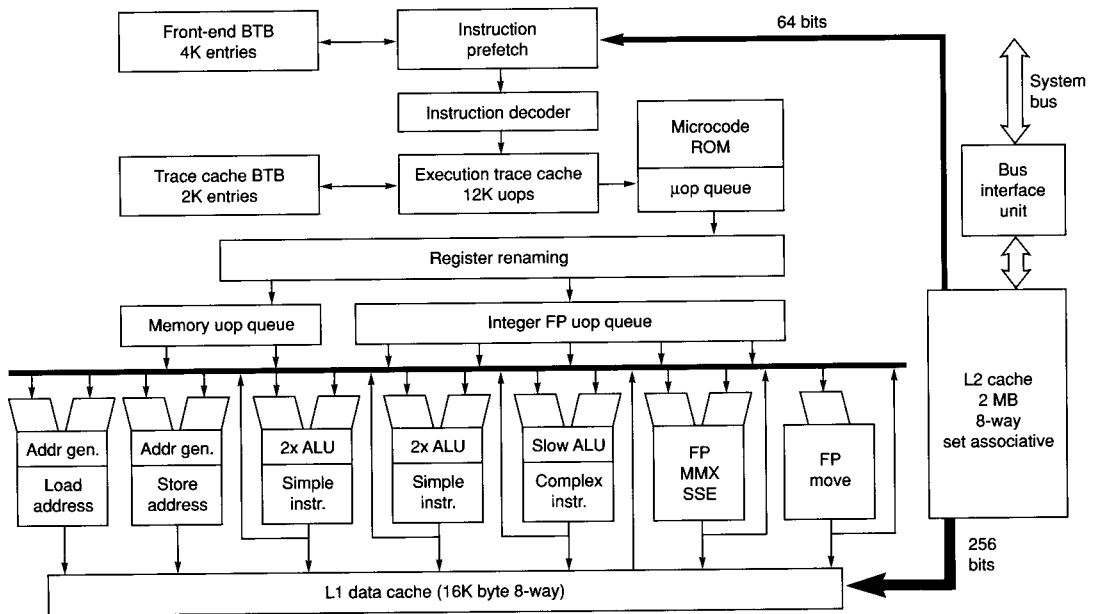


Figure 2.26 The Pentium 4 microarchitecture. The cache sizes represent the Pentium 4 640. Note that the instructions are usually coming from the trace cache; only when the trace cache misses is the front-end instruction prefetch unit consulted. This figure was adapted from Boggs et al. [2004].

the minimum time for an instruction to go from fetch to retire was 11 clock cycles, with instructions requiring multiple clock cycles in the execution stage taking longer. As in any dynamically scheduled pipeline, instructions could take much longer if they had to wait for operands. As stated earlier, the Pentium 4 introduced a much deeper pipeline, partitioning stages of the Pentium III pipeline so as to achieve a higher clock rate. In the initial Pentium 4 introduced in 1990, the minimum number of cycles to transit the pipeline was increased to 21, allowing for a 1.5 GHz clock rate. In 2004, Intel introduced a version of the Pentium 4 with a 3.2 GHz clock rate. To achieve this high clock rate, further pipelining was added so that a simple instruction takes 31 clock cycles to go from fetch to retire. This additional pipelining, together with improvements in transistor speed, allowed the clock rate to more than double over the first Pentium 4.

Obviously, with such deep pipelines and aggressive clock rates the cost of cache misses and branch mispredictions are both very high. A two-level cache is used to minimize the frequency of DRAM accesses. Branch prediction is done with a branch-target buffer using a two-level predictor with both local and global branch histories; in the most recent Pentium 4, the size of the branch-target buffer was increased, and the static predictor, used when the branch-target buffer misses, was improved. Figure 2.27 summarizes key features of the microarchitecture, and the caption notes some of the changes since the first version of the Pentium 4 in 2000.

Feature	Size	Comments
Front-end branch-target buffer	4K entries	Predicts the next IA-32 instruction to fetch; used only when the execution trace cache misses.
Execution trace cache	12K uops	Trace cache used for uops.
Trace cache branch-target buffer	2K entries	Predicts the next uop.
Registers for renaming	128 total	128 uops can be in execution with up to 48 loads and 32 stores.
Functional units	7 total: 2 simple ALU, complex ALU, load, store, FP move, FP arithmetic	The simple ALU units run at twice the clock rate, accepting up to two simple ALU uops every clock cycle. This allows execution of two dependent ALU operations in a single clock cycle.
L1 data cache	16 KB; 8-way associative; 64-byte blocks write through	Integer load to use latency is 4 cycles; FP load to use latency is 12 cycles; up to 8 outstanding load misses.
L2 cache	2 MB; 8-way associative; 128-byte blocks write back	256 bits to L1, providing 108 GB/sec; 18-cycle access time; 64 bits to memory capable of 6.4 GB/sec. A miss in L2 does not cause an automatic update of L1.

Figure 2.27 Important characteristics of the recent Pentium 4 640 implementation in 90 nm technology (code named Prescott). The newer Pentium 4 uses larger caches and branch-prediction buffers, allows more loads and stores outstanding, and has higher bandwidth between levels in the memory system. Note the novel use of double-speed ALUs, which allow the execution of back-to-back dependent ALU operations in a single clock cycle; having twice as many ALUs, an alternative design point, would not allow this capability. The original Pentium 4 used a trace cache BTB with 512 entries, an L1 cache of 8 KB, and an L2 cache of 256 KB.

An Analysis of the Performance of the Pentium 4

The deep pipeline of the Pentium 4 makes the use of speculation, and its dependence on branch prediction, critical to achieving high performance. Likewise, performance is very dependent on the memory system. Although dynamic scheduling and the large number of outstanding loads and stores supports hiding the latency of cache misses, the aggressive 3.2 GHz clock rate means that L2 misses are likely to cause a stall as the queues fill up while awaiting the completion of the miss.

Because of the importance of branch prediction and cache misses, we focus our attention on these two areas. The charts in this section use five of the integer SPEC CPU2000 benchmarks and five of the FP benchmarks, and the data is captured using counters within the Pentium 4 designed for performance monitoring. The processor is a Pentium 4 640 running at 3.2 GHz with an 800 MHz system bus and 667 MHz DDR2 DRAMs for main memory.

Figure 2.28 shows the branch-misprediction rate in terms of mispredictions per 1000 instructions. Remember that in terms of pipeline performance, what matters is the number of mispredictions per instruction; the FP benchmarks generally have fewer branches per instruction (48 branches per 1000 instructions) versus the integer benchmarks (186 branches per 1000 instructions), as well as

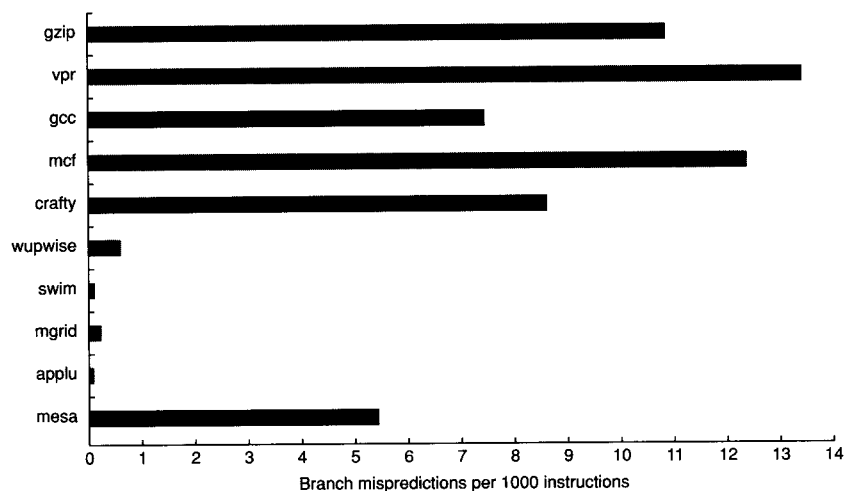


Figure 2.28 Branch misprediction rate per 1000 instructions for five integer and five floating-point benchmarks from the SPEC CPU2000 benchmark suite. This data and the rest of the data in this section were acquired by John Holm and Dileep Bhandarkar of Intel.

better prediction rates (98% versus 94%). The result, as Figure 2.28 shows, is that the misprediction rate per instruction for the integer benchmarks is more than 8 times higher than the rate for the FP benchmarks.

Branch-prediction accuracy is crucial in speculative processors, since incorrect speculation requires recovery time and wastes energy pursuing the wrong path. Figure 2.29 shows the fraction of executed uops that are the result of misspeculation. As we would suspect, the misspeculation rate results look almost identical to the misprediction rates.

How do the cache miss rates contribute to possible performance losses? The trace cache miss rate is almost negligible for this set of the SPEC benchmarks, with only one benchmark (186.craft) showing any significant misses (0.6%). The L1 and L2 miss rates are more significant. Figure 2.30 shows the L1 and L2 miss rates for these 10 benchmarks. Although the miss rate for L1 is about 14 times higher than the miss rate for L2, the miss penalty for L2 is comparably higher, and the inability of the microarchitecture to hide these very long misses means that L2 misses likely are responsible for an equal or greater performance loss than L1 misses, especially for benchmarks such as mcf and swim.

How do the effects of misspeculation and cache misses translate to actual performance? Figure 2.31 shows the effective CPI for the 10 SPEC CPU2000 benchmarks. There are three benchmarks whose performance stands out from the pack and are worth examining:

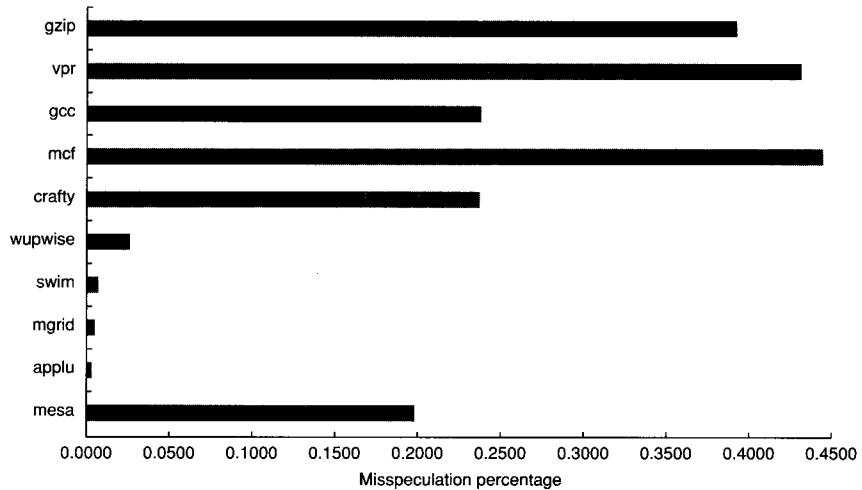


Figure 2.29 The percentage of uop instructions issued that are misspeculated.

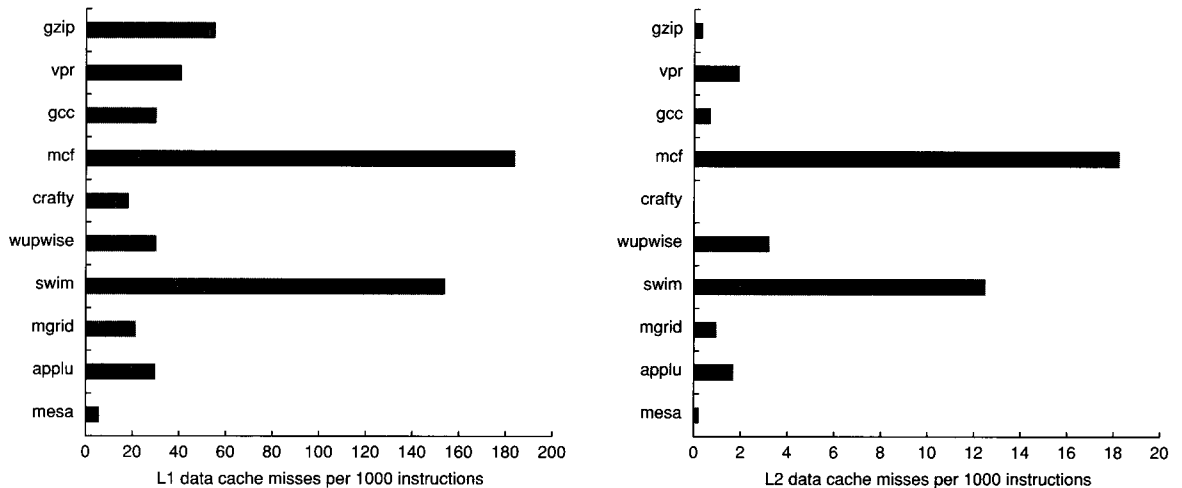


Figure 2.30 L1 data cache and L2 cache misses per 1000 instructions for 10 SPEC CPU2000 benchmarks. Note that the scale of the L1 misses is 10 times that of the L2 misses. Because the miss penalty for L2 is likely to be at least 10 times larger than for L1, the relative sizes of the bars are an indication of the relative performance penalty for the misses in each cache. The inability to hide long L2 misses with overlapping execution will further increase the stalls caused by L2 misses relative to L1 misses.

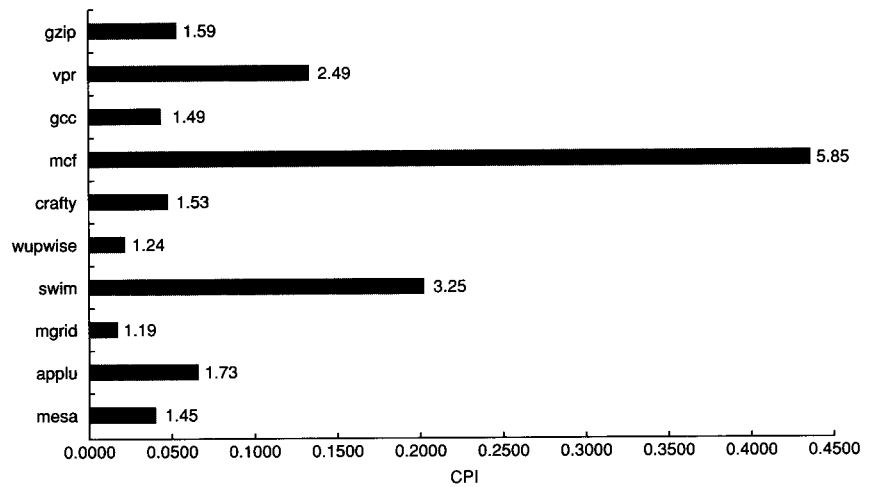


Figure 2.31 The CPI for the 10 SPEC CPU benchmarks. An increase in the CPI by a factor of 1.29 comes from the translation of IA-32 instructions into uops, which results in 1.29 uops per IA-32 instruction on average for these 10 benchmarks.

1. mcf has a CPI that is more than four times higher than that of the four other integer benchmarks. It has the worst misspeculation rate. Equally importantly, mcf has the worst L1 and the worst L2 miss rate among any benchmark, integer or floating point, in the SPEC suite. The high cache miss rates make it impossible for the processor to hide significant amounts of miss latency.
2. vpr achieves a CPI that is 1.6 times higher than three of the five integer benchmarks (excluding mcf). This appears to arise from a branch misprediction that is the worst among the integer benchmarks (although not much worse than the average) together with a high L2 miss rate, second only to mcf among the integer benchmarks.
3. swim is the lowest performing FP benchmark, with a CPI that is more than two times the average of the other four FP benchmarks. swim's problems are high L1 and L2 cache miss rates, second only to mcf. Notice that swim has excellent speculation results, but that success can probably not hide the high miss rates, especially in L2. In contrast, several benchmarks with reasonable L1 miss rates and low L2 miss rates (such as mgrid and gzip) perform well.

To close this section, let's look at the relative performance of the Pentium 4 and AMD Opteron for this subset of the SPEC benchmarks. The AMD Opteron and Intel Pentium 4 share a number of similarities:

- Both use a dynamically scheduled, speculative pipeline capable of issuing and committing three IA-32 instructions per clock.
- Both use a two-level on-chip cache structure, although the Pentium 4 uses a trace cache for the first-level instruction cache and recent Pentium 4 implementations have larger second-level caches.
- They have similar transistor counts, die size, and power, with the Pentium 4 being about 7% to 10% higher on all three measures at the highest clock rates available in 2005 for these two processors.

The most significant difference is the very deep pipeline of the Intel Netburst microarchitecture, which was designed to allow higher clock rates. Although compilers optimized for the two architectures produce slightly different code sequences, comparing CPI measures can provide important insights into how these two processors compare. Remember that differences in the memory hierarchy as well as differences in the pipeline structure will affect these measurements; we analyze the differences in memory system performance in Chapter 5. Figure 2.32 shows the CPI measures for a set of SPEC CPU2000 benchmarks for a 3.2 GHz Pentium 4 and a 2.6 GHz AMD Opteron. At these clock rates, the Opteron processor has an average improvement in CPI by 1.27 over the Pentium 4.

Of course, we should expect the Pentium 4, with its much deeper pipeline, to have a somewhat higher CPI than the AMD Opteron. The key question for the very deeply pipelined Netburst design is whether the increase in clock rate, which the deeper pipelining allows, overcomes the disadvantages of a higher

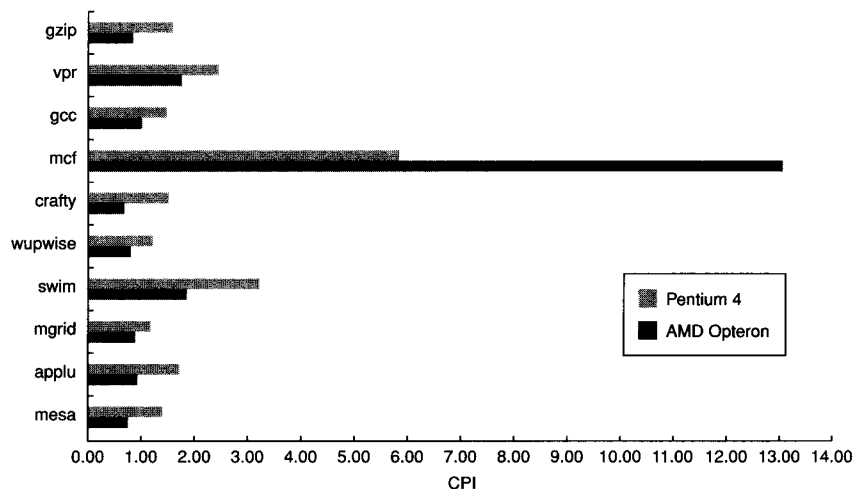


Figure 2.32 A 2.6 GHz AMD Opteron has a lower CPI by a factor of 1.27 versus a 3.2 GHz Pentium 4.

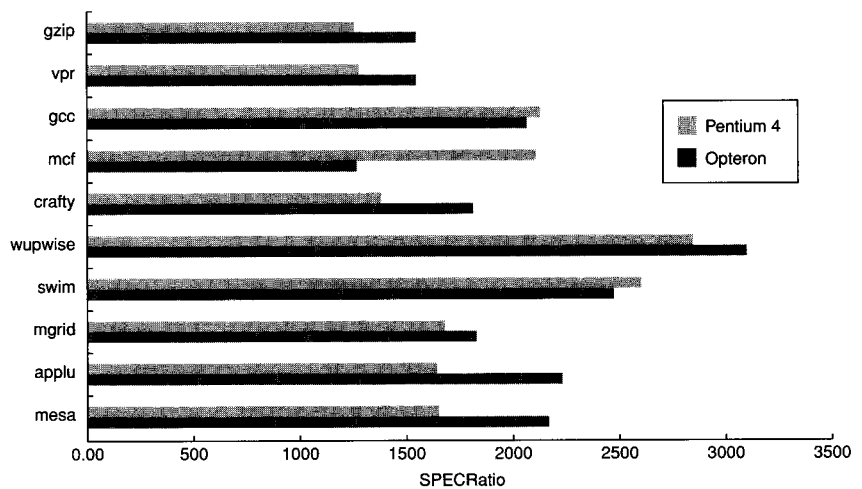


Figure 2.33 The performance of a 2.8 GHz AMD Opteron versus a 3.8 GHz Intel Pentium 4 shows a performance advantage for the Opteron of about 1.08.

CPI. We examine this by showing the SPEC CPU2000 performance for these two processors at their highest available clock rate of these processors in 2005: 2.8 GHz for the Opteron and 3.8 GHz for the Pentium 4. These higher clock rates will increase the effective CPI measurement versus those in Figure 2.32, since the cost of a cache miss will increase. Figure 2.33 shows the relative performance on the same subset of SPEC as Figure 2.32. The Opteron is slightly faster, meaning that the higher clock rate of the Pentium 4 is insufficient to overcome the higher CPI arising from more pipeline stalls.

Hence, while the Pentium 4 performs well, it is clear that the attempt to achieve both high clock rates via a deep pipeline and high instruction throughput via multiple issue is not as successful as the designers once believed it would be. We discuss this topic in depth in the next chapter.

2.11

Fallacies and Pitfalls

Our first fallacy has two parts: First, simple rules do not hold, and, second, the choice of benchmarks plays a major role.

Fallacy *Processors with lower CPIs will always be faster.*

Fallacy *Processors with faster clock rates will always be faster.*

Although a lower CPI is certainly better, sophisticated multiple-issue pipelines typically have slower clock rates than processors with simple pipelines. In appli-

cations with limited ILP or where the parallelism cannot be exploited by the hardware resources, the faster clock rate often wins. But, when significant ILP exists, a processor that exploits lots of ILP may be better.

The IBM Power5 processor is designed for high-performance integer and FP; it contains two processor cores each capable of sustaining four instructions per clock, including two FP and two load-store instructions. The highest clock rate for a Power5 processor in 2005 is 1.9 GHz. In comparison, the Pentium 4 offers a single processor with multithreading (see the next chapter). The processor can sustain three instructions per clock with a very deep pipeline, and the maximum available clock rate in 2005 is 3.8 GHz.

Thus, the Power5 will be faster if the product of the instruction count and CPI is less than one-half the same product for the Pentium 4. As Figure 2.34 shows the $\text{CPI} \times \text{instruction count}$ advantages of the Power5 are significant for the FP programs, sometimes by more than a factor of 2, while for the integer programs the $\text{CPI} \times \text{instruction count}$ advantage of the Power5 is usually not enough to overcome the clock rate advantage of the Pentium 4. By comparing the SPEC numbers, we find that the product of instruction count and CPI advantage for the Power5 is 3.1 times on the floating-point programs but only 1.5 times on the integer programs. Because the maximum clock rate of the Pentium 4 in 2005 is exactly twice that of the Power5, the Power5 is faster by 1.5 on SPECfp2000 and the Pentium 4 will be faster by 1.3 on SPECint2000.

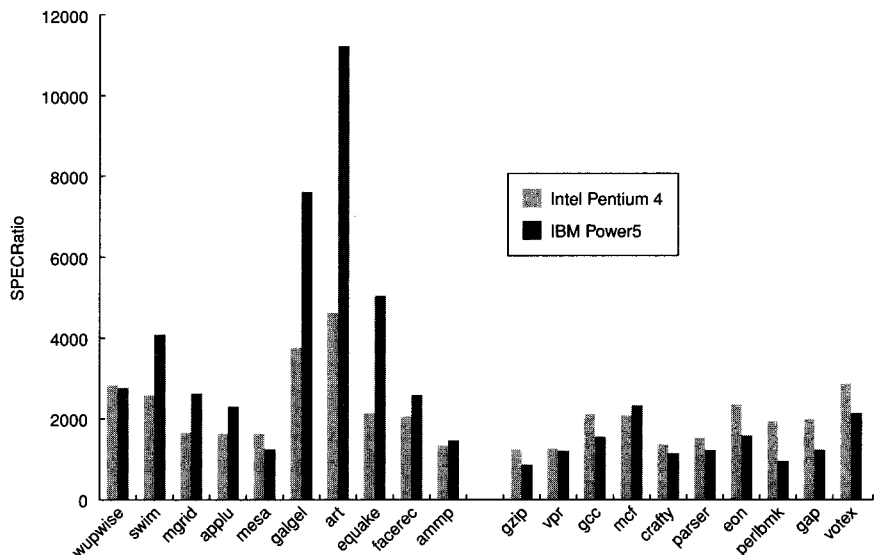


Figure 2.34 A comparison of the 1.9 GHz IBM Power5 processor versus the 3.8 GHz Intel Pentium 4 for 20 SPEC benchmarks (10 integer on the left and 10 floating point on the right) shows that the higher clock Pentium 4 is generally faster for the integer workload, while the lower CPI Power5 is usually faster for the floating-point workload.

Pitfall *Sometimes bigger and dumber is better.*

Advanced pipelines have focused on novel and increasingly sophisticated schemes for improving CPI. The 21264 uses a sophisticated tournament predictor with a total of 29K bits (see page 88), while the earlier 21164 uses a simple 2-bit predictor with 2K entries (or a total of 4K bits). For the SPEC95 benchmarks, the more sophisticated branch predictor of the 21264 outperforms the simpler 2-bit scheme on all but one benchmark. On average, for SPECint95, the 21264 has 11.5 mispredictions per 1000 instructions committed, while the 21164 has about 16.5 mispredictions.

Somewhat surprisingly, the simpler 2-bit scheme works better for the transaction-processing workload than the sophisticated 21264 scheme (17 mispredictions versus 19 per 1000 completed instructions)! How can a predictor with less than 1/7 the number of bits and a much simpler scheme actually work better? The answer lies in the structure of the workload. The transaction-processing workload has a very large code size (more than an order of magnitude larger than any SPEC95 benchmark) with a large branch frequency. The ability of the 21164 predictor to hold twice as many branch predictions based on purely local behavior (2K versus the 1K local predictor in the 21264) seems to provide a slight advantage.

This pitfall also reminds us that different applications can produce different behaviors. As processors become more sophisticated, including specific microarchitectural features aimed at some particular program behavior, it is likely that different applications will see more divergent behavior.

2.12

Concluding Remarks

The tremendous interest in multiple-issue organizations came about because of an interest in improving performance without affecting the standard uniprocessor programming model. Although taking advantage of ILP is conceptually simple, the design problems are amazingly complex in practice. It is extremely difficult to achieve the performance you might expect from a simple first-level analysis.

Rather than embracing dramatic new approaches in microarchitecture, most of the last 10 years have focused on raising the clock rates of multiple-issue processors and narrowing the gap between peak and sustained performance. The dynamically scheduled, multiple-issue processors announced in the last five years (the Pentium 4, IBM Power5, and the AMD Athlon and Opteron) have the same basic structure and similar sustained issue rates (three to four instructions per clock) as the first dynamically scheduled, multiple-issue processors announced in 1995! But the clock rates are 10–20 times higher, the caches are 4–8 times bigger, there are 2–4 times as many renaming registers, and twice as many load-store units! The result is performance that is 8–16 times higher.

The trade-offs between increasing clock speed and decreasing CPI through multiple issue are extremely hard to quantify. In the 1995 edition of this book, we stated:

Although you might expect that it is possible to build an advanced multiple-issue processor with a high clock rate, a factor of 1.5 to 2 in clock rate has consistently separated the highest clock rate processors and the most sophisticated multiple-issue processors. It is simply too early to tell whether this difference is due to fundamental implementation trade-offs, or to the difficulty of dealing with the complexities in multiple-issue processors, or simply a lack of experience in implementing such processors.

Given the availability of the Pentium 4 at 3.8 GHz, it has become clear that the limitation was primarily our understanding of how to build such processors. As we will see in the next chapter, however, it appears unclear that the initial success in achieving high-clock-rate processors that issue three to four instructions per clock can be carried much further due to limitations in available ILP, efficiency in exploiting that ILP, and power concerns. In addition, as we saw in the comparison of the Opteron and Pentium 4, it appears that the performance advantage in high clock rates achieved by very deep pipelines (20–30 stages) is largely lost by additional pipeline stalls. We analyze this behavior further in the next chapter.

One insight that was clear in 1995 and has become even more obvious in 2005 is that the peak-to-sustained performance ratios for multiple-issue processors are often quite large and typically grow as the issue rate grows. The lessons to be gleaned by comparing the Power5 and Pentium 4, or the Pentium 4 and Pentium III (which differ primarily in pipeline depth and hence clock rate, rather than issue rates), remind us that it is difficult to generalize about clock rate versus CPI, or about the underlying trade-offs in pipeline depth, issue rate, and other characteristics.

A change in approach is clearly upon us. Higher-clock-rate versions of the Pentium 4 have been abandoned. IBM has shifted to putting two processors on a single chip in the Power4 and Power5 series, and both Intel and AMD have delivered early versions of two-processor chips. We will return to this topic in the next chapter and indicate why the 20-year rapid pursuit of ILP seems to have reached its end.

2.13

Historical Perspective and References

Section K.4 on the companion CD features a discussion on the development of pipelining and instruction-level parallelism. We provide numerous references for further reading and exploration of these topics.



Case Studies with Exercises by Robert P. Colwell

Case Study 1: Exploring the Impact of Microarchitectural Techniques

Concepts illustrated by this case study

- Basic Instruction Scheduling, Reordering, Dispatch
- Multiple Issue and Hazards
- Register Renaming
- Out-of-Order and Speculative Execution
- Where to Spend Out-of-Order Resources

You are tasked with designing a new processor microarchitecture, and you are trying to figure out how best to allocate your hardware resources. Which of the hardware and software techniques you learned in Chapter 2 should you apply? You have a list of latencies for the functional units and for memory, as well as some representative code. Your boss has been somewhat vague about the performance requirements of your new design, but you know from experience that, all else being equal, faster is usually better. Start with the basics. Figure 2.35 provides a sequence of instructions and list of latencies.

- 2.1 [10] <1.8, 2.1, 2.2> What would be the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 2.35 if no new instruction execution could be initiated until the previous instruction execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a 1 cycle branch delay slot.

		Latencies beyond single cycle	
Loop:	LD F2,0(Rx)	Memory LD	+3
I0:	MULTD F2,F0,F2	Memory SD	+1
I1:	DIVD F8,F2,F0	Integer ADD, SUB	+0
I2:	LD F4,0(Ry)	Branches	+1
I3:	ADD F4,F0,F4	ADD	+2
I4:	ADD F10,F8,F2	MULTD	+4
I5:	SD F4,0(Ry)	DIVD	+10
I6:	ADDI Rx,Rx,#8		
I7:	ADDI Ry,Ry,#8		
I8:	SUB R20,R4,Rx		
I9:	BNZ R20,Loop		

Figure 2.35 Code and latencies for Exercises 2.1 through 2.6.

- 2.2 [10] <1.8, 2.1, 2.2> Think about what latency numbers really mean—they indicate the number of cycles a given function requires to produce its output, nothing more. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a “producer” followed by a “consumer”) will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 2.35 require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? Show the code with `<stall>` inserted where necessary to accommodate stated latencies. (*Hint:* An instruction with latency “+2” needs 2 `<stall>` cycles to be inserted into the code sequence. Think of it this way: a 1-cycle instruction has latency 1 + 0, meaning zero extra wait states. So latency 1 + 1 implies 1 stall cycle; latency 1 + N has N extra stall cycles.)
- 2.3 [15] <2.6, 2.7> Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependence. Now how many cycles does the loop require?
- 2.4 [10] <2.6, 2.7> In the multiple-issue design of Exercise 2.3, you may have recognized some subtle issues. Even though the two pipelines have the exact same instruction repertoire, they are not identical nor interchangeable, because there is an implicit ordering between them that must reflect the ordering of the instructions in the original program. If instruction $N + 1$ begins execution in Execution Pipe 1 at the same time that instruction N begins in Pipe 0, and $N + 1$ happens to require a shorter execution latency than N , then $N + 1$ will complete before N (even though program ordering would have implied otherwise). Recite at least two reasons why that could be hazardous and will require special considerations in the microarchitecture. Give an example of two instructions from the code in Figure 2.35 that demonstrate this hazard.
- 2.5 [20] <2.7> Reorder the instructions to improve performance of the code in Figure 2.35. Assume the two-pipe machine in Exercise 2.3, and that the out-of-order completion issues of Exercise 2.4 have been dealt with successfully. Just worry about observing true data dependences and functional unit latencies for now. How many cycles does your reordered code take?
- 2.6 [10/10] <2.1, 2.2> Every cycle that does not initiate a new operation in a pipe is a lost opportunity, in the sense that your hardware is not “living up to its potential.”
- [10] <2.1, 2.2> In your reordered code from Exercise 2.5, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?
 - [10] <2.1, 2.2> Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance.

- c. Hand-unroll two iterations of the loop in your reordered code from Exercise 2.5. What speedup did you obtain? (For this exercise, just color the $N + 1$ iteration's instructions green to distinguish them from the N th iteration's; if you were actually unrolling the loop you would have to reassign registers to prevent collisions between the iterations.)
- 2.7 [15] <2.1> Computers spend most of their time in loops, so multiple loop iterations are great places to speculatively find more work to keep CPU resources busy. Nothing is ever easy, though; the compiler emitted only one copy of that loop's code, so even though multiple iterations are handling distinct data, they will appear to use the same registers. To keep register usages multiple iterations from colliding, we rename their registers. Figure 2.36 shows example code that we would like our hardware to rename.

A compiler could have simply unrolled the loop and used different registers to avoid conflicts, but if we expect our hardware to unroll the loop, it must also do the register renaming. How? Assume your hardware has a pool of temporary registers (call them T registers, and assume there are 64 of them, T0 through T63) that it can substitute for those registers designated by the compiler. This rename hardware is indexed by the source register designation, and the value in the table is the T register of the last destination that targeted that register. (Think of these table values as producers, and the src registers are the consumers; it doesn't much matter where the producer puts its result as long as its consumers can find it.) Consider the code sequence in Figure 2.36. Every time you see a destination register in the code, substitute the next available T, beginning with T9. Then update all the src registers accordingly, so that true data dependences are maintained. Show the resulting code. (*Hint:* See Figure 2.37.)

```

Loop: LD      F2,0(Rx)
I0:  MULTD   F5,F0,F2
I1:  DIVD    F8,F0,F2
I2:  LD      F4,0(Ry)
I3:  ADDD    F6,F0,F4
I4:  ADDD    F10,F8,F2
I5:  SD      F4,0(Ry)

```

Figure 2.36 Sample code for register renaming practice.

```

I0:  LD      T9,0(Rx)
I1:  MULTD   T10,F0,T9
. . .

```

Figure 2.37 Expected output of register renaming.

```

I0:  MULTD  F5,F0,F2
I1:  ADDD   F9,F5,F4
I2:  ADDD   F5,F5,F2
I3:  DIVD   F2,F9,F0

```

Figure 2.38 Sample code for superscalar register renaming.

- 2.8 [20] <2.4> Exercise 2.7 explored simple register renaming: when the hardware register renamer sees a source register, it substitutes the destination T register of the last instruction to have targeted that source register. When the rename table sees a destination register, it substitutes the next available T for it. But superscalar designs need to handle multiple instructions per clock cycle at every stage in the machine, including the register renaming. A simple scalar processor would therefore look up both src register mappings for each instruction, and allocate a new destination mapping per clock cycle. Superscalar processors must be able to do that as well, but they must also ensure that any dest-to-src relationships between the two concurrent instructions are handled correctly. Consider the sample code sequence in Figure 2.38. Assume that we would like to simultaneously rename the first two instructions. Further assume that the next two available T registers to be used are known at the beginning of the clock cycle in which these two instructions are being renamed. Conceptually, what we want is for the first instruction to do its rename table lookups, and then update the table per its destination's T register. Then the second instruction would do exactly the same thing, and any inter-instruction dependency would thereby be handled correctly. But there's not enough time to write that T register designation into the renaming table and then look it up again for the second instruction, all in the same clock cycle. That register substitution must instead be done live (in parallel with the register rename table update). Figure 2.39 shows a circuit diagram, using multiplexers and comparators, that will accomplish the necessary on-the-fly register renaming. Your task is to show the cycle-by-cycle state of the rename table for every instruction of the code. Assume the table starts out with every entry equal to its index (T0 = 0; T1 = 1, ...).
- 2.9 [5] <2.4> If you ever get confused about what a register renamer has to do, go back to the assembly code you're executing, and ask yourself what has to happen for the right result to be obtained. For example, consider a three-way superscalar machine renaming these three instructions concurrently:

```

ADDI    R1, R1, R1
ADDI    R1, R1, R1
ADDI    R1, R1, R1

```

If the value of R1 starts out as 5, what should its value be when this sequence has executed?

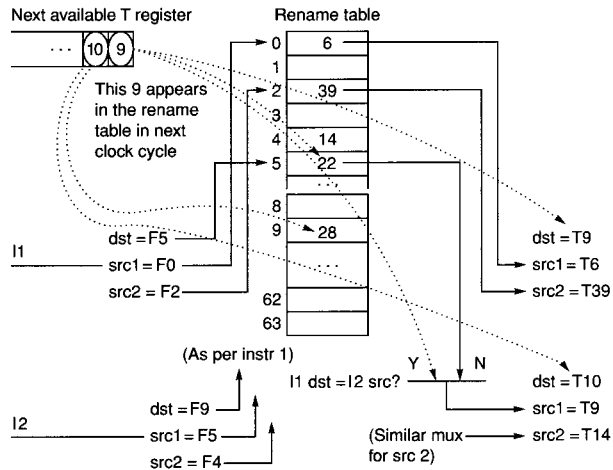


Figure 2.39 Rename table and on-the-fly register substitution logic for superscalar machines. (Note: “src” is source, “dst” is destination.)

```

Loop: LW    R1,0(R2) ; LW    R3,8(R2)
      <stall>
      <stall>
      ADDI   R10,R1,#1; ADDI   R11,R3,#1
      SW     R1,0(R2) ; SW     R3,8(R2)
      ADDI   R2,R2,#8
      SUB     R4,R3,R2
      BNZ    R4,Loop
  
```

Figure 2.40 Sample VLIW code with two adds, two loads, and two stalls.

- 2.10 [20] <2.4, 2.9> VLIW designers have a few basic choices to make regarding architectural rules for register use. Suppose a VLIW is designed with self-draining execution pipelines: once an operation is initiated, its results will appear in the destination register at most L cycles later (where L is the latency of the operation). There are never enough registers, so there is a temptation to wring maximum use out of the registers that exist. Consider Figure 2.40. If loads have a 1 + 2 cycle latency, unroll this loop once, and show how a VLIW capable of two loads and two adds per cycle can use the minimum number of registers, in the absence of any pipeline interruptions or stalls. Give an example of an event that, in the presence of self-draining pipelines, could disrupt this pipelining and yield wrong results.

- 2.11 [10/10/10] <2.3> Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write back) and the code in Figure 2.41. All ops are 1 cycle except LW and SW, which are 1 + 2 cycles, and branches, which are 1 + 1 cycles. There is no forwarding. Show the phases of each instruction per clock cycle for one iteration of the loop.
- [10] <2.3> How many clock cycles per loop iteration are lost to branch overhead?
 - [10] <2.3> Assume a static branch predictor, capable of recognizing a backwards branch in the decode stage. Now how many clock cycles are wasted on branch overhead?
 - [10] <2.3> Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?
- 2.12 [20/20/20/10/20] <2.4, 2.7, 2.10> Let's consider what dynamic scheduling might achieve here. Assume a microarchitecture as shown in Figure 2.42. Assume that the ALUs can do all arithmetic ops (MULTD, DIVD, ADDD, ADDI, SUB) and branches, and that the Reservation Station (RS) can dispatch at most one operation to each functional unit per cycle (one op to each ALU plus one memory op to the LD/ST unit).

```

Loop: LW    R1,0(R2)
      ADDI  R1,R1,#1
      SW    R1,0(R2)
      ADDI  R2,R2,#4
      SUB   R4,R3,R2
      BNZ   R4,Loop
  
```

Figure 2.41 Code loop for Exercise 2.11.

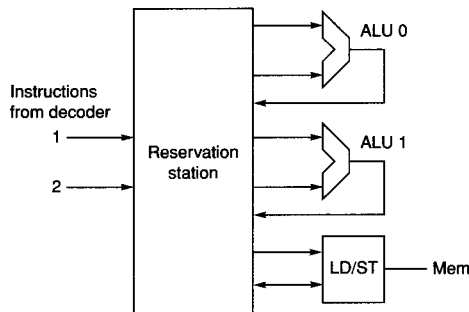


Figure 2.42 An out-of-order microarchitecture.

- a. [15] <2.4> Suppose all of the instructions from the sequence in Figure 2.35 are present in the RS, with no renaming having been done. Highlight any instructions in the code where register renaming would improve performance. *Hint:* Look for RAW and WAW hazards. Assume the same functional unit latencies as in Figure 2.35.
- b. [20] <2.4> Suppose the register-renamed version of the code from part (a) is resident in the RS in clock cycle N , with latencies as given in Figure 2.35. Show how the RS should dispatch these instructions out-of-order, clock by clock, to obtain optimal performance on this code. (Assume the same RS restrictions as in part (a). Also assume that results must be written into the RS before they're available for use; i.e., no bypassing.) How many clock cycles does the code sequence take?
- c. [20] <2.4> Part (b) lets the RS try to optimally schedule these instructions. But in reality, the whole instruction sequence of interest is not usually present in the RS. Instead, various events clear the RS, and as a new code sequence streams in from the decoder, the RS must choose to dispatch what it has. Suppose that the RS is empty. In cycle 0 the first two register-renamed instructions of this sequence appear in the RS. Assume it takes 1 clock cycle to dispatch any op, and assume functional unit latencies are as they were for Exercise 2.2. Further assume that the front end (decoder/register-renamer) will continue to supply two new instructions per clock cycle. Show the cycle-by-cycle order of dispatch of the RS. How many clock cycles does this code sequence require now?
- d. [10] <2.10> If you wanted to improve the results of part (c), which would have helped most: (1) another ALU; (2) another LD/ST unit; (3) full bypassing of ALU results to subsequent operations; (4) cutting the longest latency in half? What's the speedup?
- e. [20] <2.7> Now let's consider speculation, the act of fetching, decoding, and executing beyond one or more conditional branches. Our motivation to do this is twofold: the dispatch schedule we came up with in part (c) had lots of nops, and we know computers spend most of their time executing loops (which implies the branch back to the top of the loop is pretty predictable.) Loops tell us where to find more work to do; our sparse dispatch schedule suggests we have opportunities to do some of that work earlier than before. In part (d) you found the critical path through the loop. Imagine folding a second copy of that path onto the schedule you got in part (b). How many more clock cycles would be required to do two loops' worth of work (assuming all instructions are resident in the RS)? (Assume all functional units are fully pipelined.)

Case Study 2: Modeling a Branch Predictor

Concept illustrated by this case study

■ Modeling a Branch Predictor


Besides studying microarchitecture techniques, to really understand computer architecture you must also program computers. Getting your hands dirty by directly modeling various microarchitectural ideas is better yet. Write a C or Java program to model a 2,1 branch predictor. Your program will read a series of lines from a file named `history.txt` (available on the companion CD—see Figure 2.43).


Each line of that file has three data items, separated by tabs. The first datum on each line is the address of the branch instruction in hex. The second datum is the branch target address in hex. The third datum is a 1 or a 0; 1 indicates a taken branch, and 0 indicates not taken. The total number of branches your model will consider is, of course, equal to the number of lines in the file. Assume a direct-mapped BTB, and don't worry about instruction lengths or alignment (i.e., if your BTB has four entries, then branch instructions at 0x0, 0x1, 0x2, and 0x3 will reside in those four entries, but a branch instruction at 0x4 will overwrite BTB[0]). For each line in the input file, your model will read the pair of data values, adjust the various tables per the branch predictor being modeled, and collect key performance statistics. The final output of your program will look like that shown in Figure 2.44.

Make the number of BTB entries in your model a command-line option.

- 2.13 [20/10/10/10/10/10] <2.3> Write a model of a simple four-state branch target buffer with 64 entries.
- a. [20] <2.3> What is the overall hit rate in the BTB (the fraction of times a branch was looked up in the BTB and found present)?

0x40074cdb	0x40074cdf	1
0x40074ce2	0x40078d12	0
0x4009a247	0x4009a2bb	0
0x4009a259	0x4009a2c8	0
0x4009a267	0x4009a2ac	1
0x4009a2b4	0x4009a2ac	1
...		


 Address of branch
instruction


 Branch target
address



 1: taken
0: not taken

Figure 2.43 Sample `history.txt` input file format.

- b. [10] <2.3> What is the overall branch misprediction rate on a cold start (the fraction of times a branch was correctly predicted taken or not taken, regardless of whether that prediction “belonged to” the branch being predicted)?
- c. [10] <2.3> Find the most common branch. What was its contribution to the overall number of correct predictions? (*Hint:* Count the number of times that branch occurs in the history.txt file, then track how each instance of that branch fares within the BTB model.)
- d. [10] <2.3> How many capacity misses did your branch predictor suffer?
- e. [10] <2.3> What is the effect of a cold start versus a warm start? To find out, run the same input data set once to initialize the history table, and then again to collect the new set of statistics.
- f. [10] <2.3> Cold-start the BTB 4 more times, with BTB sizes 16, 32, and 64. Graph the resulting five misprediction rates. Also graph the five hit rates.
- g. [10] Submit the well-written, commented source code for your branch target buffer model.

Exercise 2.13 (a)

Number of hits BTB: 54390. Total brs: 55493. Hit rate: 99.8%

Exercise 2.13 (b)

Incorrect predictions: 1562 of 55493, or 2.8%

Exercise 2.13 (c)

<a simple unix command line shell script will give you the most common branch...show how you got it here.>

Most signif. branch seen 15418 times, out of 55493 tot brs ;
27.8%

MS branch = 0x80484ef, correct predictions = 19151 (of 36342
total correct preds) or 52.7%

Exercise 2.13 (d)

Total unique branches (1 miss per br compulsory): 121

Total misses seen: 104.

So total capacity misses = total misses - compulsory misses = 17

Exercise 2.13 (e)

Number of hits in BTB: 54390. Total brs: 55493. Hit rate: 99.8%

Incorrect predictions: 1103 of 54493, or 2.0%

Exercise 2.13 (f)

BTB Length	mispredict rate
1	32.91%
2	6.42%
4	0.28%
8	0.23%
16	0.21%
32	0.20%
64	0.20%

Figure 2.44 Sample program output format.