
3.1	Instruction-Level Parallelism: Concepts and Challenges	148
3.2	Basic Compiler Techniques for Exposing ILP	156
3.3	Reducing Branch Costs with Advanced Branch Prediction	162
3.4	Overcoming Data Hazards with Dynamic Scheduling	167
3.5	Dynamic Scheduling: Examples and the Algorithm	176
3.6	Hardware-Based Speculation	183
3.7	Exploiting ILP Using Multiple Issue and Static Scheduling	192
3.8	Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation	197
3.9	Advanced Techniques for Instruction Delivery and Speculation	202
3.10	Studies of the Limitations of ILP	213
3.11	Cross-Cutting Issues: ILP Approaches and the Memory System	221
3.12	Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput	223
3.13	Putting It All Together: The Intel Core i7 and ARM Cortex-A8	233
3.14	Fallacies and Pitfalls	241
3.15	Concluding Remarks: What's Ahead?	245
3.16	Historical Perspective and References	247
	Case Studies and Exercises by Jason D. Bakos and Robert P. Colwell	247

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.*

3.1

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 H, 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 C](#). If you are not thoroughly familiar with the ideas in [Appendix C](#), 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.

There are two largely separable approaches to exploiting ILP: (1) an approach that relies on hardware to help discover and exploit the parallelism dynamically, and (2) an approach that relies on software technology to find parallelism statically at compile time. Processors using the dynamic, hardware-based approach, including the Intel Core series, dominate in the desktop and server markets. In the personal mobile device market, where energy efficiency is often the key objective, designers exploit lower levels of instruction-level parallelism. Thus, in 2011, most processors for the PMD market use static approaches, as we will see in the ARM Cortex-A8; however, future processors (e.g., the new ARM Cortex-A9) are using dynamic approaches. Aggressive compiler-based approaches have been attempted numerous times beginning in the 1980s and most recently in the Intel Itanium series. Despite enormous efforts, such approaches have not been successful outside of the narrow range of scientific applications.

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. A discussion of the limitations on ILP approaches is included in this chapter, and it was such limitations that directly led to the movement to multicore. Understanding the limitations remains important in balancing the use of ILP and thread-level parallelism.

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:

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

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Delayed branches and simple branch scheduling	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Figure 3.1 The major techniques examined in [Appendix C](#), [Chapter 3](#), and [Appendix H](#) are shown together with the component of the CPI equation that the technique affects.

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 decrease 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 3.1](#) shows the techniques we examine in this chapter and in [Appendix H](#), as well as the topics covered in the introductory material in [Appendix C](#). 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 that adds two 1000-element arrays and is completely parallel:

```

for (i=0; i<=999; 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.

We will examine a number of techniques 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 3.5](#) and [3.6](#)).

An important alternative method for exploiting loop-level parallelism is the use of SIMD in both vector processors and Graphics Processing Units (GPUs), both of which are covered in [Chapter 4](#). A SIMD instruction exploits data-level parallelism by operating on a small to moderate number of data items in parallel (typically two to eight). A vector instruction exploits data-level parallelism by operating on many data items in parallel using both parallel execution units and a deep pipeline. For example, the above code sequence, which in simple form requires seven instructions per iteration (two loads, an add, a store, two address updates, and a branch) for a total of 7000 instructions, might execute in one-quarter as many instructions in some SIMD architecture where four data items are processed per instruction. On some vector processors, this sequence might take only four instructions: 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.

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*.
- 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

```



In both of the above dependent sequences, as shown by the arrows, each instruction depends 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 must execute in order and 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 C 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.

Dependencies 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 [Section 3.10](#) and in Appendix H in more detail.

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: (1) maintaining the dependence but avoiding a hazard, and (2) 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.

Dependencies 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.

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 151, 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. Because 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 exists whenever there is a name or data 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 C, 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 (or name dependence). WAR hazards cannot occur in most static issue pipelines—even deeper pipelines or floating-point pipelines—because all reads are early

(in ID in the pipeline in Appendix C) and all writes are late (in WB in the pipeline in Appendix C). 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 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, two constraints are 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. Thus, 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 dependences—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 3.6](#), we will look at a hardware technique, *speculation*, which allows us to overcome this exception problem. Appendix H 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 3.6](#).

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 H. 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 3.3](#).

3.2

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 issuing. Appendix H 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, the execution of 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 3.2 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 one clock cycle. We assume that the functional units are fully pipelined or replicated (as many times as the pipeline depth), so that an operation of any type can be issued on every clock cycle and there are no structural hazards.

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 H. We will rely on the following code segment, which adds a scalar to a vector:

```
for (i=999; 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 formalize this notion in Appendix H and describe how we can test whether loop iterations are independent at compile time. First, let's look at the performance of this loop, showing how we can use the parallelism to improve its performance for a MIPS pipeline with the latencies shown above.

The first step is to translate the above segment to MIPS assembly language. In the following code segment, R1 is initially the address of the element in the array with the highest address, and F2 contains the scalar value s . Register R2 is precomputed, so that $8(R2)$ is the address of the last element to operate on.

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 3.2 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 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 3.2](#).

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 nine 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 seven 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 seven clock cycles, but the actual work of operating on the array element takes just three (the load, add, and store) of those seven clock

cycles. The remaining four clock cycles consist of loop overhead—the DADDUI and BNE—and two stalls. To eliminate these four 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 H.

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. (As we shall see in [Chapter 4](#), this technique is similar to a technique called *strip mining*, used in compilers for vector processors.) 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 from [Figure 3.2](#).

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 H, 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 (e.g., name dependences).
- 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.

Three different effects limit the gains from loop unrolling: (1) a decrease in the amount of overhead amortized with each unroll, (2) code size limitations, and (3) 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 difficult 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.

3.3

Reducing Branch Costs with Advanced Branch 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. In Appendix C, we examine simple branch predictors that rely either on compile-time information or on the observed dynamic behavior of a branch in isolation. As the number of instructions in flight has increased, the importance of more accurate branch prediction has grown. In this section, we examine techniques for improving dynamic prediction accuracy.

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 displays particularly bad branch prediction behavior:

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

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.

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 3.3 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, but it also 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 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. In addition to the Alpha processors that pioneered tournament predictors, recent AMD processors, including both the Opteron and Phenom, have used tournament-style predictors.

Figure 3.4 looks at the performance of three different predictors (a local 2-bit predictor, a correlating predictor, and a tournament predictor) for different

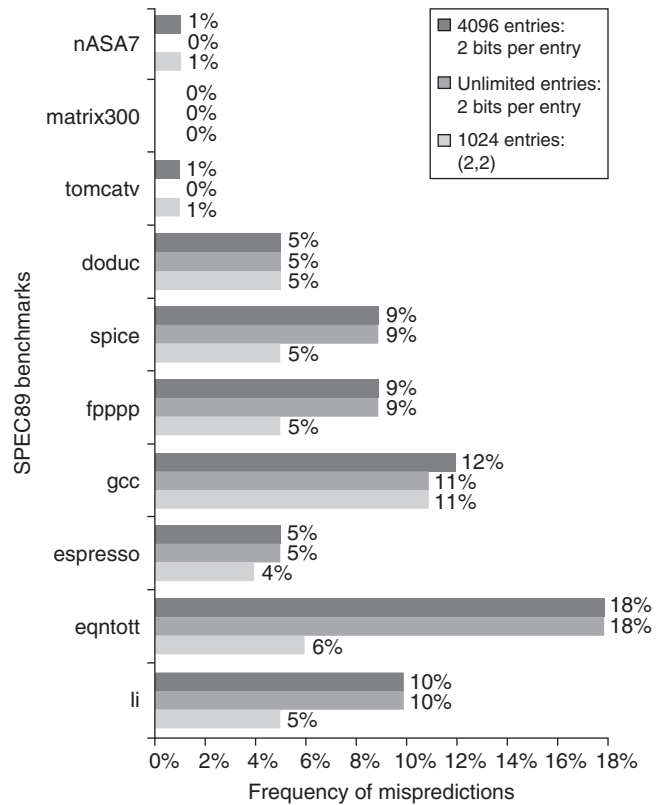


Figure 3.3 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

numbers of bits using SPEC89 as the benchmark. As we saw earlier, the prediction 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 predictor sizes.

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

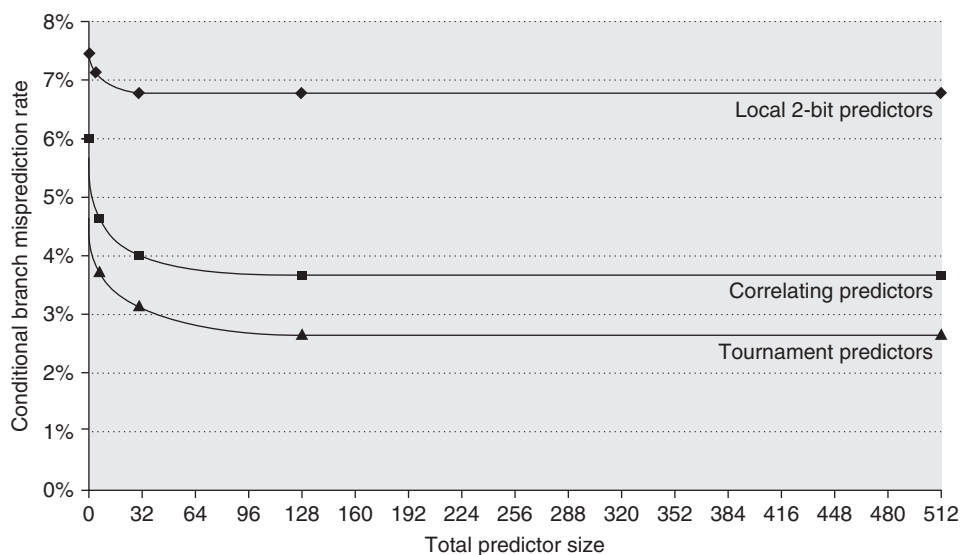


Figure 3.4 The misprediction rate for three different predictors on SPEC89 as the total number of bits is increased. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are 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.

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.

The Intel Core i7 Branch Predictor

Intel has released only limited amounts of information about the Core i7's branch predictor, which is based on earlier predictors used in the Core Duo chip. The i7 uses a two-level predictor that has a smaller first-level predictor, designed to meet the cycle constraints of predicting a branch every clock cycle, and a larger second-level predictor as a backup. Each predictor combines three different predictors: (1) the simple two-bit predictor, which was introduced in Appendix C (and used in the tournament predictor discussed above); (2) a global history predictor, like those we just saw; and (3) a loop exit predictor. The loop exit predictor uses a counter to predict the exact number of taken branches (which is the number of loop iterations) for a branch that is detected as a loop branch. For each branch, the best prediction is chosen from among the three predictors by tracking the accuracy of each prediction, like a tournament predictor. In addition to this multilevel main predictor, a separate unit predicts target addresses for indirect branches, and a stack to predict return addresses is also used.

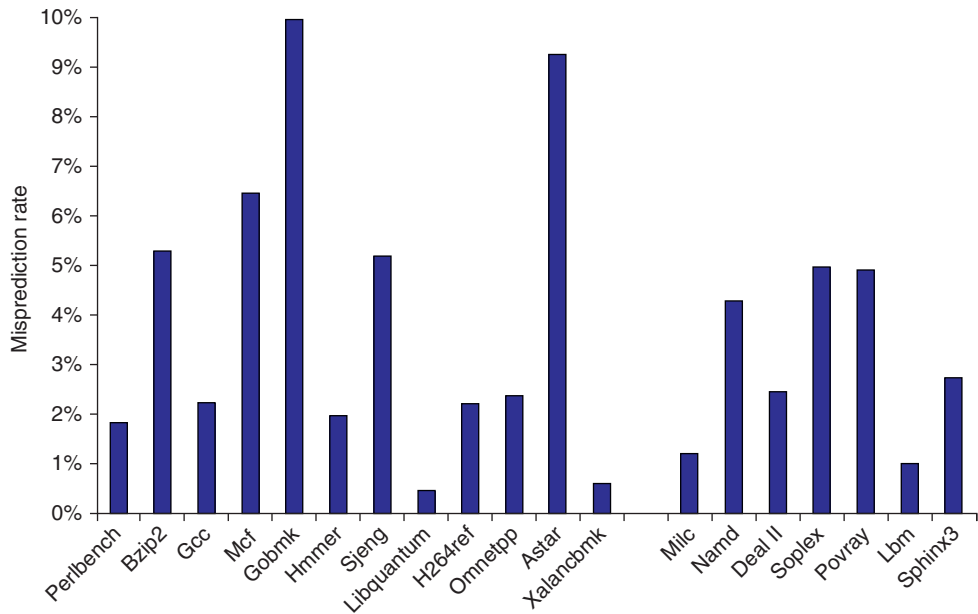


Figure 3.5 The misprediction rate for 19 of the SPEC CPU2006 benchmarks versus the number of successfully retired branches is slightly higher on average for the integer benchmarks than for the FP (4% versus 3%). More importantly, it is much higher for a few benchmarks.

As in other cases, speculation causes some challenges in evaluating the predictor, since a mispredicted branch may easily lead to another branch being fetched and mispredicted. To keep things simple, we look at the number of mispredictions as a percentage of the number of successfully completed branches (those that were not the result of misspeculation). [Figure 3.5](#) shows these data for 19 of the SPEC CPU 2006 benchmarks. These benchmarks are considerably larger than SPEC89 or SPEC2000, with the result being that the misprediction rates are slightly higher than those in [Figure 3.4](#) even with a more elaborate combination of predictors. Because branch misprediction leads to ineffective speculation, it contributes to the wasted work, as we will see later in this chapter.

3.4

Overcoming Data Hazards with Dynamic Scheduling

A simple statically scheduled pipeline fetches an instruction and issues it, unless there is 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. First, it allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline, eliminating the need to have multiple binaries and recompile for a different microarchitecture. In today's computing environment, where much of the software is from third parties and distributed in binary form, this advantage is significant. Second, it enables handling some cases when dependences are unknown at compile time; for example, they may involve a memory reference or a data-dependent branch, or they may result from a modern programming environment that uses dynamic linking or dispatching. Third, and 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. In [Section 3.6](#), we explore hardware speculation, a technique with additional performance advantages, which builds on dynamic scheduling. As we will see, the advantages of dynamic scheduling are gained at a cost of 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 3.2](#)) 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 delaying the notification of an associated exception until the processor knows that the instruction should be the next one completed.

Although exception behavior must be preserved, dynamically scheduled processors could 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 3.6](#). 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 execution 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; 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. Here, we focus on a more sophisticated technique, called *Tomasulo's algorithm*. The primary difference is that Tomasulo's algorithm handles antidependences and output dependences by effectively renaming the registers dynamically. Additionally, Tomasulo's algorithm can be extended to handle *speculation*, a technique to reduce the effect of control dependences by predicting the outcome of a branch, executing instructions at the predicted destination address, and taking corrective actions when the prediction was wrong. While the use of scoreboarding is probably sufficient to support a simple two-issue superscalar like the ARM A8, a more aggressive processor, like the four-issue Intel i7, benefits from the use of out-of-order execution.

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 in hardware 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, which is exactly what the simpler scoreboarding approach provides. 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 potential WAR and WAW hazards:

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 are two antidependences: between the ADD.D and the SUB.D and between the S.D and the MUL.D. There is also an output dependence between the ADD.D and the MUL.D, leading to three possible hazards: WAR hazards on the use of F8 by ADD.D and the use of F6 by the SUB.D, as well as 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 three name dependences can all 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 determines 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 3.6 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 is 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.

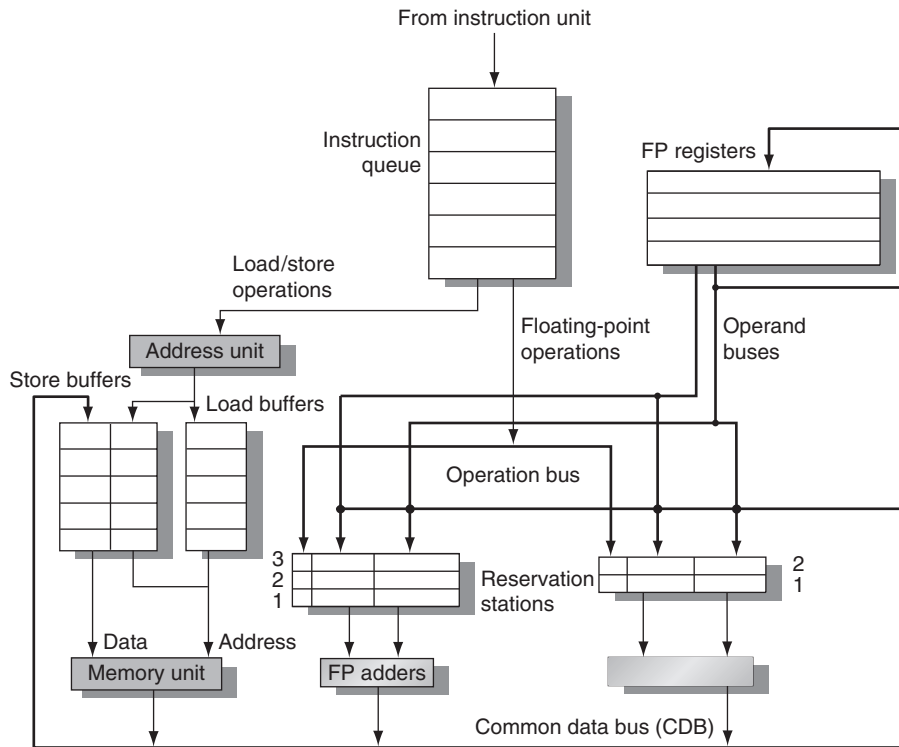


Figure 3.6 The basic structure of a MIPS floating-point unit using Tomasulo's algorithm. Instructions are sent from the instruction unit into the instruction queue from which they are issued in first-in, first-out (FIFO) order. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: (1) hold the components of the effective address until it is computed, (2) track outstanding loads that are waiting on the memory, and (3) hold the results of completed loads that are waiting for the CDB. Similarly, store buffers have three functions: (1) hold the components of the effective address until it is computed, (2) hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and (3) 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.

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 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 four 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 3.6](#).

In Tomasulo's scheme, as well as the subsequent methods we look at for supporting speculation, results are broadcast 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.

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. (This is a key difference between Tomasulo's scheme and scoreboarding: In scoreboarding, operands stay in the registers and are only read after the producing instruction completes and the consuming instruction is ready to execute.)

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.
- Vj, Vk—The value of the source operands. Note that only one of the V fields or the Q field is valid for each operand. For loads, the Vk field is used to hold the offset field.
- 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.

3.5

Dynamic Scheduling: Examples and the Algorithm

Before we examine Tomasulo's algorithm in detail, let's consider a few examples that 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 3.7](#) 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		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					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

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

Figure 3.7 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.

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 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 3.7](#) 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.

Example Using the same code segment as in the previous example (page 176), 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 3.8](#). 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.

Tomasulo's Algorithm: The Details

[Figure 3.9](#) 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 3.6](#).

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[44 + Regs[R3]]	Regs[F4]			
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

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

Figure 3.8 Multiply and divide are the only instructions not finished.

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.

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[rs].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 3.9 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 values of Qj or Qk are 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 3.6](#), we will see how speculation support removes this restriction.

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 3.10 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 four clock cycles. With a latency of six 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.

Instruction status				
Instruction	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 3.10 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.

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 3.9](#).

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 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 have 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. 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.
2. 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, dynamic scheduling, and speculation became more important.
3. 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.

3.6

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 H 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: (1) dynamic branch prediction to choose which instructions to execute, (2) 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 (3) 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 3.11 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

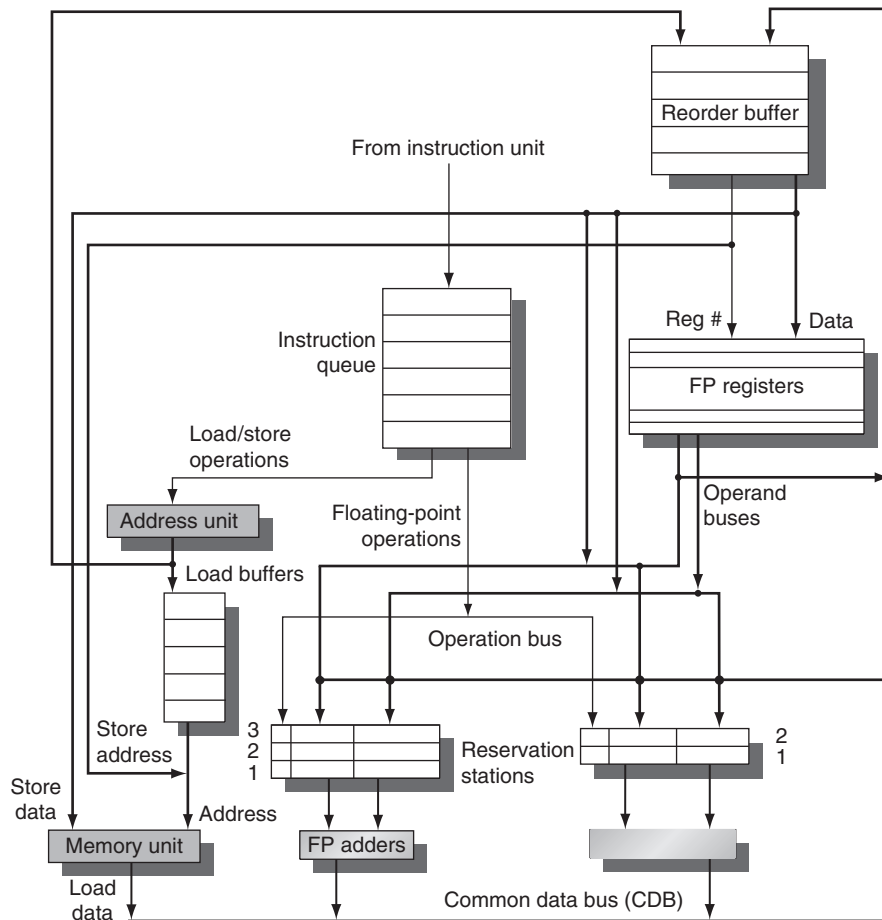


Figure 3.11 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.6 on page 173, which implemented Tomasulo's algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to multiple issue by making the CDB wider to allow for multiple completions per clock.

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 a queue that replaces the ROB to decide 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 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 3.8](#), 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,F2,F6
DIV.D	F10,F0,F6
ADD.D	F6,F8,F2

Answer [Figure 3.12](#) 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 information that they did for Tomasulo's algorithm (see page 176 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 3.12](#) with that of [Figure 3.8](#) on page 179, 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 3.8](#) 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.

Reorder buffer						
Entry	Busy	Instruction		State	Destination	Value
1	No	L.D	F6, 32(R2)	Commit	F6	Mem[32 + Regs[R2]]
2	No	L.D	F2, 44(R3)	Commit	F2	Mem[44 + 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[44 + Regs[R3]]	Regs[F4]			#3	
Mult2	Yes	DIV.D		Mem[32 + 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 3.12 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 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 J 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 3.10 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 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 3.13 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 3.13 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 is shown. The remaining instructions will be committed as quickly 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 3.13](#). 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 3.14](#) 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 3.14](#) with the comparable figure for Tomasulo's algorithm in [Figure 3.9](#) 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 3.14](#) has one significant simplification for stores, which is unneeded in practice. [Figure 3.14](#) 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

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 3.14 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.

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 3.14](#), which is already in a small font, would be even longer! Although [Figure 3.14](#) 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.
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 3.9](#).

Although this explanation of speculative execution has focused on floating point, the techniques easily extend to the integer registers and functional units. 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.

3.7

Exploiting ILP Using Multiple Issue and Static Scheduling

The techniques of the preceding sections can be used to eliminate data, 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
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 H, they also introduced the name EPIC—explicitly parallel instruction computer—for this architectural style.

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 3.15 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 the 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 later in this chapter, the growth in overhead is a major factor limiting wider issue processors.

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, including the ARM Cortex-A8
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	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Figure 3.15 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 H 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.

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 to 24 bits per unit, yielding an instruction length of between 80 and 120 bits. By comparison, the Intel Itanium 1 and 2 contain six operations per instruction packet (i.e., they allow concurrent issue of two three-instruction bundles, as Appendix H describes).

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 require 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 H, 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 158 for the MIPS code) for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore delayed branches.

Answer Figure 3.16 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 3.2 that used unrolled and scheduled code.

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 H, we examine software scheduling

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 3.16 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.

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 H, 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 are 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 the next chapter). 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.

3.8

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 multiple-issue superscalar pipeline with separate integer, load/store, and floating-point units (both FP multiply and FP add), 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. As [Figure 3.17](#) shows, the basic organization is similar to that of a processor with speculation with one issue per clock, except that the issue and completion logic must be enhanced to allow multiple instructions to be processed per clock.

Issuing multiple instructions per clock in a dynamically scheduled processor (with or without speculation) is very complex for the simple reason that the multiple instructions may depend on one another. Because of this the tables must be updated for the instructions in parallel; otherwise, the tables will be incorrect or the dependence may be lost.

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; this approach cannot be easily extended to handle four instructions per clock, unfortunately.

A second alternative is to build the logic necessary to handle two or more instructions at once, including any possible dependences between the instructions. Modern superscalar processors that issue four or more instructions per clock may include both approaches: They both pipeline and widen the issue logic. A key observation is that we cannot simply pipeline away the problem. By making instruction issues take multiple clocks because new instructions are issuing every clock cycle, we must be able to assign the reservation station and to update the pipeline tables, so that a dependent instruction issuing on the next clock can use the updated information.

This issue step is one of the most fundamental bottlenecks in dynamically scheduled superscalars. To illustrate the complexity of this process, [Figure 3.18](#)

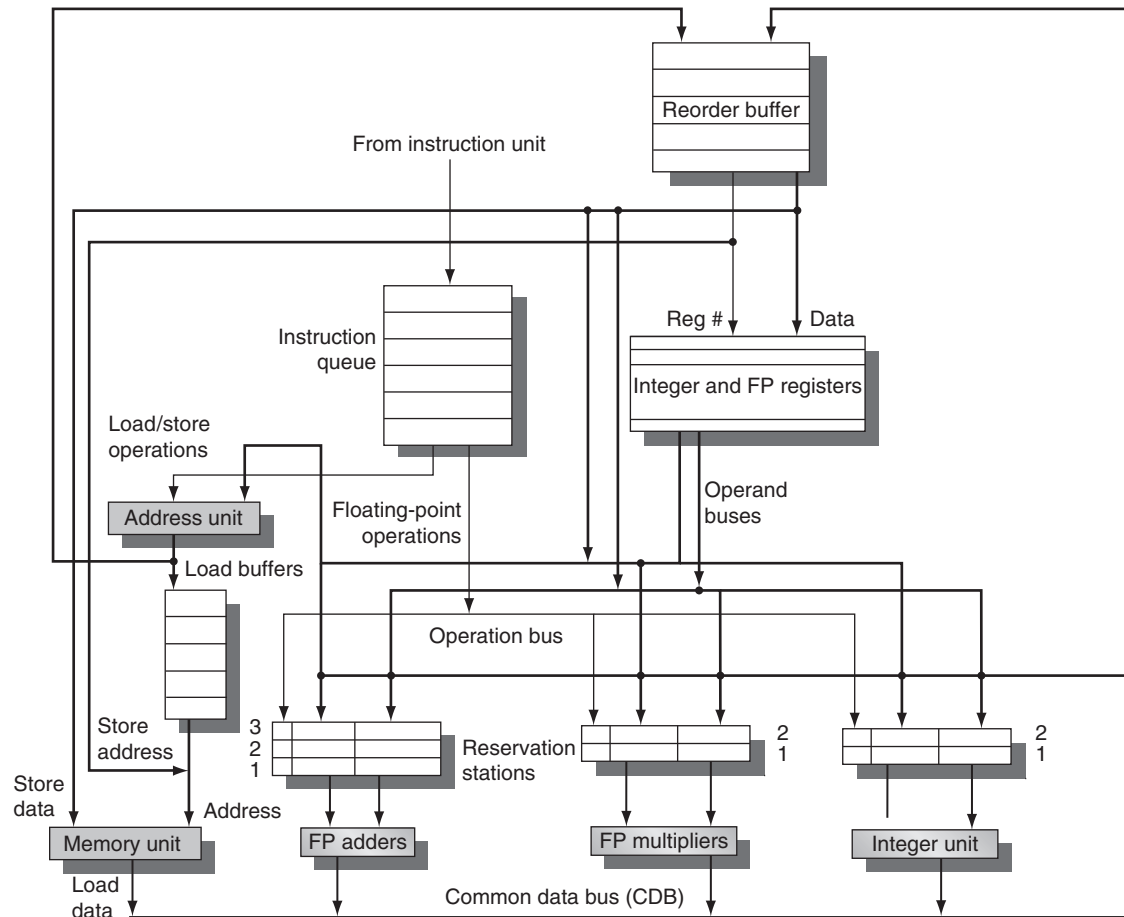


Figure 3.17 The basic organization of a multiple issue processor with speculation. In this case, the organization could allow a FP multiply, FP add, integer, and load/store to all issues simultaneously (assuming one issue per clock per functional unit). Note that several datapaths must be widened to support multiple issues: the CDB, the operand buses, and, critically, the instruction issue logic, which is not shown in this figure. The last is a difficult problem, as we discuss in the text.

shows the issue logic for one case: issuing a load followed by a dependent FP operation. The logic is based on that in Figure 3.14 on page 191, but represents only one case. In a modern superscalar, every possible combination of dependent instructions that is allowed to issue in the same clock cycle must be considered. Since the number of possibilities climbs as the square of the number of instructions that can be issued in a clock, the issue step is a likely bottleneck for attempts to go beyond four instructions per clock.

We can generalize the detail of Figure 3.18 to describe the basic strategy for updating the issue logic and the reservation tables in a dynamically scheduled superscalar with up to n issues per clock as follows:

Action or bookkeeping	Comments
<pre> if (RegisterStat[rs1].Busy)/*in-flight instr. writes rs*/ {h ← RegisterStat[rs1].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r1].Vj ← ROB[h].Value; RS[r1].Qj ← 0;} else {RS[r1].Qj ← h;} /* wait for instruction */ } else {RS[r1].Vj ← Regs[rs]; RS[r1].Qj ← 0;;} RS[r1].Busy ← yes; RS[r1].Dest ← b1; ROB[b1].Instruction ← Load; ROB[b1].Dest ← rd1; ROB[b1].Ready ← no; RS[r].A ← imm1; RegisterStat[rt1].Reorder ← b1; RegisterStat[rt1].Busy ← yes; ROB[b1].Dest ← rt1; RS[r2].Qj ← b1;} /* wait for load instruction */ </pre>	<p>Updating the reservation tables for the load instruction, which has a single source operand. Because this is the first instruction in this issue bundle, it looks no different than what would normally happen for a load.</p>
<pre> if (RegisterStat[rt2].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt2].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r2].Vk ← ROB[h].Value; RS[r2].Qk ← 0;} else {RS[r2].Qk ← h;} /* wait for instruction */ } else {RS[r2].Vk ← Regs[rt2]; RS[r2].Qk ← 0;;} RegisterStat[rd2].Reorder ← b2; RegisterStat[rd2].Busy ← yes; ROB[b2].Dest ← rd2; </pre>	<p>Since we assumed that the second operand of the FP instruction was from a prior issue bundle, this step looks like it would in the single-issue case. Of course, if this instruction was dependent on something in the same issue bundle the tables would need to be updated using the assigned reservation buffer.</p>
<pre> RS[r2].Busy ← yes; RS[r2].Dest ← b2; ROB[b2].Instruction ← FP operation; ROB[b2].Dest ← rd2; ROB[b2].Ready ← no; </pre>	<p>This section simply updates the tables for the FP operation, and is independent of the load. Of course, if further instructions in this issue bundle depended on the FP operation (as could happen with a four-issue superscalar), the updates to the reservation tables for those instructions would be effected by this instruction.</p>

Figure 3.18 The issue steps for a pair of dependent instructions (called 1 and 2) where instruction 1 is FP load and instruction 2 is an FP operation whose first operand is the result of the load instruction; r1 and r2 are the assigned reservation stations for the instructions; and b1 and b2 are the assigned reorder buffer entries. For the issuing instructions, rd1 and rd2 are the destinations; rs1, rs2, and rt2 are the sources (the load only has one source); r1 and r2 are the reservation stations allocated; and b1 and b2 are the assigned ROB entries. RS is the reservation station data structure. RegisterStat is the register data structure, Regs represents the actual registers, and ROB is the reorder buffer data structure. Notice that we need to have assigned reorder buffer entries for this logic to operate properly and recall that all these updates happen in a single clock cycle in parallel, not sequentially!

1. Assign a reservation station and a reorder buffer for *every* instruction that *might* be issued in the next issue bundle. This assignment can be done before the instruction types are known, by simply preallocating the reorder buffer entries sequentially to the instructions in the packet using n available reorder buffer entries and by ensuring that enough reservation stations are available to issue the whole bundle, independent of what it contains. By limiting the number of instructions of a given class (say, one FP, one integer, one load,

one store), the necessary reservation stations can be preallocated. Should sufficient reservation stations not be available (such as when the next few instructions in the program are all of one instruction type), the bundle is broken, and only a subset of the instructions, in the original program order, is issued. The remainder of the instructions in the bundle can be placed in the next bundle for potential issue.

2. Analyze all the dependences among the instructions in the issue bundle.
3. If an instruction in the bundle depends on an earlier instruction in the bundle, use the assigned reorder buffer number to update the reservation table for the dependent instruction. Otherwise, use the existing reservation table and reorder buffer information to update the reservation table entries for the issuing instruction.

Of course, what makes the above very complicated is that it is all done in parallel in a single clock cycle!

At the back-end of the pipeline, we must be able to complete and commit multiple instructions per clock. These steps are somewhat easier than the issue problems since multiple instructions that can actually commit in the same clock cycle must have already dealt with and resolved any dependences. As we will see, designers have figured out how to handle this complexity: The Intel i7, which we examine in [Section 3.13](#), uses essentially the scheme we have described for speculative multiple issue, including a large number of reservation stations, a reorder buffer, and a load and store buffer that is also used to handle nonblocking cache misses.

From a performance viewpoint, 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 3.19](#) and [3.20](#) 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 3.19 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 3.20](#) 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 nonspeculative 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 does not improve performance; in fact, it typically harms performance and, as we shall see, dramatically lowers energy efficiency.

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 3.20 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.

3.9

Advanced Techniques for Instruction Delivery and Speculation

In a high-performance pipeline, especially one with multiple issues, 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 to 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 attempts to predict the result of a computation and 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 program counter (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 3.21 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 an address 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.

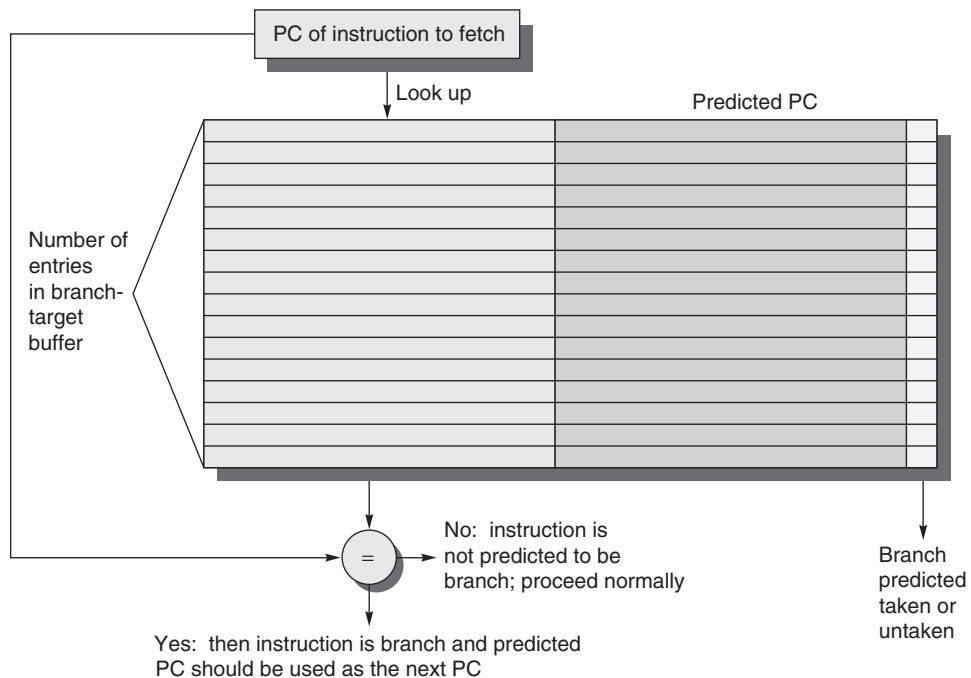


Figure 3.21 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and 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 worse performance. 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 3.22 shows the steps when using a branch-target buffer for a simple five-stage pipeline. From this figure we can see that there will be no branch delay

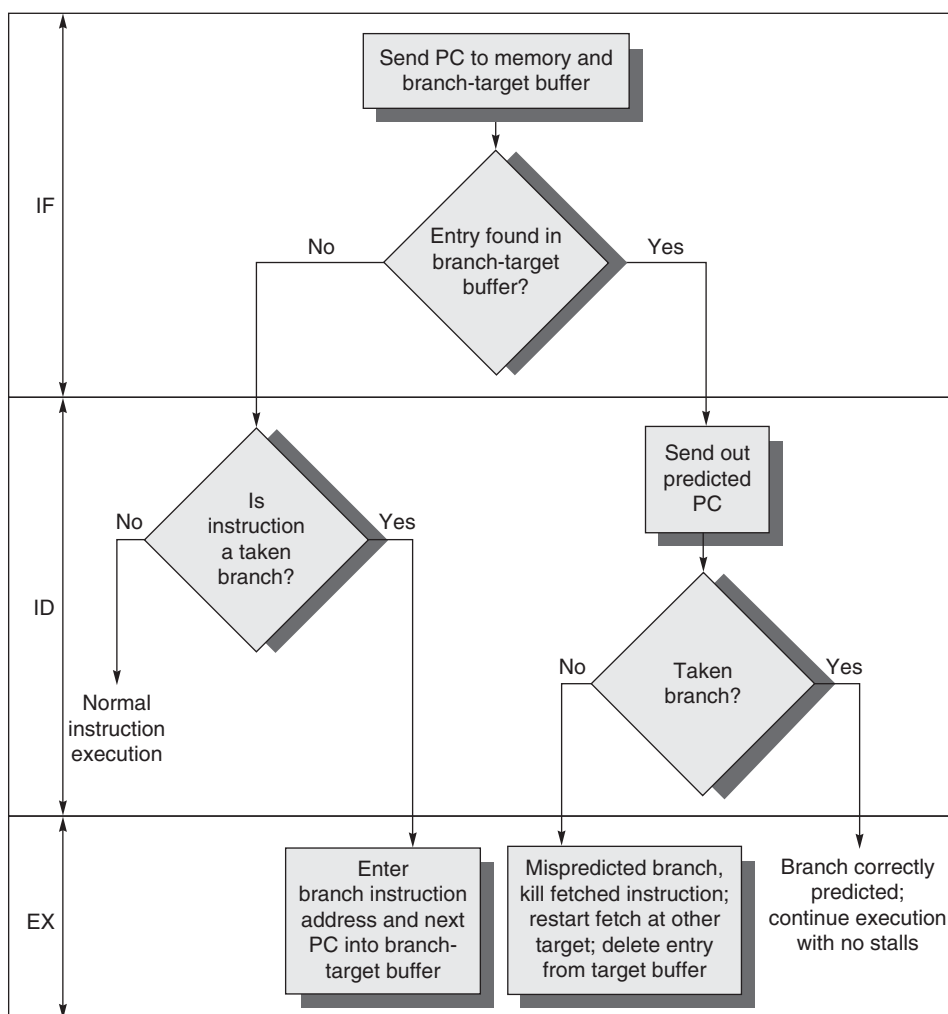


Figure 3.22 The steps involved in handling an instruction with a branch-target buffer.

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 3.23 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 one clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and one clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and taken, a two-cycle penalty is encountered, during which time the buffer is updated.

if a branch-prediction entry is found in the buffer and the prediction is correct. Otherwise, there will be a penalty of at least two 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 3.23 contains this information for a simple five-stage pipeline.

Example Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions from Figure 3.23. 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 two 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 C, 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. Modern high-performance processors have branch misprediction delays on the order of 15 clock cycles; clearly, accurate prediction is critical!

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.

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 runtime. 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 such as 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 3.24 shows the performance of such a return buffer with 0 to 16 elements for a number of the SPEC CPU95 benchmarks. We will use a similar return predictor when we examine the studies of

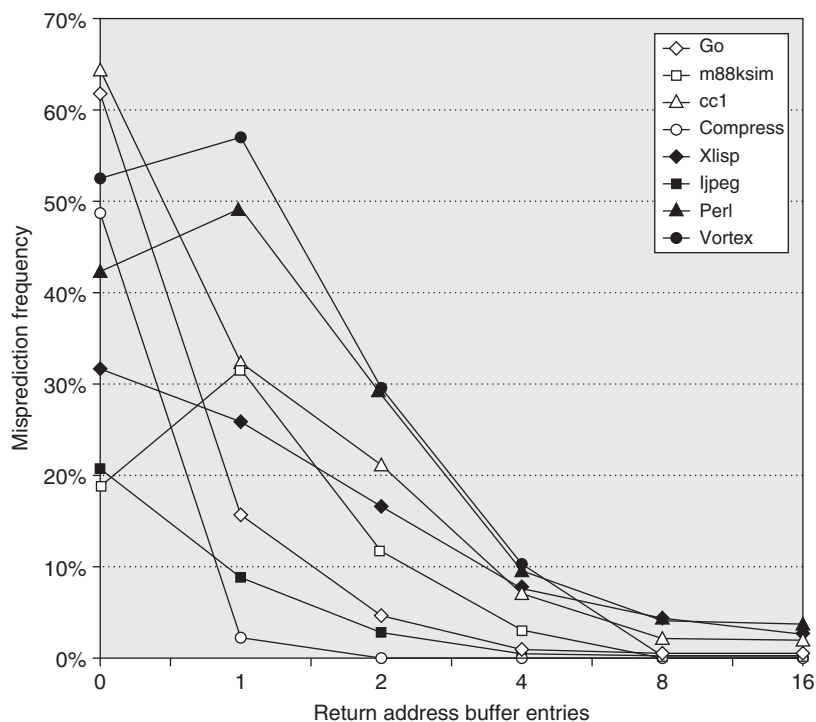


Figure 3.24 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Since call depths are typically not large, with some exceptions, a modest buffer works well. These data come from Skadron et al. [1999] and use a fix-up mechanism to prevent corruption of the cached return addresses.

ILP in [Section 3.10](#). Both the Intel Core processors and the AMD Phenom processors have return address predictors.

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 2](#) 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.

Virtually all high-end processors now use a separate instruction fetch unit connected to the rest of the pipeline by a buffer containing pending instructions.

Speculation: Implementation Issues and Extensions

In this section we explore four issues that involve the design trade-offs in speculation, starting with the use of register renaming, the approach that is often used instead 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 most of the function of the ROB and the reservation stations; only a queue to ensure that instructions complete in order is needed. 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, a function performed by 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. Although an ROB is not necessary with register renaming, the hardware must still track instructions in a queue-like structure and update the renaming table in strict order.

An advantage of the renaming approach versus the ROB approach is that instruction commit is slightly simplified, since it requires only two simple actions: (1) record that the mapping between an architectural register number and physical register number is no longer speculative, and (2) 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 is used in most 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.

Both register renaming and reorder buffers continue to be used in high-end processors, which now feature the ability to have as many as 40 or 50 instructions (including loads and stores waiting on the cache) in flight. Whether renaming or a reorder buffer is used, the key complexity bottleneck for a dynamically scheduled superscalar remains issuing bundles of instructions with dependences within the bundle. In particular, dependent instructions in an issue bundle must be issued with the assigned virtual registers of the instructions on which they depend. A strategy for instruction issue with register renaming similar to that used for multiple issue with reorder buffers (see page 198) can be deployed, as follows:

1. The issue logic pre-reserves enough physical registers for the entire issue bundle (say, four registers for a four-instruction bundle with at most one register result per instruction).
2. The issue logic determines what dependences exist within the bundle. If a dependence does not exist within the bundle, the register renaming structure is used to determine the physical register that holds, or will hold, the result on which instruction depends. When no dependence exists within the bundle the result is from an earlier issue bundle, and the register renaming table will have the correct register number.
3. If an instruction depends on an instruction that is earlier in the bundle, then the pre-reserved physical register in which the result will be placed is used to update the information for the issuing instruction.

Note that just as in the reorder buffer case, the issue logic must both determine dependences within the bundle and update the renaming tables in a single clock, and, as before, the complexity of doing this for a larger number of instructions per clock becomes a chief limitation in the issue width.

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 translation lookaside buffer (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 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 shortly.

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: (1) a very high branch frequency, (2) significant clustering of branches, and (3) 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. As of 2011, no processor has yet combined full speculation with resolving multiple branches per cycle, and it is unlikely that the costs of doing so would be justified in terms of performance versus complexity and power.

Speculation and the Challenge of Energy Efficiency

What is the impact of speculation on energy efficiency? At first glance, one might argue that using speculation always decreases energy efficiency, since whenever speculation is wrong it consumes excess energy in two ways:

1. The instructions that were speculated and whose results were not needed generated excess work for the processor, wasting energy.
2. Undoing the speculation and restoring the state of the processor to continue execution at the appropriate address consumes additional energy that would not be needed without speculation.

Certainly, speculation will raise the power consumption and, if we could control speculation, it would be possible to measure the cost (or at least the dynamic power cost). But, if speculation lowers the execution time by more than it increases the average power consumption, then the total energy consumed may be less.

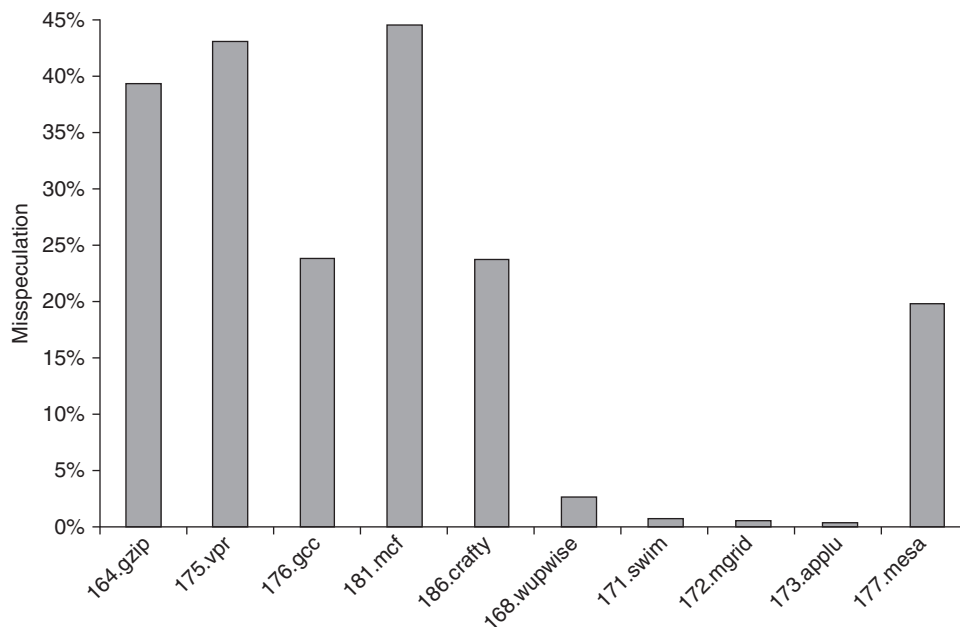


Figure 3.25 The fraction of instructions that are executed as a result of misspeculation is typically much higher for integer programs (the first five) versus FP programs (the last five).

Thus, to understand the impact of speculation on energy efficiency, we need to look at how often speculation is leading to unnecessary work. If a significant number of unneeded instructions is executed, it is unlikely that speculation will improve running time by a comparable amount! [Figure 3.25](#) shows the fraction of instructions that are executed from misspeculation. As we can see, this fraction is small in scientific code and significant (about 30% on average) in integer code. Thus, it is unlikely that speculation is energy efficient for integer applications. Designers could avoid speculation, try to reduce the misspeculation, or think about new approaches, such as only speculating on branches that are known to be highly predictable.

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 with other program behavior.

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.

Although many researchers have focused on value prediction in the past ten years, the results have never been sufficiently attractive to justify their incorporation in real processors. Instead, a simpler and older idea, related to value prediction, has been used: 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. This limited form of address value speculation has been used in several processors already and may become universal in the future.

3.10 Studies of the Limitations of ILP

Exploiting ILP to increase performance began with the first pipelined processors in the 1960s. In the 1980s and 1990s, these techniques were key to achieving rapid performance improvements. The question of how much ILP exists was critical to our long-term ability to enhance performance at a rate that exceeds the increase in speed of the base integrated circuit technology. On a shorter scale, the critical question of what is needed to exploit more ILP is crucial to both computer designers and compiler writers. The data in this section also provide us with a way to examine the value of ideas that we have introduced in this chapter, including memory disambiguation, register renaming, and speculation.

In this section we review a portion of one of the studies done of these questions (based on Wall's 1993 study). All of these studies of available parallelism operate by making a set of assumptions and seeing how much parallelism is available under those assumptions. The data we examine here are from a study that makes the fewest assumptions; in fact, the ultimate hardware model is probably unrealizable. Nonetheless, all such studies assume a certain level of compiler technology, and some of these assumptions could affect the results, despite the use of incredibly ambitious hardware.

As we will see, for hardware models that have reasonable cost, it is unlikely that the costs of very aggressive speculation can be justified: the inefficiencies in power and use of silicon are simply too high. While many in the research community and the major processor manufacturers were betting in favor of much greater exploitable ILP and were initially reluctant to accept this possibility, by 2005 they were forced to change their minds.

The Hardware Model

To see what the limits of ILP might be, we first need to define an ideal processor. An ideal processor is one where all constraints on ILP are removed. The only limits on ILP in such a processor are those imposed by the actual data flows through either registers or memory.

The assumptions made for an ideal or perfect processor are as follows:

1. *Infinite register renaming*—There are an infinite number of virtual registers available, and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.
2. *Perfect branch prediction*—Branch prediction is perfect. All conditional branches are predicted exactly.
3. *Perfect jump prediction*—All jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.
4. *Perfect memory address alias analysis*—All memory addresses are known exactly, and a load can be moved before a store provided that the addresses are not identical. Note that this implements perfect address alias analysis.
5. *Perfect caches*—All memory accesses take one clock cycle. In practice, superscalar processors will typically consume large amounts of ILP hiding cache misses, making these results highly optimistic.

Assumptions 2 and 3 eliminate *all* control dependences. Likewise, assumptions 1 and 4 eliminate *all but the true* data dependences. Together, these four assumptions mean that *any* instruction in the program's execution can be scheduled on the cycle immediately following the execution of the predecessor on which it depends. It is even possible, under these assumptions, for the *last* dynamically executed instruction in the program to be scheduled on the very first cycle! Thus, this set of assumptions subsumes both control and address speculation and implements them as if they were perfect.

Initially, we examine a processor that can issue an unlimited number of instructions at once, looking arbitrarily far ahead in the computation. For all the processor models we examine, there are no restrictions on what types of instructions can execute in a cycle. For the unlimited-issue case, this means there may be an unlimited number of loads or stores issuing in one clock cycle. In addition, all functional unit latencies are assumed to be one cycle, so that any sequence of dependent instructions can issue on successive cycles. Latencies longer than one cycle would decrease the number of issues per cycle, although not the number of instructions under execution at any point. (The instructions in execution at any point are often referred to as *in flight*.)

Of course, this ideal processor is probably unrealizable. For example, the IBM Power7 (see Wendell et. al. [2010]) is the most advanced superscalar processor

announced to date. The Power7 issues up to six instructions per clock and initiates execution on up to 8 of 12 execution units (only two of which are load/store units), supports a large set of renaming registers (allowing hundreds of instructions to be in flight), uses a large aggressive branch predictor, and employs dynamic memory disambiguation. The Power7 continued the move toward using more thread-level parallelism by increasing the width of simultaneous multithreading (SMT) support (to four threads per core) and the number of cores per chip to eight. After looking at the parallelism available for the perfect processor, we will examine what might be achievable in any processor likely to be designed in the near future.

To measure the available parallelism, a set of programs was compiled and optimized with the standard MIPS optimizing compilers. The programs were instrumented and executed to produce a trace of the instruction and data references. Every instruction in the trace is then scheduled as early as possible, limited only by the data dependences. Since a trace is used, perfect branch prediction and perfect alias analysis are easy to do. With these mechanisms, instructions may be scheduled much earlier than they would otherwise, moving across large numbers of instructions on which they are not data dependent, including branches, since branches are perfectly predicted.

Figure 3.26 shows the average amount of parallelism available for six of the SPEC92 benchmarks. Throughout this section the parallelism is measured by the average instruction issue rate. Remember that all instructions have a one-cycle latency; a longer latency would reduce the average number of instructions per clock. Three of these benchmarks (fpppp, doduc, and tomcatv) are floating-point intensive, and the other three are integer programs. Two of the floating-point benchmarks (fpppp and tomcatv) have extensive parallelism, which could be exploited by a vector computer or by a multiprocessor (the structure in fpppp is quite messy, however, since some hand transformations have been done on the code). The doduc program has extensive parallelism, but the parallelism does not occur in simple parallel loops as it does in fpppp and tomcatv. The program li is a LISP interpreter that has many short dependences.

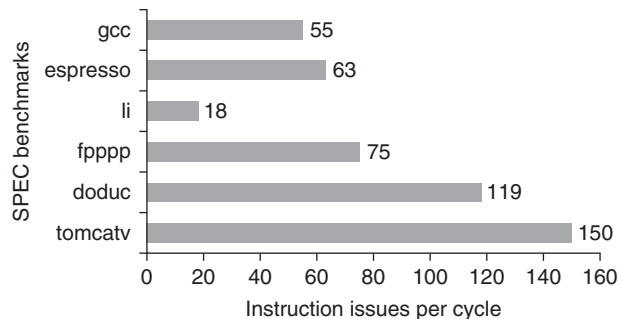


Figure 3.26 ILP available in a perfect processor for six of the SPEC92 benchmarks. The first three programs are integer programs, and the last three are floating-point programs. The floating-point programs are loop intensive and have large amounts of loop-level parallelism.

Limitations on ILP for Realizable Processors

In this section we look at the performance of processors with ambitious levels of hardware support equal to or better than what is available in 2011 or, given the events and lessons of the last decade, likely to be available in the near future. In particular, we assume the following fixed attributes:

1. Up to 64 instruction issues per clock with *no* issue restrictions, or more than 10 times the total issue width of the widest processor in 2011. As we discuss later, the practical implications of very wide issue widths on clock rate, logic complexity, and power may be the most important limitations on exploiting ILP.
2. A tournament predictor with 1K entries and a 16-entry return predictor. This predictor is comparable to the best predictors in 2011; the predictor is not a primary bottleneck.
3. Perfect disambiguation of memory references done dynamically—this is ambitious but perhaps attainable for small window sizes (and hence small issue rates and load/store buffers) or through address aliasing prediction.
4. Register renaming with 64 additional integer and 64 additional FP registers, which is slightly less than the most aggressive processor in 2011. The Intel Core i7 has 128 entries in its reorder buffer, although they are not split between integer and FP, while the IBM Power7 has almost 200. Note that we assume a pipeline latency of one cycle, which significantly reduces the need for reorder buffer entries. Both the Power7 and the i7 have latencies of 10 cycles or greater.

Figure 3.27 shows the result for this configuration as we vary the window size. This configuration is more complex and expensive than any existing implementations, especially in terms of the number of instruction issues, which is more than 10 times larger than the largest number of issues available on any processor in 2011. Nonetheless, it gives a useful bound on what future implementations might yield. The data in these figures are likely to be very optimistic for another reason. There are no issue restrictions among the 64 instructions: They may all be memory references. No one would even contemplate this capability in a processor in the near future. Unfortunately, it is quite difficult to bound the performance of a processor with reasonable issue restrictions; not only is the space of possibilities quite large, but the existence of issue restrictions requires that the parallelism be evaluated with an accurate instruction scheduler, making the cost of studying processors with large numbers of issues very expensive.

In addition, remember that in interpreting these results cache misses and non-unit latencies have not been taken into account, and both these effects will have significant impact!

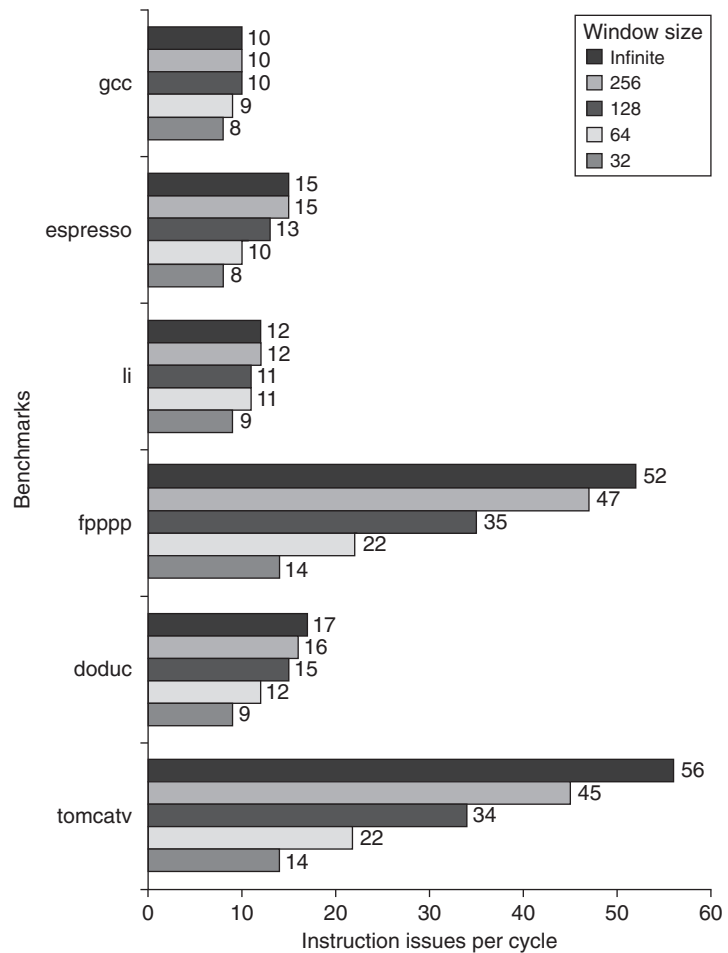


Figure 3.27 The amount of parallelism available versus the window size for a variety of integer and floating-point programs with up to 64 arbitrary instruction issues per clock. Although there are fewer renaming registers than the window size, the fact that all operations have one-cycle latency and the number of renaming registers equals the issue width allows the processor to exploit parallelism within the entire window. In a real implementation, the window size and the number of renaming registers must be balanced to prevent one of these factors from overly constraining the issue rate.

The most startling observation from Figure 3.27 is that, with the realistic processor constraints listed above, the effect of the window size for the integer programs is not as severe as for FP programs. This result points to the key difference between these two types of programs. The availability of loop-level parallelism in two of the FP programs means that the amount of ILP that can be exploited is higher, but for integer programs other factors—such as branch prediction, register renaming, and less parallelism, to start with—are all important limitations. This observation is critical because of the increased emphasis on integer

performance since the explosion of the World Wide Web and cloud computing starting in the mid-1990s. Indeed, most of the market growth in the last decade—transaction processing, Web servers, and the like—depended on integer performance, rather than floating point. As we will see in the next section, for a realistic processor in 2011, the actual performance levels are much lower than those shown in [Figure 3.27](#).

Given the difficulty of increasing the instruction rates with realistic hardware designs, designers face a challenge in deciding how best to use the limited resources available on an integrated circuit. One of the most interesting trade-offs is between simpler processors with larger caches and higher clock rates versus more emphasis on instruction-level parallelism with a slower clock and smaller caches. The following example illustrates the challenges, and in the next chapter we will see an alternative approach to exploiting fine-grained parallelism in the form of GPUs.

Example Consider the following three hypothetical, but not atypical, processors, which we run with the SPEC gcc benchmark:

1. A simple MIPS two-issue static pipe running at a clock rate of 4 GHz and achieving a pipeline CPI of 0.8. This processor has a cache system that yields 0.005 misses per instruction.
2. A deeply pipelined version of a two-issue MIPS processor with slightly smaller caches and a 5 GHz clock rate. The pipeline CPI of the processor is 1.0, and the smaller caches yield 0.0055 misses per instruction on average.
3. A speculative superscalar with a 64-entry window. It achieves one-half of the ideal issue rate measured for this window size. (Use the data in [Figure 3.27](#).) This processor has the smallest caches, which lead to 0.01 misses per instruction, but it hides 25% of the miss penalty on every miss by dynamic scheduling. This processor has a 2.5 GHz clock.

Assume that the main memory time (which sets the miss penalty) is 50 ns. Determine the relative performance of these three processors.

Answer First, we use the miss penalty and miss rate information to compute the contribution to CPI from cache misses for each configuration. We do this with the following formula:

$$\text{Cache CPI} = \text{Misses per instruction} \times \text{Miss penalty}$$

We need to compute the miss penalties for each system:

$$\text{Miss penalty} = \frac{\text{Memory access time}}{\text{Clock cycle}}$$

The clock cycle times for the processors are 250 ps, 200 ps, and 400 ps, respectively. Hence, the miss penalties are

$$\text{Miss penalty}_1 = \frac{50 \text{ ns}}{250 \text{ ps}} = 200 \text{ cycles}$$

$$\text{Miss penalty}_2 = \frac{50 \text{ ns}}{200 \text{ ps}} = 250 \text{ cycles}$$

$$\text{Miss penalty}_3 = \frac{0.75 \times 50 \text{ ns}}{400 \text{ ps}} = 94 \text{ cycles}$$

Applying this for each cache:

$$\text{Cache CPI}_1 = 0.005 \times 200 = 1.0$$

$$\text{Cache CPI}_2 = 0.0055 \times 250 = 1.4$$

$$\text{Cache CPI}_3 = 0.01 \times 94 = 0.94$$

We know the pipeline CPI contribution for everything but processor 3; its pipeline CPI is given by:

$$\text{Pipeline CPI}_3 = \frac{1}{\text{Issue rate}} = \frac{1}{9 \times 0.5} = \frac{1}{4.5} = 0.22$$

Now we can find the CPI for each processor by adding the pipeline and cache CPI contributions:

$$\text{CPI}_1 = 0.8 + 1.0 = 1.8$$

$$\text{CPI}_2 = 1.0 + 1.4 = 2.4$$

$$\text{CPI}_3 = 0.22 + 0.94 = 1.16$$

Since this is the same architecture, we can compare instruction execution rates in millions of instructions per second (MIPS) to determine relative performance:

$$\text{Instruction execution rate} = \frac{\text{CR}}{\text{CPI}}$$

$$\text{Instruction execution rate}_1 = \frac{4000 \text{ MHz}}{1.8} = 2222 \text{ MIPS}$$

$$\text{Instruction execution rate}_2 = \frac{5000 \text{ MHz}}{2.4} = 2083 \text{ MIPS}$$

$$\text{Instruction execution rate}_3 = \frac{2500 \text{ MHz}}{1.16} = 2155 \text{ MIPS}$$

In this example, the simple two-issue static superscalar looks best. In practice, performance depends on both the CPI and clock rate assumptions.

Beyond the Limits of This Study

Like any limit study, the study we have examined in this section has its own limitations. We divide these into two classes: limitations that arise even for the

perfect speculative processor, and limitations that arise for one or more realistic models. Of course, all the limitations in the first class apply to the second. The most important limitations that apply even to the perfect model are

1. *WAW and WAR hazards through memory*—The study eliminated WAW and WAR hazards through register renaming, but not in memory usage. Although at first glance it might appear that such circumstances are rare (especially WAW hazards), they arise due to the allocation of stack frames. A called procedure reuses the memory locations of a previous procedure on the stack, and this can lead to WAW and WAR hazards that are unnecessarily limiting. Austin and Sohi [1992] examined this issue.
2. *Unnecessary dependences*—With infinite numbers of registers, all but true register data dependences are removed. There are, however, dependences arising from either recurrences or code generation conventions that introduce unnecessary true data dependences. One example of these is the dependence on the control variable in a simple for loop. Since the control variable is incremented on every loop iteration, the loop contains at least one dependence. As we show in Appendix H, loop unrolling and aggressive algebraic optimization can remove such dependent computation. Wall's study includes a limited amount of such optimizations, but applying them more aggressively could lead to increased amounts of ILP. In addition, certain code generation conventions introduce unneeded dependences, in particular the use of return address registers and a register for the stack pointer (which is incremented and decremented in the call/return sequence). Wall removes the effect of the return address register, but the use of a stack pointer in the linkage convention can cause "unnecessary" dependences. Postiff et al. [1999] explored the advantages of removing this constraint.
3. *Overcoming the data flow limit*—If value prediction worked with high accuracy, it could overcome the data flow limit. As of yet, none of the more than 100 papers on the subject has achieved a significant enhancement in ILP when using a realistic prediction scheme. Obviously, perfect data value prediction would lead to effectively infinite parallelism, since every value of every instruction could be predicted *a priori*.

For a less-than-perfect processor, several ideas have been proposed that could expose more ILP. One example is to speculate along multiple paths. This idea was discussed by Lam and Wilson [1992] and explored in the study covered in this section. By speculating on multiple paths, the cost of incorrect recovery is reduced and more parallelism can be uncovered. It only makes sense to evaluate this scheme for a limited number of branches because the hardware resources required grow exponentially. Wall [1993] provided data for speculating in both directions on up to eight branches. Given the costs of pursuing both paths, knowing that one will be thrown away (and the growing amount of useless computation as such a process is followed through multiple branches), every commercial design has instead devoted additional hardware to better speculation on the correct path.

It is critical to understand that none of the limits in this section is fundamental in the sense that overcoming them requires a change in the laws of physics! Instead, they are practical limitations that imply the existence of some formidable barriers to exploiting additional ILP. These limitations—whether they be window size, alias detection, or branch prediction—represent challenges for designers and researchers to overcome.

Attempts to break through these limits in the first five years of this century met with frustration. Some techniques produced small improvements, but often at significant increases in complexity, increases in the clock cycle, and disproportionate increases in power. In summary, designers discovered that trying to extract more ILP was simply too inefficient. We will return to this discussion in our concluding remarks.

3.11

Cross-Cutting Issues: ILP Approaches and the Memory System

Hardware versus Software Speculation

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

- To speculate extensively, we must be able to disambiguate memory references. This capability is difficult to do at compile time for integer programs that contain pointers. In a hardware-based scheme, dynamic runtime disambiguation of memory addresses is done using the techniques we saw earlier for Tomasulo's algorithm. This disambiguation allows us to move loads past stores at runtime. Support for speculative memory references can help overcome the conservatism of the compiler, but unless such approaches are used carefully, the overhead of the recovery mechanisms may swamp the advantages.
- Hardware-based speculation works better when control flow is unpredictable and when hardware-based branch prediction is superior to software-based branch prediction done at compile time. These properties hold for many integer programs. For example, a good static predictor has a misprediction rate of about 16% for four major integer SPEC92 programs, and a hardware predictor has a misprediction rate of under 10%. Because speculated instructions may slow down the computation when the prediction is incorrect, this difference is significant. One result of this difference is that even statically scheduled processors normally include dynamic branch predictors.
- Hardware-based speculation maintains a completely precise exception model even for speculated instructions. Recent software-based approaches have added special support to allow this as well.

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

The major disadvantage of supporting speculation in hardware is the complexity and additional hardware resources required. This hardware cost must be evaluated against both the complexity of a compiler for a software-based approach and the amount and usefulness of the simplifications in a processor that relies on such a compiler.

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

The Intel Itanium processor was the most ambitious computer ever designed based on the software support for ILP and speculation. It did not deliver on the hopes of the designers, especially for general-purpose, nonscientific code. As designers' ambitions for exploiting ILP were reduced in light of the difficulties discussed in [Section 3.10](#), most architectures settled on hardware-based mechanisms with issue rates of three to four instructions per clock.

Speculative Execution and the Memory System

Inherent in processors that support speculative execution or conditional instructions is the possibility of generating invalid addresses that would not occur without speculative execution. Not only would this be incorrect behavior if protection exceptions were taken, but the benefits of speculative execution would be swamped by false exception overhead. Hence, the memory system must identify speculatively executed instructions and conditionally executed instructions and suppress the corresponding exception.

By similar reasoning, we cannot allow such instructions to cause the cache to stall on a miss because again unnecessary stalls could overwhelm the benefits of speculation. Hence, these processors must be matched with nonblocking caches.

In reality, the penalty of an L2 miss is so large that compilers normally only speculate on L1 misses. Figure 2.5 on page 84 shows that for some well-behaved scientific programs the compiler can sustain multiple outstanding L2 misses to cut the L2 miss penalty effectively. Once again, for this to work the memory system behind the cache must match the goals of the compiler in number of simultaneous memory accesses.

3.12

Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput

The topic we cover in this section, multithreading, is truly a cross-cutting topic, since it has relevance to pipelining and superscalars, to graphics processing units ([Chapter 4](#)), and to multiprocessors ([Chapter 5](#)). We introduce the topic here and explore the use of multithreading to increase uniprocessor throughput by using multiple threads to hide pipeline and memory latencies. In the next chapter, we will see how multithreading provides the same advantages in GPUs, and finally, [Chapter 5](#) will explore the combination of multithreading and multiprocessing. These topics are closely interwoven, since multithreading is a primary technique for exposing more parallelism to the hardware. In a strict sense, multithreading uses thread-level parallelism, and thus is properly the subject of [Chapter 5](#), but its role in both improving pipeline utilization and in GPUs motivates us to introduce the concept here.

Although increasing performance by using ILP has the great advantage that it is reasonably transparent to the programmer, as we have seen ILP can be quite limited or difficult to exploit in some applications. In particular, with reasonable instruction issue rates, cache misses that go to memory or off-chip caches are unlikely to be hidden by available ILP. Of course, when the processor is stalled waiting on a cache miss, the utilization of the functional units drops dramatically.

Since attempts to cover long memory stalls with more ILP have limited effectiveness, it is natural to ask whether other forms of parallelism in an application could be used to hide memory delays. For example, an online transaction-processing system has natural parallelism among the multiple queries and updates that are presented by requests. Of course, many scientific applications contain natural parallelism since they often model the three-dimensional, parallel structure of nature, and that structure can be exploited by using separate threads. Even desktop applications that use modern Windows-based operating systems often have multiple active applications running, providing a source of parallelism.

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. In contrast, a more general method to exploit *thread-level parallelism* (TLP) is with a multiprocessor that has multiple independent threads operating at once and in parallel. Multithreading, however, does not duplicate the entire processor as a multiprocessor does. Instead, multithreading shares most of the processor core among a set of threads, duplicating only private state, such as the registers and program counter. As we will see in

Chapter 5, many recent processors incorporate both multiple processor cores on a single chip and provide multithreading within each core.

Duplicating the per-thread state of a processor core means creating a separate register file, a separate PC, and a separate page table for each thread. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming. In addition, the hardware must support the ability to change to a different thread relatively quickly; in particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles. Of course, for multithreading hardware to achieve performance improvements, a program must contain multiple threads (we sometimes say that the application is multithreaded) that could execute in concurrent fashion. These threads are identified either by a compiler (typically from a language with parallelism constructs) or by the programmer.

There are three main hardware approaches to multithreading. *Fine-grained multithreading* switches between threads on each clock, causing the execution of instructions from multiple threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time. One key advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls, even if the stall is only for a few cycles. The primary disadvantage of fine-grained multithreading is that it slows down the execution of an individual thread, since a thread that is ready to execute without stalls will be delayed by instructions from other threads. It trades an uncrease in multithreaded throughput for a loss in the performance (as measured by latency) of a single thread. The Sun Niagara processor, which we examine shortly, uses simple fine-grained multithreading, as do the Nvidia GPUs, which we look at in the next chapter.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level two or three cache misses. This change relieves the need to have thread-switching be essentially free and is much less likely to slow down the execution of any one thread, since instructions from other threads will only be issued when a thread encounters a costly stall.

Coarse-grained multithreading suffers, however, from a major drawback: It is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grained multithreading. Because a CPU with coarse-grained multithreading issues instructions from a single thread, when a stall occurs the pipeline will see a bubble before the new thread begins executing. Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of very high-cost stalls, where pipeline refill is negligible compared to the stall time. Several research projects have explored coarse grained multithreading, but no major current processors use this technique.

The most common implementation of multithreading is called *Simultaneous multithreading* (SMT). Simultaneous multithreading is a variation on fine-grained multithreading that arises naturally when fine-grained multithreading is implemented on top of a multiple-issue, dynamically scheduled processor. As

with other forms of multithreading, SMT uses thread-level parallelism to hide long-latency events in a processor, thereby increasing the usage of the functional units. The key insight in SMT is that register renaming and dynamic scheduling allow multiple instructions from independent threads to be executed without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

Figure 3.28 conceptually illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configurations:

- A superscalar with no multithreading support
- A superscalar with coarse-grained multithreading
- A superscalar with fine-grained multithreading
- A superscalar with simultaneous multithreading

In the superscalar without multithreading support, the use of issue slots is limited by a lack of ILP, including ILP to hide memory latency. Because of the length of L2 and L3 cache misses, much of the processor can be left idle.

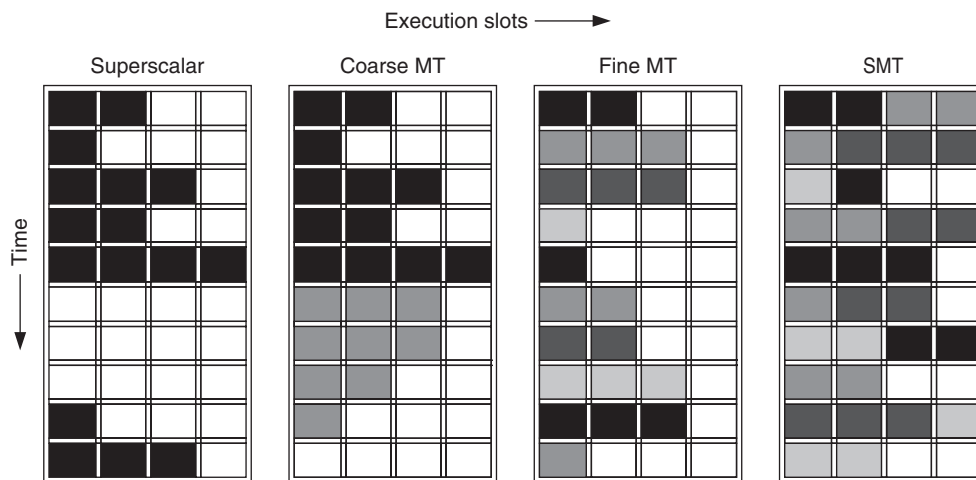


Figure 3.28 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 and T2 (aka Niagara) processors are fine-grained multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has eight threads, the Power7 has four, and the Intel i7 has two. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. This switching reduces the number of completely idle clock cycles. In a coarse-grained multithreaded processor, however, thread switching only occurs when there is a stall. Because the new thread has a start-up period, there are likely to be some fully idle cycles remaining.

In the fine-grained case, the interleaving of threads can eliminate fully empty slots. In addition, because the issuing thread is changed on every clock cycle, longer latency operations can be hidden. Because instruction issue and execution are connected, a thread can only issue as many instructions as are ready. With a narrow issue width this is not a problem (a cycle is either occupied or not), which is why fine-grained multithreading works perfectly for a single issue processor, and SMT would make no sense. Indeed, in the Sun T2, there are two issues per clock, but they are from different threads. This eliminates the need to implement the complex dynamic scheduling approach and relies instead on hiding latency with more threads.

If one implements fine-grained threading on top of a multiple-issue dynamically schedule processor, the result is SMT. In all existing SMT implementations, all issues come from one thread, although instructions from different threads can initiate execution in the same cycle, using the dynamic scheduling hardware to determine what instructions are ready. Although [Figure 3.28](#) greatly simplifies the real operation of these processors, it does illustrate the potential performance advantages of multithreading in general and SMT in wider issue, dynamically scheduled processors.

Simultaneous multithreading uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support the mechanism, including a large virtual register set. Multithreading can be built on top of an out-of-order processor by adding a per-thread renaming table, keeping separate PCs, and providing the capability for instructions from multiple threads to commit.

Effectiveness of Fine-Grained Multithreading on the Sun T1

In this section, we use the Sun T1 processor to examine the ability of multithreading to hide latency. The T1 is a fine-grained multithreaded multicore microprocessor introduced by Sun in 2005. What makes T1 especially interesting is that it is almost totally focused on exploiting thread-level parallelism (TLP) rather than instruction-level parallelism (ILP). The T1 abandoned the intense focus on ILP (just shortly after the most aggressive ILP processors ever were introduced), returned to a simple pipeline strategy, and focused on exploiting TLP, using both multiple cores and multithreading to produce throughput.

Each T1 processor contains eight processor cores, each supporting four threads. Each processor core consists of a simple six-stage, single-issue pipeline (a standard five-stage RISC pipeline like that of [Appendix C](#), with one stage added for thread switching). T1 uses fine-grained multithreading (but not SMT), switching to a new thread on each clock cycle, and threads that are idle because they are waiting due to

Characteristic	Sun T1
Multiprocessor and multithreading support	Eight cores per chip; four threads per core. Fine-grained thread scheduling. One shared floating-point unit for eight cores. Supports only on-chip multiprocessing.
Pipeline structure	Simple, in-order, six-deep pipeline with three-cycle delays for loads and branches.
L1 caches	16 KB instructions; 8 KB data. 64-byte block size. Miss to L2 is 23 cycles, assuming no contention.
L2 caches	Four separate L2 caches, each 750 KB and associated with a memory bank. 64-byte block size. Miss to main memory is 110 clock cycles assuming no contention.
Initial implementation	90 nm process; maximum clock rate of 1.2 GHz; power 79 W; 300 M transistors; 379 mm ² die.

Figure 3.29 A summary of the T1 processor.

a pipeline delay or cache miss are bypassed in the scheduling. The processor is idle only when all four threads are idle or stalled. Both loads and branches incur a three-cycle delay that can only be hidden by other threads. A single set of floating-point functional units is shared by all eight cores, as floating-point performance was not a focus for T1. [Figure 3.29](#) summarizes the T1 processor.

T1 Multithreading Unicore Performance

The T1 makes TLP its focus, both through the multithreading on an individual core and through the use of many simple cores on a single die. In this section, we will look at the effectiveness of the T1 in increasing the performance of a single core through fine-grained multithreading. In [Chapter 5](#), we will return to examine the effectiveness of combining multithreading with multiple cores.

To examine the performance of the T1, we use three server-oriented benchmarks: TPC-C, SPECJBB (the SPEC Java Business Benchmark), and SPECWeb99. Since multiple threads increase the memory demands from a single processor, they could overload the memory system, leading to reductions in the potential gain from multithreading. [Figure 3.30](#) shows the relative increase in the miss rate and the observed miss latency when executing with one thread per core versus executing four threads per core for TPC-C. Both the miss rates and the miss latencies increase, due to increased contention in the memory system. The relatively small increase in miss latency indicates that the memory system still has unused capacity.

By looking at the behavior of an average thread, we can understand the interaction among the threads and their ability to keep a core busy. [Figure 3.31](#) shows the percentage of cycles for which a thread is executing, ready but not executing, and not ready. Remember that not ready does not imply that the core with that thread is stalled; it is only when all four threads are not ready that the core will stall.

Threads can be not ready due to cache misses, pipeline delays (arising from long latency instructions such as branches, loads, floating point, or integer multiply/divide), and a variety of smaller effects. [Figure 3.32](#) shows the relative

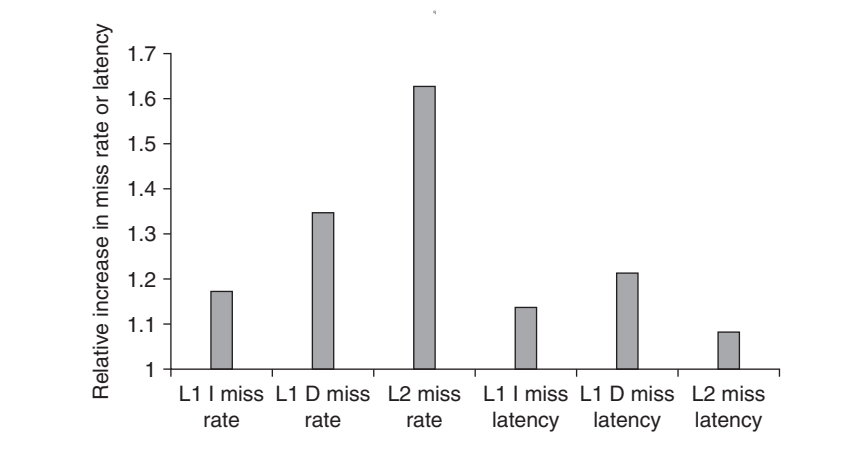


Figure 3.30 The relative change in the miss rates and miss latencies when executing with one thread per core versus four threads per core on the TPC-C benchmark. The latencies are the actual time to return the requested data after a miss. In the four-thread case, the execution of other threads could potentially hide much of this latency.

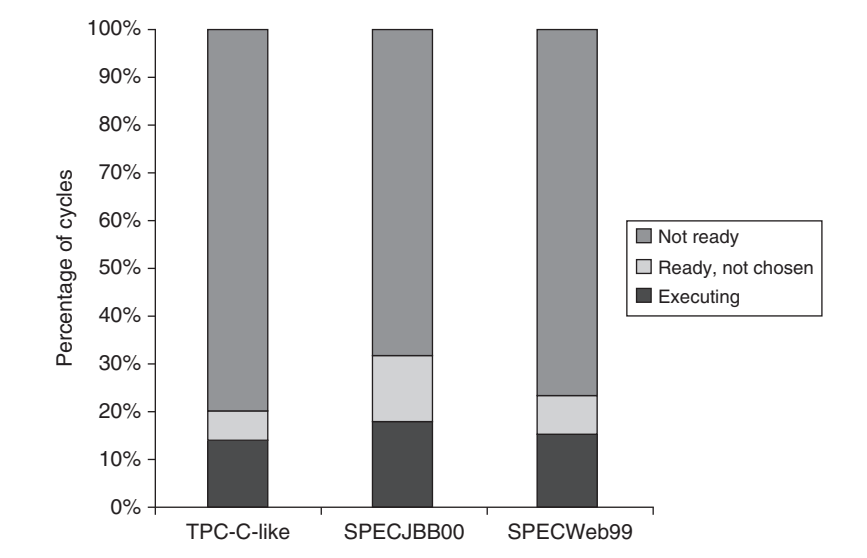


Figure 3.31 Breakdown of the status on an average thread. “Executing” indicates the thread issues an instruction in that cycle. “Ready but not chosen” means it could issue but another thread has been chosen, and “not ready” indicates that the thread is awaiting the completion of an event (a pipeline delay or cache miss, for example).

frequency of these various causes. Cache effects are responsible for the thread not being ready from 50% to 75% of the time, with L1 instruction misses, L1 data misses, and L2 misses contributing roughly equally. Potential delays from the pipeline (called “pipeline delay”) are most severe in SPECJBB and may arise from its higher branch frequency.

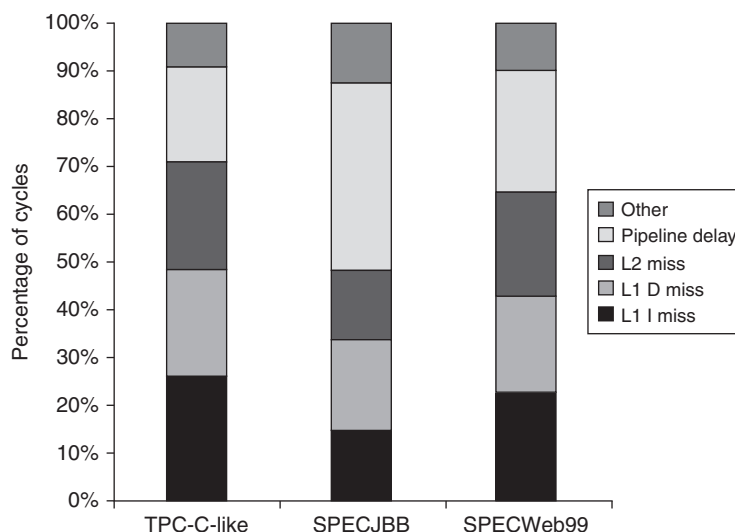


Figure 3.32 The breakdown of causes for a thread being not ready. The contribution to the “other” category varies. In TPC-C, store buffer full is the largest contributor; in SPEC-JBB, atomic instructions are the largest contributor; and in SPECWeb99, both factors contribute.

Benchmark	Per-thread CPI	Per-core CPI
TPC-C	7.2	1.80
SPECJBB	5.6	1.40
SPECWeb99	6.6	1.65

Figure 3.33 The per-thread CPI, the per-core CPI, the effective eight-core CPI, and the effective IPC (inverse of CPI) for the eight-core T1 processor.

Figure 3.33 shows the per-thread and per-core CPI. Because T1 is a fine-grained multithreaded processor with four threads per core, with sufficient parallelism the ideal effective CPI per thread would be four, since that would mean that each thread was consuming one cycle out of every four. The ideal CPI per core would be one. In 2005, the IPC for these benchmarks running on aggressive ILP cores would have been similar to that seen on a T1 core. The T1 core, however, was very modest in size compared to the aggressive ILP cores of 2005, which is why the T1 had eight cores compared to the two to four offered on other processors of the same vintage. As a result, in 2005 when it was introduced, the Sun T1 processor had the best performance on integer applications with extensive TLP and demanding memory performance, such as SPECJBB and transaction processing workloads.

Effectiveness of Simultaneous Multithreading on Superscalar Processors

A key question is, How much performance can be gained by implementing SMT? When this question was explored in 2000–2001, researchers assumed that dynamic superscalars would get much wider in the next five years, supporting six to eight issues per clock with speculative dynamic scheduling, many simultaneous loads and stores, large primary caches, and four to eight contexts with simultaneous issue and retirement from multiple contexts. No processor has gotten close to this level.

As a result, simulation research results that showed gains for multiprogrammed workloads of two or more times are unrealistic. In practice, the existing implementations of SMT offer only two to four contexts with fetching and issue from only one, and up to four issues per clock. The result is that the gain from SMT is also more modest.

For example, in the Pentium 4 Extreme, as implemented in HP-Compaq servers, the use of SMT yields a performance improvement of 1.01 when running the SPECintRate benchmark and about 1.07 when running the SPECfpRate benchmark. Tuck and Tullsen [2003] reported that, on the SPLASH parallel benchmarks, they found single-core multithreaded speedups ranging from 1.02 to 1.67, with an average speedup of about 1.22.

With the availability of recent extensive and insightful measurements done by Esmailzadeh et al. [2011], we can look at the performance and energy benefits of using SMT in a single i7 core using a set of multithreaded applications. The benchmarks we use consist of a collection of parallel scientific applications and a set of multithreaded Java programs from the DaCapo and SPEC Java suite, as summarized in Figure 3.34. The Intel i7 supports SMT with two threads. Figure 3.35 shows the performance ratio and the energy efficiency ratio of the these benchmarks run on one core of the i7 with SMT turned off and on. (We plot the energy efficiency ratio, which is the inverse of energy consumption, so that, like speedup, a higher ratio is better.)

The harmonic mean of the speedup for the Java benchmarks is 1.28, despite the two benchmarks that see small gains. These two benchmarks, *pjbb2005* and *tradebeans*, while multithreaded, have limited parallelism. They are included because they are typical of a multithreaded benchmark that might be run on an SMT processor with the hope of extracting some performance, which they find in limited amounts. The PARSEC benchmarks obtain somewhat better speedups than the full set of Java benchmarks (harmonic mean of 1.31). If *tradebeans* and *pjbb2005* were omitted, the Java workload would actually have significantly better speedup (1.39) than the PARSEC benchmarks. (See the discussion of the implication of using harmonic mean to summarize the results in the caption of Figure 3.36.)

Energy consumption is determined by the combination of speedup and increase in power consumption. For the Java benchmarks, on average, SMT delivers the same energy efficiency as non-SMT (average of 1.0), but it is brought down by the two poor performing benchmarks; without *tradebeans* and *pjbb2005*, the average

blackscholes	Prices a portfolio of options with the Black-Scholes PDE
bodytrack	Tracks a markerless human body
canneal	Minimizes routing cost of a chip with cache-aware simulated annealing
facesim	Simulates motions of a human face for visualization purposes
ferret	Search engine that finds a set of images similar to a query image
fluidanimate	Simulates physics of fluid motion for animation with SPH algorithm
raytrace	Uses physical simulation for visualization
streamcluster	Computes an approximation for the optimal clustering of data points
swaptions	Prices a portfolio of swap options with the Heath–Jarrow–Morton framework
vips	Applies a series of transformations to an image
x264	MPG-4 AVC/H.264 video encoder
eclipse	Integrated development environment
lusearch	Text search tool
sunflow	Photo-realistic rendering system
tomcat	Tomcat servlet container
tradebeans	Tradebeans Daytrader benchmark
xalan	An XSLT processor for transforming XML documents
pjbb2005	Version of SPEC JBB2005 (but fixed in problem size rather than time)

Figure 3.34 The parallel benchmarks used here to examine multithreading, as well as in [Chapter 5](#) to examine multiprocessing with an i7. The top half of the chart consists of PARSEC benchmarks collected by Bienia et al. [2008]. The PARSEC benchmarks are meant to be indicative of compute-intensive, parallel applications that would be appropriate for multicore processors. The lower half consists of multithreaded Java benchmarks from the DaCapo collection (see Blackburn et al. [2006]) and pjbb2005 from SPEC. All of these benchmarks contain some parallelism; other Java benchmarks in the DaCapo and SPEC Java workloads use multiple threads but have little or no true parallelism and, hence, are not used here. See Esmaeilzadeh et al. [2011] for additional information on the characteristics of these benchmarks, relative to the measurements here and in [Chapter 5](#).

energy efficiency for the Java benchmarks is 1.06, which is almost as good as the PARSEC benchmarks. In the PARSEC benchmarks, SMT reduces energy by $1 - (1/1.08) = 7\%$. Such energy-reducing performance enhancements are *very difficult* to find. Of course, the static power associated with SMT is paid in both cases, thus the results probably slightly overstate the energy gains.

These results clearly show that SMT in an aggressive speculative processor with extensive support for SMT can improve performance in an energy efficient fashion, which the more aggressive ILP approaches have failed to do. In 2011, the balance between offering multiple simpler cores and fewer more sophisticated cores has shifted in favor of more cores, with each core typically being a three- to four-issue superscalar with SMT supporting two to four threads. Indeed, Esmaeilzadeh et al. [2011] show that the energy improvements from SMT are even larger on the Intel i5 (a processor similar to the i7, but with smaller caches and a lower clock rate) and the Intel Atom (an 80×86 processor designed for the netbook market and described in [Section 3.14](#)).

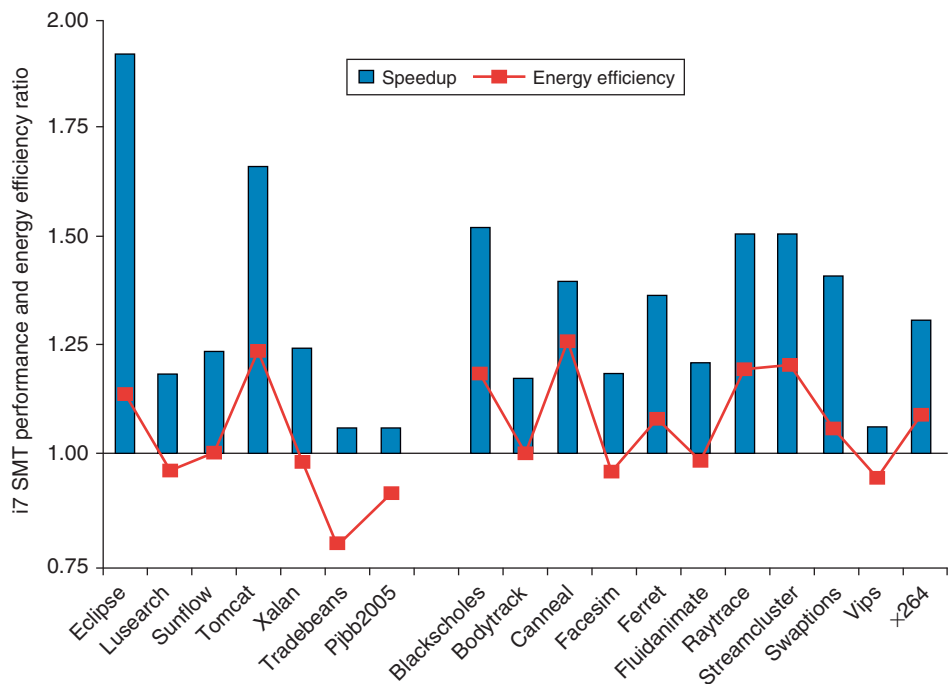


Figure 3.35 The speedup from using multithreading on one core on an i7 processor averages 1.28 for the Java benchmarks and 1.31 for the PARSEC benchmarks (using an unweighted harmonic mean, which implies a workload where the total time spent executing each benchmark in the single-threaded base set was the same). The energy efficiency averages 0.99 and 1.07, respectively (using the harmonic mean). Recall that anything above 1.0 for energy efficiency indicates that the feature reduces execution time by more than it increases average power. Two of the Java benchmarks experience little speedup and have significant negative energy efficiency because of this. Turbo Boost is off in all cases. These data were collected and analyzed by Esmailzadeh et al. [2011] using the Oracle (Sun) HotSpot build 16.3-b01 Java 1.6.0 Virtual Machine and the gcc v4.4.1 native compiler.

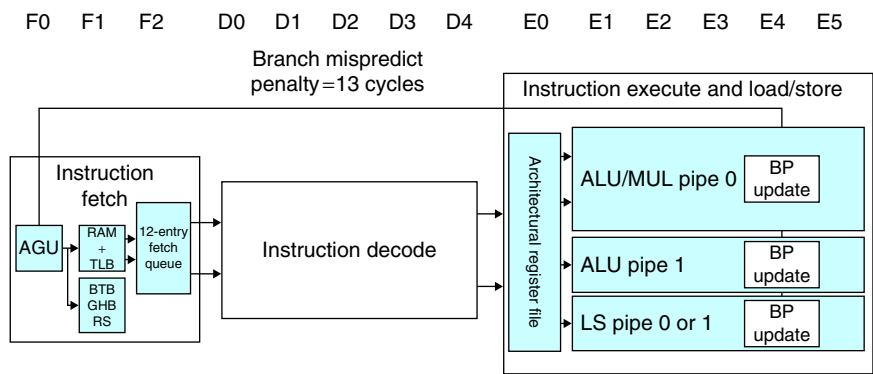


Figure 3.36 The basic structure of the A8 pipeline is 13 stages. Three cycles are used for instruction fetch and four for instruction decode, in addition to a five-cycle integer pipeline. This yields a 13-cycle branch misprediction penalty. The instruction fetch unit tries to keep the 12-entry instruction queue filled.

3.13

Putting It All Together: The Intel Core i7 and ARM Cortex-A8

In this section we explore the design of two multiple issue processors: the ARM Cortex-A8 core, which is used as the basis for the Apple A9 processor in the iPad, as well as the processor in the Motorola Droid and the iPhones 3GS and 4, and the Intel Core i7, a high-end, dynamically scheduled, speculative processor, intended for high-end desktops and server applications. We begin with the simpler processor.

The ARM Cortex-A8

The A8 is a dual-issue, statically scheduled superscalar with dynamic issue detection, which allows the processor to issue one or two instructions per clock. [Figure 3.36](#) shows the basic pipeline structure of the 13-stage pipeline.

The A8 uses a dynamic branch predictor with a 512-entry two-way set associative branch target buffer and a 4K-entry global history buffer, which is indexed by the branch history and the current PC. In the event that the branch target buffer misses, a prediction is obtained from the global history buffer, which can then be used to compute the branch address. In addition, an eight-entry return stack is kept to track return addresses. An incorrect prediction results in a 13-cycle penalty as the pipeline is flushed.

[Figure 3.37](#) shows the instruction decode pipeline. Up to two instructions per clock can be issued using an in-order issue mechanism. A simple scoreboard structure is used to track when an instruction can issue. A pair of dependent instructions can be processed through the issue logic, but, of course, they will be serialized at the scoreboard, unless they can be issued so that the forwarding paths can resolve the dependence.

[Figure 3.38](#) shows the execution pipeline for the A8 processor. Either instruction 1 or instruction 2 can go to the load/store pipeline. Fully bypassing is supported among the pipelines. The ARM Cortex-A8 pipeline uses a simple two-issue statically scheduled superscalar to allow reasonably high clock rate with lower power. In contrast, the i7 uses a reasonably aggressive, four-issue dynamically scheduled speculative pipeline structure.

Performance of the A8 Pipeline

The A8 has an ideal CPI of 0.5 due to its dual-issue structure. Pipeline stalls can arise from three sources:

1. Functional hazards, which occur because two adjacent instructions selected for issue simultaneously use the same functional pipeline. Since the A8 is statically scheduled, it is the compiler's task to try to avoid such conflicts. When they cannot be avoided, the A8 can issue at most one instruction in that cycle.

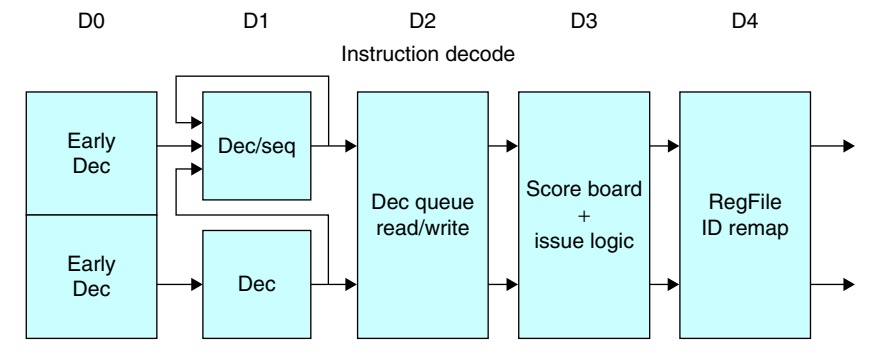


Figure 3.37 The five-stage instruction decode of the A8. In the first stage, a PC produced by the fetch unit (either from the branch target buffer or the PC incrementer) is used to retrieve an 8-byte block from the cache. Up to two instructions are decoded and placed into the decode queue; if neither instruction is a branch, the PC is incremented for the next fetch. Once in the decode queue, the scoreboard logic decides when the instructions can issue. In the issue, the register operands are read; recall that in a simple scoreboard, the operands always come from the registers. The register operands and opcode are sent to the instruction execution portion of the pipeline.

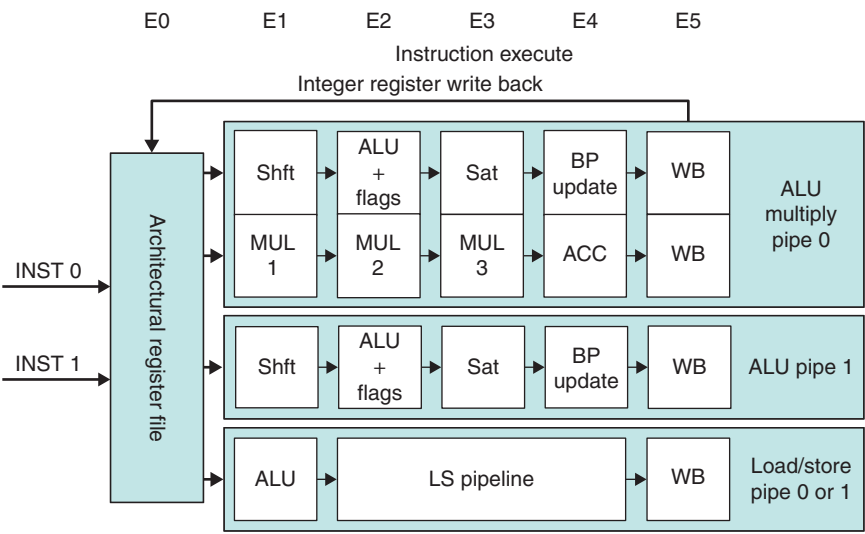


Figure 3.38 The five-stage instruction decode of the A8. Multiply operations are always performed in ALU pipeline 0.

2. Data hazards, which are detected early in the pipeline and may stall either both instructions (if the first cannot issue, the second is always stalled) or the second of a pair. The compiler is responsible for preventing such stalls when possible.
3. Control hazards, which arise only when branches are mispredicted.

In addition to pipeline stalls, L1 and L2 misses both cause stalls.

Figure 3.39 shows an estimate of the factors that contribute to the actual CPI for the Minnespec benchmarks, which we saw in Chapter 2. As we can see, pipeline delays rather than memory stalls are the major contributor to the CPI. This result is partially due to the effect that Minnespec has a smaller cache footprint than full SPEC or other large programs.

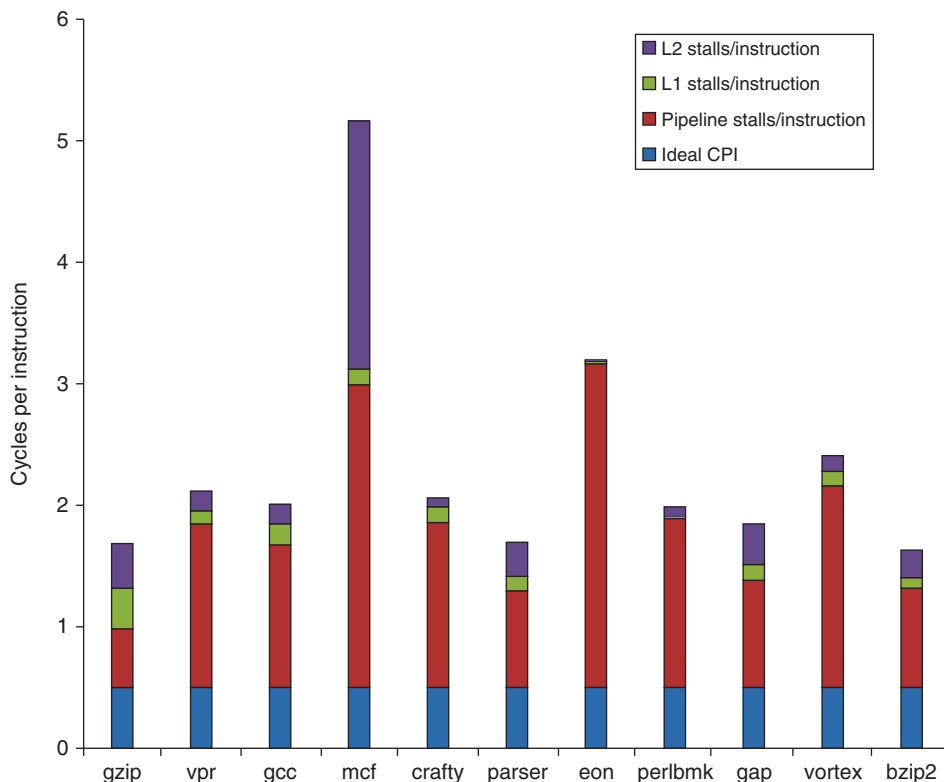


Figure 3.39 The estimated composition of the CPI on the ARM A8 shows that pipeline stalls are the primary addition to the base CPI. *eon* deserves some special mention, as it does integer-based graphics calculations (ray tracing) and has very few cache misses. It is computationally intensive with heavy use of multiples, and the single multiply pipeline becomes a major bottleneck. This estimate is obtained by using the L1 and L2 miss rates and penalties to compute the L1 and L2 generated stalls per instruction. These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards plus minor effects such as way misprediction.

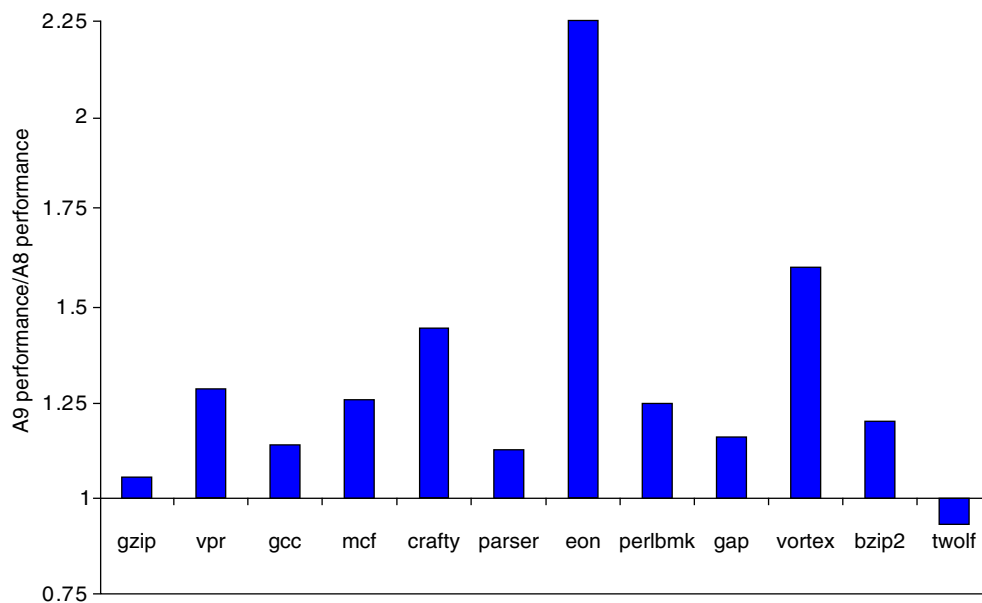


Figure 3.40 The performance ratio for the A9 compared to the A8, both using a 1 GHz clock and the same size caches for L1 and L2, shows that the A9 is about 1.28 times faster. Both runs use a 32 KB primary cache and a 1 MB secondary cache, which is 8-way set associative for the A8 and 16-way for the A9. The block sizes in the caches are 64 bytes for the A8 and 32 bytes for the A9. As mentioned in the caption of [Figure 3.39](#), eon makes intensive use of integer multiply, and the combination of dynamic scheduling and a faster multiply pipeline significantly improves performance on the A9. twolf experiences a small slowdown, likely due to the fact that its cache behavior is worse with the smaller L1 block size of the A9.

The insight that the pipeline stalls created significant performance losses probably played a key role in the decision to make the ARM Cortex-A9 a dynamically scheduled superscalar. The A9, like the A8, issues up to two instructions per clock, but it uses dynamic scheduling and speculation. Up to four pending instructions (two ALUs, one load/store or FP/multimedia, and one branch) can begin execution in a clock cycle. The A9 uses a more powerful branch predictor, instruction cache prefetch, and a nonblocking L1 data cache. [Figure 3.40](#) shows that the A9 outperforms the A8 by a factor of 1.28 on average, assuming the same clock rate and virtually identical cache configurations.

The Intel Core i7

The i7 uses an aggressive out-of-order speculative microarchitecture with reasonably deep pipelines with the goal of achieving high instruction throughput by combining multiple issue and high clock rates. [Figure 3.41](#) shows the overall structure of the i7 pipeline. We will examine the pipeline by starting with

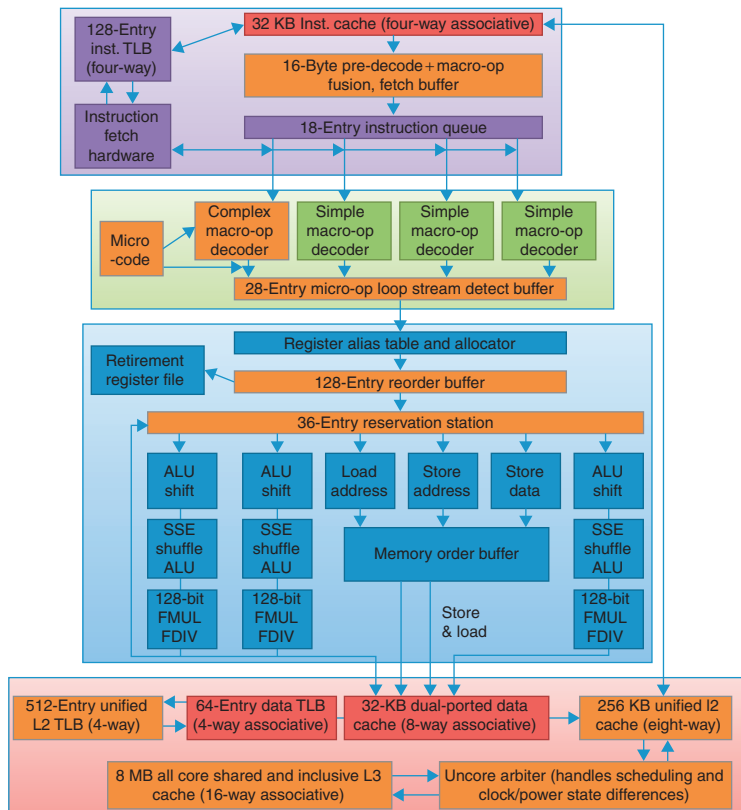


Figure 3.41 The Intel Core i7 pipeline structure shown with the memory system components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 cycles. There are 48 load and 32 store buffers. The six independent functional units can each begin execution of a ready micro-op in the same cycle.

instruction fetch and continuing on to instruction commit, following steps labeled on the figure.

1. **Instruction fetch**—The processor uses a multilevel branch target buffer to achieve a balance between speed and prediction accuracy. There is also a return address stack to speed up function return. Mispredictions cause a penalty of about 15 cycles. Using the predicted address, the instruction fetch unit fetches 16 bytes from the instruction cache.
2. **The 16 bytes are placed in the predecode instruction buffer**—In this step, a process called macro-op fusion is executed. *Macro-op fusion* takes instruction combinations such as compare followed by a branch and fuses them into a single operation. The predecode stage also breaks the 16 bytes into individual x86 instructions. This predecode is nontrivial since the length of an x86

instruction can be from 1 to 17 bytes and the predecoder must look through a number of bytes before it knows the instruction length. Individual x86 instructions (including some fused instructions) are placed into the 18-entry instruction queue.

3. **Micro-op decode**—Individual x86 instructions are translated into micro-ops. Micro-ops are simple MIPS-like instructions that can be executed directly by the pipeline; this approach of translating the x86 instruction set into simple operations that are more easily pipelined was introduced in the Pentium Pro in 1997 and has been used since. Three of the decoders handle x86 instructions that translate directly into one micro-op. For x86 instructions that have more complex semantics, there is a microcode engine that is used to produce the micro-op sequence; it can produce up to four micro-ops every cycle and continues until the necessary micro-op sequence has been generated. The micro-ops are placed according to the order of the x86 instructions in the 28-entry micro-op buffer.
4. **The micro-op buffer preforms *loop stream detection* and *microfusion***—If there is a small sequence of instructions (less than 28 instructions or 256 bytes in length) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-ops from the buffer, eliminating the need for the instruction fetch and instruction decode stages to be activated. Microfusion combines instruction pairs such as load/ALU operation and ALU operation/store and issues them to a single reservation station (where they can still issue independently), thus increasing the usage of the buffer. In a study of the Intel Core architecture, which also incorporated microfusion and macrofusion, Bird et al. [2007] discovered that microfusion had little impact on performance, while macrofusion appears to have a modest positive impact on integer performance and little impact on floating-point performance.
5. **Perform the basic instruction issue**—Looking up the register location in the register tables, renaming the registers, allocating a reorder buffer entry, and fetching any results from the registers or reorder buffer before sending the micro-ops to the reservation stations.
6. **The i7 uses a 36-entry centralized reservation station shared by six functional units**. Up to six micro-ops may be dispatched to the functional units every clock cycle.
7. **Micro-ops are executed by the individual function units and then results are sent back to any waiting reservation station as well as to the register retirement unit**, where they will update the register state, once it is known that the instruction is no longer speculative. The entry corresponding to the instruction in the reorder buffer is marked as complete.
8. **When one or more instructions at the head of the reorder buffer have been marked as complete**, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.

Performance of the i7

In earlier sections, we examined the performance of the i7's branch predictor and also the performance of SMT. In this section, we look at single-thread pipeline performance. Because of the presence of aggressive speculation as well as non-blocking caches, it is difficult to attribute the gap between idealized performance and actual performance accurately. As we will see, relatively few stalls occur because instructions cannot issue. For example, only about 3% of the loads are delayed because no reservation station is available. Most losses come either from branch mispredicts or cache misses. The cost of a branch mispredict is 15 cycles, while the cost of an L1 miss is about 10 cycles; L2 misses are slightly more than three times as costly as an L1 miss, and L3 misses cost about 13 times what an L1 miss costs (130–135 cycles)! Although the processor will attempt to find alternative instructions to execute for L3 misses and some L2 misses, it is likely that some of the buffers will fill before the miss completes, causing the processor to stop issuing instructions.

To examine the cost of mispredicts and incorrect speculation, [Figure 3.42](#) shows the fraction of the work (measured by the numbers of micro-ops dispatched into the pipeline) that do not retire (i.e., their results are annulled),

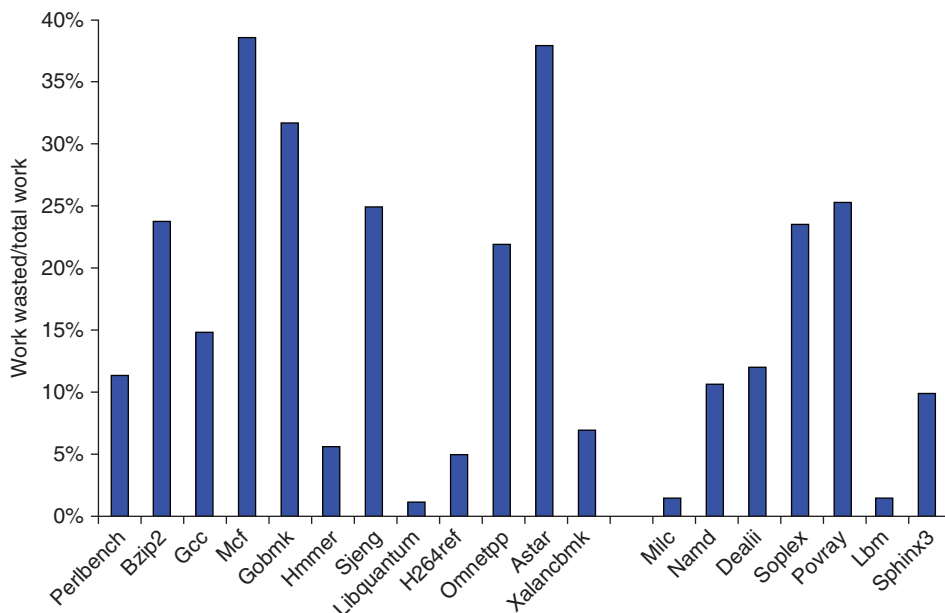


Figure 3.42 The amount of “wasted work” is plotted by taking the ratio of dispatched micro-ops that do not graduate to all dispatched micro-ops. For example, the ratio is 25% for *sjeng*, meaning that 25% of the dispatched and executed micro-ops are thrown away. The data in this section were collected by Professor Lu Peng and Ph.D. student Ying Zhang, both of Louisiana State University.

relative to all micro-op dispatches. For *sjeng*, for example, 25% of the work is wasted, since 25% of the dispatched micro-ops are never retired.

Notice that the wasted work in some cases closely matches the branch misprediction rates shown in Figure 3.5 on page 167, but in several instances, such as *mcf*, the wasted work seems relatively larger than the misprediction rate. In such cases, a likely explanation arises from the memory behavior. With the very high data cache miss rates, *mcf* will dispatch many instructions during an incorrect speculation as long as sufficient reservation stations are available for the stalled memory references. When the branch misprediction is detected, the micro-ops corresponding to these instructions will be flushed, but there will be congestion around the caches, as speculated memory references try to complete. There is no simple way for the processor to halt such cache requests once they are initiated.

Figure 3.43 shows the overall CPI for the 19 SPECCPU2006 benchmarks. The integer benchmarks have a CPI of 1.06 with very large variance (0.67 standard deviation). MCF and OMNETPP are the major outliers, both having a CPI over 2.0 while most other benchmarks are close to, or less than, 1.0 (*gcc*, the next highest, is 1.23). This variance derives from differences in the accuracy of branch

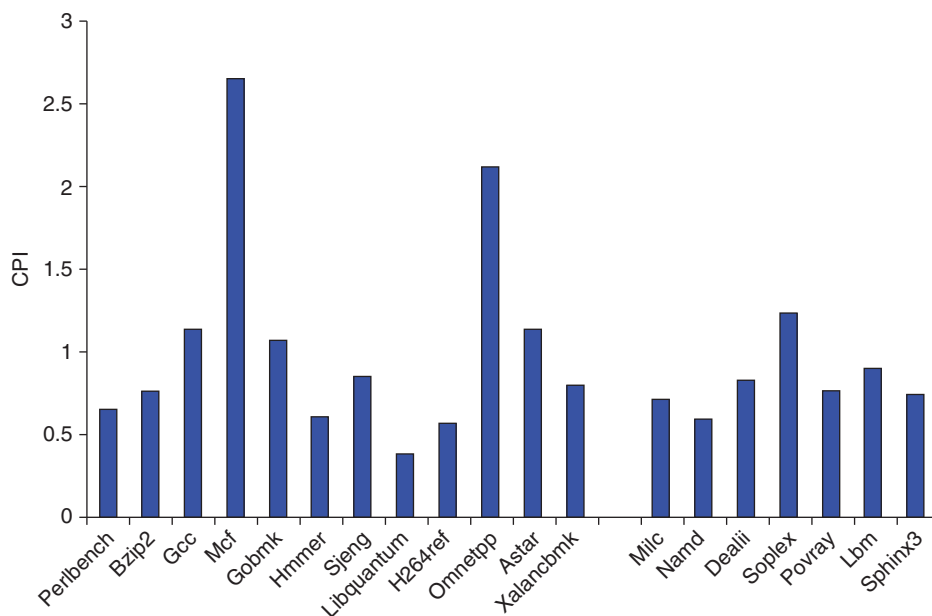


Figure 3.43 The CPI for the 19 SPECCPU2006 benchmarks shows an average CPI for 0.83 for both the FP and integer benchmarks, although the behavior is quite different. In the integer case, the CPI values range from 0.44 to 2.66 with a standard deviation of 0.77, while the variation in the FP case is from 0.62 to 1.38 with a standard deviation of 0.25. The data in this section were collected by Professor Lu Peng and Ph.D. student Ying Zhang, both of Louisiana State University.

prediction and in cache miss rates. For the integer benchmarks, the L2 miss rate is the best predictor of CPI, and the L3 miss rate (which is very small) has almost no effect.

The FP benchmarks achieve higher performance with a lower average CPI (0.89) and a lower standard deviation (0.25). For the FP benchmarks, L1 and L2 are equally important in determining the CPI, while L3 plays a smaller but significant role. While the dynamic scheduling and nonblocking capabilities of the i7 can hide some miss latency, cache memory behavior is still a major contributor. This reinforces the role of multithreading as another way to hide memory latency.

3.14

Fallacies and Pitfalls

Our few fallacies focus on the difficulty of predicting performance and energy efficiency and extrapolating from single measures such as clock rate or CPI. We also show that different architectural approaches can have radically different behaviors for different benchmarks.

Fallacy *It is easy to predict the performance and energy efficiency of two different versions of the same instruction set architecture, if we hold the technology constant.*

Intel manufactures a processor for the low-end Netbook and PMD space that is quite similar in its microarchitecture of the ARM A8, called the Atom 230. Interestingly, the Atom 230 and the Core i7 920 have both been fabricated in the same 45 nm Intel technology. [Figure 3.44](#) summarizes the Intel Core i7, the ARM Cortex-A8, and Intel Atom 230. These similarities provide a rare opportunity to directly compare two radically different microarchitectures for the same instruction set while holding constant the underlying fabrication technology. Before we do the comparison, we need to say a little more about the Atom 230.

The Atom processors implement the x86 architecture using the standard technique of translating x86 instructions into RISC-like instructions (as every x86 implementation since the mid-1990s has done). Atom uses a slightly more powerful microoperation, which allows an arithmetic operation to be paired with a load or a store. This means that on average for a typical instruction mix only 4% of the instructions require more than one microoperation. The microoperations are then executed in a 16-deep pipeline capable of issuing two instructions per clock, in order, as in the ARM A8. There are dual-integer ALUs, separate pipelines for FP add and other FP operations, and two memory operation pipelines, supporting more general dual execution than the ARM A8 but still limited by the in-order issue capability. The Atom 230 has a 32 KB instruction cache and a 24 KB data cache, both backed by a shared 512 KB L2 on the same die. (The Atom 230 also supports multithreading with two threads, but we will consider only one single threaded comparisons.) [Figure 3.46](#) summarizes the i7, A8, and Atom processors and their key characteristics.

We might expect that these two processors, implemented in the same technology and with the same instruction set, would exhibit predictable behavior, in

Area	Specific characteristic	Intel i7 920	ARM A8	Intel Atom 230
		Four cores, each with FP	One core, no FP	One core, with FP
Physical chip properties	Clock rate	2.66 GHz	1 GHz	1.66 GHz
	Thermal design power	130 W	2 W	4 W
	Package	1366-pin BGA	522-pin BGA	437-pin BGA
Memory system	TLB	Two-level All four-way set associative 128 I/64 D 512 L2	One-level fully associative 32 I/32 D	Two-level All four-way set associative 16 I/16 D 64 L2
	Caches	Three-level 32 KB/32 KB 256 KB 2–8 MB	Two-level 16/16 or 32/32 KB 128 KB–1MB	Two-level 32/24 KB 512 KB
	Peak memory BW	17 GB/sec	12 GB/sec	8 GB/sec
	Peak issue rate	4 ops/clock with fusion	2 ops/clock	2 ops/clock
Pipeline structure	Pipeline scheduling	Speculating out of order	In-order dynamic issue	In-order dynamic issue
	Branch prediction	Two-level	Two-level 512-entry BTB 4K global history 8-entry return stack	Two-level

Figure 3.44 An overview of the four-core Intel i7 920, an example of a typical Arm A8 processor chip (with a 256 MB L2, 32K L1s, and no floating point), and the Intel ARM 230 clearly showing the difference in design philosophy between a processor intended for the PMD (in the case of ARM) or netbook space (in the case of Atom) and a processor for use in servers and high-end desktops. Remember, the i7 includes four cores, each of which is several times higher in performance than the one-core A8 or Atom. All these processors are implemented in a comparable 45 nm technology.

terms of relative performance and energy consumption, meaning that power and performance would scale close to linearly. We examine this hypothesis using three sets of benchmarks. The first sets is a group of Java, single-threaded benchmarks that come from the DaCapo benchmarks, and the SPEC JVM98 benchmarks (see Esmaeilzadeh et al. [2011] for a discussion of the benchmarks and measurements). The second and third sets of benchmarks are from SPEC CPU2006 and consist of the integer and FP benchmarks, respectively.

As we can see in Figure 3.45, the i7 significantly outperforms the Atom. All benchmarks are at least four times faster on the i7, two SPECFP benchmarks are over ten times faster, and one SPECINT benchmark runs over eight times faster!

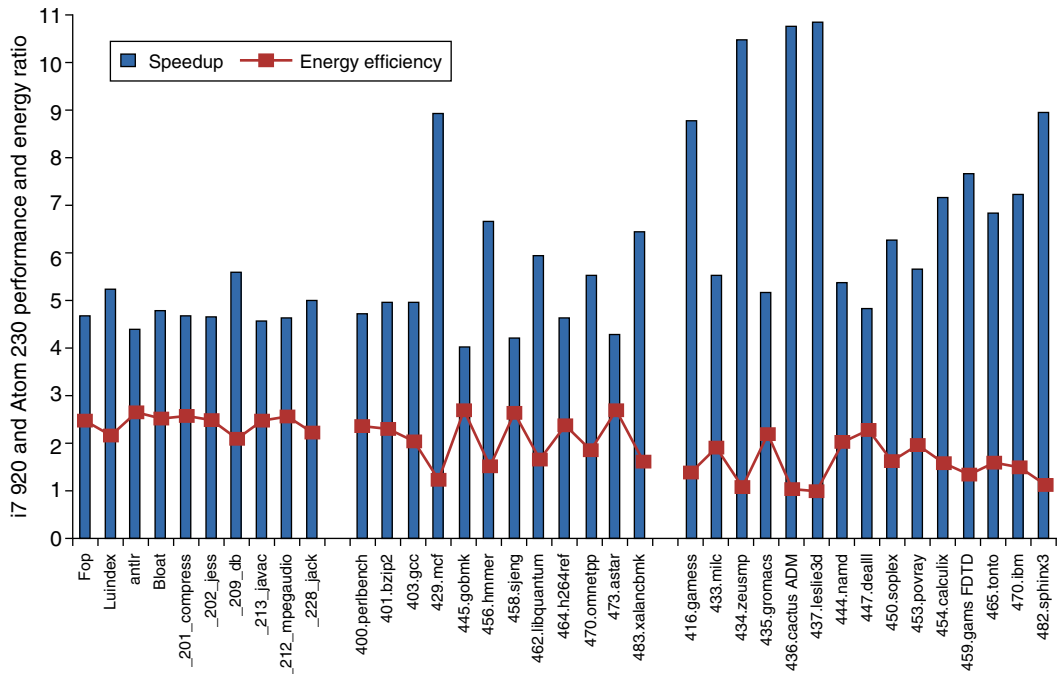


Figure 3.45 The relative performance and energy efficiency for a set of single-threaded benchmarks shows the i7 920 is 4 to over 10 times faster than the Atom 230 but that it is about 2 times *less* power efficient on average! Performance is shown in the columns as i7 relative to Atom, which is execution time (i7)/execution time (Atom). Energy is shown with the line as Energy (Atom)/Energy (i7). The i7 never beats the Atom in energy efficiency, although it is essentially as good on four benchmarks, three of which are floating point. The data shown here were collected by Esmailzadeh et al. [2011]. The SPEC benchmarks were compiled with optimization on using the standard Intel compiler, while the Java benchmarks use the Sun (Oracle) Hotspot Java VM. Only one core is active on the i7, and the rest are in deep power saving mode. Turbo Boost is used on the i7, which increases its performance advantage but slightly decreases its relative energy efficiency.

Since the ratio of clock rates of these two processors is 1.6, most of the advantage comes from a much lower CPI for the i7: a factor of 2.8 for the Java benchmarks, a factor of 3.1 for the SPECINT benchmarks, and a factor of 4.3 for the SPECFP benchmarks.

But, the average power consumption for the i7 is just under 43 W, while the average power consumption of the Atom is 4.2 W, or about one-tenth of the power! Combining the performance and power leads to a energy efficiency advantage for the Atom that is typically more than 1.5 times better and often 2 times better! This comparison of two processors using the same underlying technology makes it clear that the performance advantages of an aggressive superscalar with dynamic scheduling and speculation come with a significant disadvantage in energy efficiency.

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

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

The key is that it is the product of CPI and clock rate that determines performance. A high clock rate obtained by deeply pipelining the CPU must maintain a low CPI to get the full benefit of the faster clock. Similarly, a simple processor with a high clock rate but a low CPI may be slower.

As we saw in the previous fallacy, performance and energy efficiency can diverge significantly among processors designed for different environments even when they have the same ISA. In fact, large differences in performance can show up even within a family of processors from the same company all designed for high-end applications. Figure 3.46 shows the integer and FP performance of two different implementations of the x86 architecture from Intel, as well as a version of the Itanium architecture, also by Intel.

The Pentium 4 was the most aggressively pipelined processor ever built by Intel. It used a pipeline with over 20 stages, had seven functional units, and cached micro-ops rather than x86 instructions. Its relatively inferior performance given the aggressive implementation, was a clear indication that the attempt to exploit more ILP (there could easily be 50 instructions in flight) had failed. The Pentium's power consumption was similar to the i7, although its transistor count was lower, as its primary caches were half as large as the i7, and it included only a 2 MB secondary cache with no tertiary cache.

The Intel Itanium is a VLIW-style architecture, which despite the potential decrease in complexity compared to dynamically scheduled superscalars, never attained competitive clock rates with the mainline x86 processors (although it appears to achieve an overall CPI similar to that of the i7). In examining these results, the reader should be aware that they use different implementation technologies, giving the i7 an advantage in terms of transistor speed and hence clock rate for an equivalently pipelined processor. Nonetheless, the wide variation in performance—more than three times between the Pentium and i7—is astonishing. The next pitfall explains where a significant amount of this advantage comes from.

Processor	Clock rate	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	3.8 GHz	11.5	12.2
Intel Itanium -2	1.66 GHz	14.5	17.3
Intel i7	3.3 GHz	35.5	38.4

Figure 3.46 Three different Intel processors vary widely. Although the Itanium processor has two cores and the i7 four, only one core is used in the benchmarks.

Pitfall *Sometimes bigger and dumber is better.*

Much of the attention in the early 2000s went to building aggressive processors to exploit ILP, including the Pentium 4 architecture, which used the deepest pipeline ever seen in a microprocessor, and the Intel Itanium, which had the highest peak issue rate per clock ever seen. What quickly became clear was that the main limitation in exploiting ILP often turned out to be the memory system. Although speculative out-of-order pipelines were fairly good at hiding a significant fraction of the 10- to 15-cycle miss penalties for a first-level miss, they could do very little to hide the penalties for a second-level miss that, when going to main memory, were likely to be 50 to 100 clock cycles.

The result was that these designs never came close to achieving the peak instruction throughput despite the large transistor counts and extremely sophisticated and clever techniques. The next section discusses this dilemma and the turning away from more aggressive ILP schemes to multicore, but there was another change that exemplifies this pitfall. Instead of trying to hide even more memory latency with ILP, designers simply used the transistors to build much larger caches. Both the Itanium 2 and the i7 use three-level caches compared to the two-level cache of the Pentium 4, and the third-level caches are 9 MB and 8 MB compared to the 2 MB second-level cache of the Pentium 4. Needless to say, building larger caches is a lot easier than designing the 20+ -stage Pentium 4 pipeline and, from the data in [Figure 3.46](#), seems to be more effective.

3.15**Concluding Remarks: What's Ahead?**

As 2000 began, the focus on exploiting instruction-level parallelism was at its peak. Intel was about to introduce Itanium, a high-issue-rate statically scheduled processor that relied on a VLIW-like approach with intensive compiler support. MIPS, Alpha, and IBM processors with dynamically scheduled speculative execution were in their second generation and had gotten wider and faster. The Pentium 4, which used speculative scheduling, had also been announced that year with seven functional units and a pipeline more than 20 stages deep. But there were storm clouds on the horizon.

Research such as that covered in [Section 3.10](#) was showing that pushing ILP much further would be extremely difficult, and, while peak instruction throughput rates had risen from the first speculative processors some 3 to 5 years earlier, sustained instruction execution rates were growing much more slowly.

The next five years were telling. The Itanium turned out to be a good FP processor but only a mediocre integer processor. Intel still produces the line, but there are not many users, the clock rate lags the mainline Intel processors, and Microsoft no longer supports the instruction set. The Intel Pentium 4, while achieving good performance, turned out to be inefficient in terms of performance/watt (i.e., energy use), and the complexity of the processor made it unlikely that further advances would be possible by increasing the issue rate. The

end of a 20-year road of achieving new performance levels in microprocessors by exploiting ILP had come. The Pentium 4 was widely acknowledged to have gone beyond the point of diminishing returns, and the aggressive and sophisticated Netburst microarchitecture was abandoned.

By 2005, Intel and all the other major processor manufacturers had revamped their approach to focus on multicore. Higher performance would be achieved through thread-level parallelism rather than instruction-level parallelism, and the responsibility for using the processor efficiently would largely shift from the hardware to the software and the programmer. This change was the most significant change in processor architecture since the early days of pipelining and instruction-level parallelism some 25+ years earlier.

During the same period, designers began to explore the use of more data-level parallelism as another approach to obtaining performance. SIMD extensions enabled desktop and server microprocessors to achieve moderate performance increases for graphics and similar functions. More importantly, graphics processing units (GPUs) pursued aggressive use of SIMD, achieving significant performance advantages for applications with extensive data-level parallelism. For scientific applications, such approaches represent a viable alternative to the more general, but less efficient, thread-level parallelism exploited in multicores. The next chapter explores these developments in the use of data-level parallelism.

Many researchers predicted a major retrenchment in the use of ILP, predicting that two issue superscalar processors and larger numbers of cores would be the future. The advantages, however, of slightly higher issue rates and the ability of speculative dynamic scheduling to deal with unpredictable events, such as level-one cache misses, led to moderate ILP being the primary building block in multicore designs. The addition of SMT and its effectiveness (both for performance and energy efficiency) further cemented the position of the moderate issue, out-of-order, speculative approaches. Indeed, even in the embedded market, the newest processors (e.g., the ARM Cortex-A9) have introduced dynamic scheduling, speculation, and wider issues rates.

It is highly unlikely that future processors will try to increase the width of issue significantly. It is simply too inefficient both from the viewpoint of silicon utilization and power efficiency. Consider the data in [Figure 3.47](#) that show the most recent four processors in the IBM Power series. Over the past decade, there has been a modest improvement in the ILP support in the Power processors, but the dominant portion of the increase in transistor count (a factor of almost 7 from the Power 4 to the Power7) went to increasing the caches and the number of cores per die. Even the expansion in SMT support seems to be more a focus than an increase in the ILP throughput: The ILP structure from Power4 to Power7 went from 5 issues to 6, from 8 functional units to 12 (but not increasing from the original 2 load/store units), while the SMT support went from nonexistent to 4 threads/processor. It seems clear that even for the most advanced ILP processor in 2011 (the Power7), the focus has moved beyond instruction-level parallelism. The next two chapters focus on approaches that exploit data-level and thread-level parallelism.

	Power4	Power5	Power6	Power7
Introduced	2001	2004	2007	2010
Initial clock rate (GHz)	1.3	1.9	4.7	3.6
Transistor count (M)	174	276	790	1200
Issues per clock	5	5	7	6
Functional units	8	8	9	12
Cores/chip	2	2	2	8
SMT threads	0	2	2	4
Total on-chip cache (MB)	1.5	2	4.1	32.3

Figure 3.47 Characteristics of four IBM Power processors. All except the Power6 were dynamically scheduled, which is static, and in-order, and all the processors support two load/store pipelines. The Power6 has the same functional units as the Power5 except for a decimal unit. Power7 uses DRAM for the L3 cache.

3.16

Historical Perspective and References

Section L.5 (available online) features a discussion on the development of pipelining and instruction-level parallelism. We provide numerous references for further reading and exploration of these topics. Section L.5 covers both [Chapter 3](#) and Appendix H.

Case Studies and Exercises by Jason D. Bakos and Robert P. Colwell

Case Study: 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 3](#) 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 3.48 provides a sequence of instructions and list of latencies.

- 3.1 [10] <1.8, 3.1, 3.2> What would be the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.48 if no new instruction's execution could be initiated until the previous instruction's 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 one-cycle branch delay slot.
- 3.2 [10] <1.8, 3.1, 3.2> 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 3.48 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 requires two `<stall>` cycles to be inserted into the code sequence. Think of it this way: A one-cycle instruction has latency 1 + 0, meaning zero extra wait states. So, latency 1 + 1 implies one stall cycle; latency 1 + N has N extra stall cycles.
- 3.3 [15] <3.6, 3.7> 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

Latencies beyond single cycle			
Loop:	LD	F2,0(RX)	Memory LD +4
I0:	DIVD	F8,F2,F0	Memory SD +1
I1:	MULTD	F2,F6,F2	Integer ADD, SUB +0
I2:	LD	F4,0(Ry)	Branches +1
I3:	ADDD	F4,F0,F4	ADDD +1
I4:	ADDD	F10,F8,F2	MULTD +5
I5:	ADDI	Rx,Rx,#8	DIVD +12
I6:	ADDI	Ry,Ry,#8	
I7:	SD	F4,0(Ry)	
I8:	SUB	R20,R4,Rx	
I9:	BNZ	R20,Loop	

Figure 3.48 Code and latencies for Exercises 3.1 through 3.6.

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 dependency. Now how many cycles does the loop require?

- 3.4 [10] <3.6, 3.7> In the multiple-issue design of Exercise 3.3, you may have recognized some subtle issues. Even though the two pipelines have the exact same instruction repertoire, they are neither 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 3.48 that demonstrate this hazard.
- 3.5 [20] <3.7> Reorder the instructions to improve performance of the code in Figure 3.48. Assume the two-pipe machine in Exercise 3.3 and that the out-of-order completion issues of Exercise 3.4 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?
- 3.6 [10/10/10] <3.1, 3.2> 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.
 - a. [10] <3.1, 3.2> In your reordered code from Exercise 3.5, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?
 - b. [10] <3.1, 3.2> Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance. Hand-unroll two iterations of the loop in your reordered code from Exercise 3.5.
 - c. [10] <3.1, 3.2> 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 instructions; if you were actually unrolling the loop, you would have to reassign registers to prevent collisions between the iterations.)
- 3.7 [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 multiple iterations' register usages from colliding, we rename their registers. Figure 3.49 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 that

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

Figure 3.49 Sample code for register renaming practice.

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

Figure 3.50 *Hint:* Expected output of register renaming.

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 `src` (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 3.49](#). 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 3.50](#).)

- 3.8 [20] <3.4> Exercise 3.7 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 `dest` 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 3.51](#). 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

I0:	SUBD	F1, F2, F3
I1:	ADDD	F4, F1, F2
I2:	MULTD	F6, F4, F1
I3:	DIVD	F0, F2, F6

Figure 3.51 Sample code for superscalar register renaming.

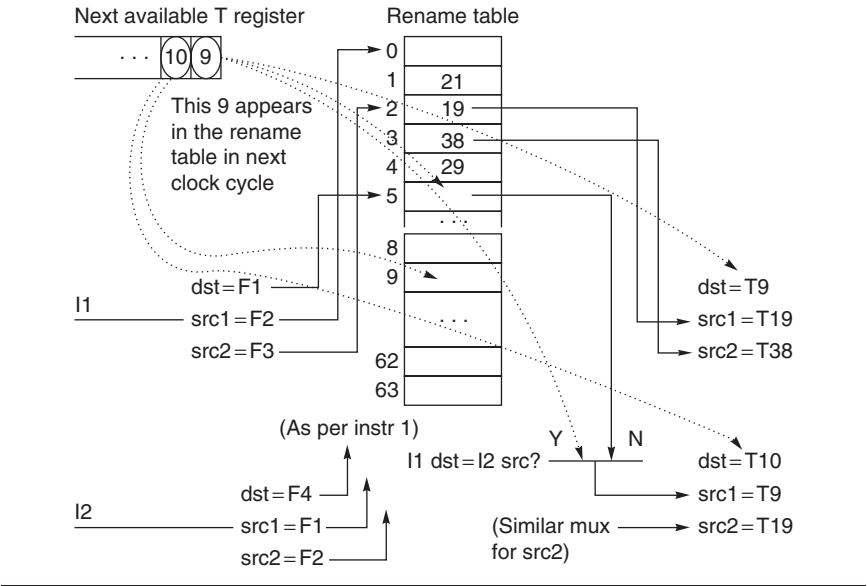


Figure 3.52 Rename table and on-the-fly register substitution logic for superscalar machines. (Note that src is source, and dest is destination.)

interinstruction 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 3.52 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 shown in Figure 3.51. Assume the table starts out with every entry equal to its index (T0 = 0; T1 = 1, ...).

- 3.9 [5] <3.4> 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?

- 3.10
- [20] <3.4, 3.9> Very long instruction word (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 3.53. 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.
- 3.11
- [10/10/10] <3.3> Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write-back) and the code in Figure 3.54. All ops are one 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.
- a.
- [10] <3.3> How many clock cycles per loop iteration are lost to branch overhead?
- b.
- [10] <3.3> 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?
- c.
- [10] <3.3> Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?

Loop:	LW	R4,0(R0) ;	ADDI	R11,R3,#1
	LW	R5,8(R1) ;	ADDI	R20,R0,#1
	<stall>			
	ADDI	R10,R4,#1;		
	SW	R7,0(R6) ;	SW	R9,8(R8)
	ADDI	R2,R2,#8		
	SUB	R4,R3,R2		
	BNZ	R4,Loop		

Figure 3.53 Sample VLIW code with two adds, two loads, and two stalls.

Loop:	LW	R3,0(R0)
	LW	R1,0(R3)
	ADDI	R1,R1,#1
	SUB	R4,R3,R2
	SW	R1,0(R3)
	BNZ	R4, Loop

Figure 3.54 Code loop for Exercise 3.11.

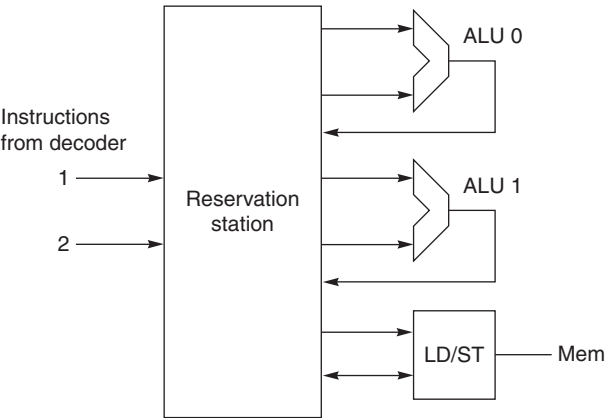


Figure 3.55 An out-of-order microarchitecture.

- 3.12 [15/20/20/10/20] <3.4, 3.7, 3.14> Let's consider what dynamic scheduling might achieve here. Assume a microarchitecture as shown in Figure 3.55. Assume that the arithmetic-logical units (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).
- a. [15] <3.4> Suppose all of the instructions from the sequence in Figure 3.48 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 read-after-write and write-after-write hazards. Assume the same functional unit latencies as in Figure 3.48.)
 - b. [20] <3.4> 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 3.48. 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—no bypassing.) How many clock cycles does the code sequence take?

- c. [20] <3.4> 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 one clock cycle to dispatch any op, and assume functional unit latencies are as they were for Exercise 3.2. 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] <3.14> 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? or (4) Cutting the longest latency in half? What's the speedup?
- e. [20] <3.7> 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.)

Exercises

- 3.13 [25] <3.13> In this exercise, you will explore performance trade-offs between three processors that each employ different types of multithreading. Each of these processors is superscalar, uses in-order pipelines, requires a fixed three-cycle stall following all loads and branches, and has identical L1 caches. Instructions from the same thread issued in the same cycle are read in program order and must not contain any data or control dependences.
 - Processor A is a superscalar SMT architecture, capable of issuing up to two instructions per cycle from two threads.
 - Processor B is a fine MT architecture, capable of issuing up to four instructions per cycle from a single thread and switches threads on any pipeline stall.