3

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 in 1851, later named "The America's Cup," which was the inspiration for John Cocke's naming of an IBM research processor as "America," the first superscalar processor, and a precursor to the PowerPC.

"*Thus, the IA-64 gambles that, in the future, power will not be the critical limitation, and massive resources…will not penalize clock speed, path length, or CPI factors. My view is clearly skeptical…*"

Marty Hopkins (2000), IBM Fellow and Early RISC pioneer commenting in 2000 on the new Intel Itanium, a joint development of Intel and HP. The Itanium used a static ILP approach (see Appendix H) and was a massive investment for Intel. It never accounted for more than 0.5% of Intel's microprocessor sales.

Computer Architecture. https://doi.org/10.1016/B978-0-443-15406-5.00003-1

3.1 Introduction

All processors since about 1985 have used pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called *instruction-level parallelism* (ILP), because 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 limitations 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. The first two sections introduce a large number of concepts, which we build on throughout this chapter and the next.

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. This chapter introduces the key concepts for both approaches since many of the techniques developed for one approach have been exploited within a design relying primarily on the other. Processors using the dynamic, hardware-based approach, including all recent Intel and AMD processors and many Arm processors, now dominate in the desktop and server markets. The same approaches are now common in processors found in tablets and most smartphones. In the IoT space, where power and cost constraints dominate performance goals, designers exploit lower levels of ILP. Aggressive compiler-based approaches have been attempted numerous times beginning in the 1980s and most recently in the Intel Itanium series, introduced in 1999. Despite enormous efforts, these approaches have been successful only in domain-specific architectures or in well-structured scientific applications with significant data-level parallelism (see Chapter 7).

This chapter also includes a discussion of the limitations of ILP approaches as it was such limitations that directly led to the movement toward multicore. Understanding the limitations remains important in balancing the use of ILP and thread-level parallelism (TLP).

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

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

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

The *ideal pipeline CPI* is a measure of the maximum performance attainable by the implementation. By reducing each of the terms on the right-hand side, we decrease the overall pipeline CPI or, alternatively, increase the IPC (instructions per clock). The preceding equation 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 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 RISC 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. Because 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-*

Technique	Reduces	Section
Forwarding and bypassing	Data hazard stalls	C.2
Simple branch scheduling and prediction	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling	Data hazard stalls from true dependencies	C.7
Loop unrolling	Control hazards stalls	3.2
Issuing multiple instructions per cycle	Ideal CPI	3.3, 3.8
Deep pipelining	Higher clock frequency	3.3
Advanced branch prediction	Control hazard stalls	3.4
Register renaming	Data hazards stalls from output dependences (WAW) and anti-dependences (WAR)	3.5
Dynamic scheduling with speculation	Data hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls due to memory instructions	3.7
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, control hazard stalls	H.4, H.5
Simultaneous multithreading (SMT)	Idle cycles due to stalls	3.11

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.

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 instruction overlap.

We will examine a number of techniques for converting such loop-level parallelism into ILP. 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 3.3 to 3.8).

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 2−16). 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 preceding 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 vector instructions to load the vectors `x` and `y` from memory, one vector instruction to add the two vectors, and a vector 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 ILP, we must determine which instructions can be executed in parallel. If two instructions are *independent*, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls, assuming the pipeline has sufficient resources (and thus 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 a preceding 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 `add x1,x1,x1`) is not considered a dependence.

For example, consider the following RISC-V code sequence that increments a vector of values in memory, starting at address `0(x1)` and ending with the last element at address `0(x2)`, by a scalar in register `f2`.

```
Loop:  fld     f0,0(x1)    //f0=array element
       fadd.d  f4,f0,f2    //add scalar in f2
       fsd     f4,0(x1)    //store result
       addi    x1,x1,-8    //decrement pointer 8 bytes
       bne     x1,x2,Loop  //branch x1≠x2
```

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

```
Loop: fld     f0,0(x1)    //f0=array element
      fadd.d  f4,f0,f2    //add scalar in f2
      fsd     f4,0(x1)    //store result
```

and integer data:

```
      addi    x1,x1,-8    //decrement pointer
                          //8 bytes (per DW)
      bne     x1,x2,Loop//branch x1ax2
```

In both of the preceding dependent sequences, as shown by the arrows, each instruction depends on the previous one. The arrows here and in the 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, or else the program will not execute correctly. The presence of a data dependence in an instruction sequence reflects a data dependence in the source code from which the instruction sequence was generated. The effect of the original data dependence must be preserved.

Dependences are a property of *programs*. Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the *pipeline organization*. This difference is critical to understanding how ILP 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 a pitfall at the end of this chapter and in Appendix H in more detail.

Because a data dependence can limit the amount of ILP 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 through a register, detecting the dependence is straightforward because the register names are fixed in the instruction encodings, although it gets more complicated when branches intervene, and correctness concerns force a compiler or hardware to be conservative.

Dependences that flow through memory locations occur as load instructions consume the data values produced by earlier store instructions. These dependences are more difficult to detect because two addresses may refer to the same location but look different: for example, `100(x4)` and `20(x6)` 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 `100(x6)` and `20(x4)` may alternate between being the same and being different over time), 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 171 there is an antidependence between `fsd` and `addi` on register `x1`.

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, because 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 that 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.

Data Hazards

Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards. A hazard exists whenever there is a name or data dependence between two instructions, and the two execute with sufficient overlap to change the order of access to the register operand or memory location 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 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 preceding instruction *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 register or memory destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

- WAR *(write after read)*—*j* tries to write a destination before it is read by *i,* so *i* incorrectly gets the *new* value. This hazard arises from an antidependence. WAR hazards cannot occur in most static issue pipelines—even deeper pipelines or floating-point pipelines—because all reads are early (in ID 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 the 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 the correct 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. However, we may be willing to execute instructions that should not have been executed, 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:

```
add x2,x3,x4
beq x2,x0,L1
ld x1,0(x2)
L1:
```

In this case it is easy to see that if we do not maintain the data dependence involving `x2`, 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 `beq` and the `ld`; it is only the control dependence. To allow us to reorder these instructions (and still preserve the data dependence), we want to just ignore the exception when the branch is taken. In Section 3.3 we will look at a hardware technique, *speculation*, which allows us to overcome the limitations imposed by control dependencies including 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 because 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:

```
 add x1,x2,x3
 beq x4,x0,L
 sub x1,x5,x6
L: …
 or x7,x1,x8
```

In this example the value of `x1` 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 `add` and `sub` 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 `x1` computed by the `sub` should be used by the `or`, and if the branch is taken, the value of `x1` computed by the `add` 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 `sub` 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 3.3.

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

```
        add   x1, x2, x3
        beq   x12, x0, skip
        sub   x4, x5, x6
        add   x5, x4, x9
skip:   or    x7, x8, x9
```

Suppose we knew that the register destination of the `sub` instruction (`x4`) was unused after the instruction labeled `skip`. (The property of whether a value will be used by an upcoming instruction is called *liveness*.) If `x4` were unused, then changing the value of `x4` just before the branch would not affect the data flow because `x4` would be *dead* (rather than live) in the code region after `skip`. Thus, if `x4` were dead and the existing `sub` instruction could not generate an exception (other than those from which the processor resumes the same process), we could move the `sub` instruction before the branch because the data flow could not be affected by this change.

If the branch is taken, the `sub` 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, because 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, we will specifically 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 software and hardware techniques, which we examine in Sections 3.2 and 3.4, respectively.

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 later 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 ILP.

Basic Pipeline Scheduling and Loop Unrolling

To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of independent 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 section 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 of double-precision (64-bit) floating-point numbers:

```
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, which shows how we can use the parallelism to improve its performance for a RISC-V pipeline with the given latencies.

The first step is to translate the preceding segment to RISC-V assembly language. In the following code segment register `x1` is initialized with the address of

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 between two instructions needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a FP load to a store is 0 because 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 (which includes ALU operation to branch).

the element in the array with the highest address, and `f2` contains the scalar value *s*. Register `x2` is initialized so that `x2+8` is the address of the last array element to operate on.

The straightforward RISC-V code, not scheduled for the pipeline, looks like this:

```
Loop: fld f0,0(x1) //f0=array element
      fadd.d f4,f0,f2 //add scalar in f2
      fsd f4,0(x1) //store result
      addi x1,x1,-8 //decrement pointer by 8 bytes (64-bit numbers)
      bne x1,x2,Loop //branch x1≠x2
```

Let's start by seeing how well this loop will run when it is scheduled on the simple pipeline for RISC-V with the latencies in Figure 3.2.

Example Show how the execution of the loop would look on the RISC-V pipeline, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for delays from floating-point operations.

Answer Without any scheduling, the loop will execute as follows, taking nine cycles per array element:

```
                Clock cycle issued
Loop: fld f0,0(x1) 1
      stall 2
      fadd.d f4,f0,f2 3
      stall 4
      stall 5
      fsd f4,0(x1) 6
      addi x1,x1,-8 7
      bne x1,x2,Loop 8
```

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

```
Loop: fld f0,0(x1)
      addi x1,x1,-8
      fadd.d f4,f0,f2
      stall
      stall
      fsd f4,8(x1)
      bne x1,x2,Loop
```

The stalls after `fadd.d` are for use by the `fsd`, and repositioning the `addi` prevents the stall after the `fld`.

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 `addi` 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 reduces the frequency of branches, it allows instructions from multiple iterations to be scheduled together. In this case we can eliminate the data use stalls by interleaving instructions from the loop iterations that we unrolled. 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 `x1 – x2` (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 `addi` instructions and dropping the unnecessary `bne` operations that are duplicated during unrolling. Note that `x2` must now be initialized so that `x2+32` is the starting address of the last four elements.

```
Loop: fld f0,0(x1) // code for 1st element
      fadd.d f4,f0,f2
      fsd f4,0(x1) //drop addi & bne
      fld f6,-8(x1) // code for 2nd element
      fadd.d f8,f6,f2
      fsd f8,-8(x1) //drop addi & bne
      fld f0,-16(x1) // code for 3rd element
      fadd.d f12,f0,f2
      fsd f12,-16(x1) //drop addi & bne
      fld f14,-24(x1) // code for 4th element
      fadd.d f16,f14,f2
      fsd f16,-24(x1)
      addi x1,x1,-32
      bne x1,x2,Loop
```

We have eliminated three branches and three decrements of `x1`. The addresses on the loads and stores have been compensated to allow the `addi` instructions on `x1` 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. Loop unrolling is normally done early in the compilation process so that redundant computations can be exposed and eliminated by the optimizer.

Without scheduling, every FP load or operation in the unrolled loop is followed by a dependent operation and thus will cause a stall. This unrolled loop will run in 26 clock cycles—each `fld` has 1 stall, each `fadd.d` has 2, plus 14 instruction issue cycles—or 6.5 clock cycles for each of the four elements, but it can be scheduled to improve performance significantly.

In real programs we do not usually know the upper bound on the loop. Suppose it is *n,* and we want to unroll the loop to make *k* copies of the body. Instead of a single unrolled loop, we generate a pair of consecutive loops. The first executes (*n* mod *k*) times and has a body that is the original loop. The second is the unrolled body surrounded by an outer loop that iterates (*n*/*k*) times. (As we will 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 in Figure 3.2.

Answer

```
Loop: fld f0,0(x1)
      fld f6,-8(x1)
      fld f10,-16(x1)
      fld f14,-24(x1)
      fadd.d f4,f0,f2
      fadd.d f8,f6,f2
      fadd.d f12,f10,f2
      fadd.d f16,f14,f2
      fsd f4,0(x1)
      fsd f8,-8(x1)
      fsd f12,16(x1)
      fsd f16,8(x1)
      addi x1,x1,-32
      bne x1,x2,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 8 cycles per element before any unrolling or scheduling and 6.5 cycles when unrolled but not scheduled.

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 independent instructions that can be scheduled to minimize the stalls; the preceding code has no stalls. Scheduling the loop in this fashion necessitates realizing that the loads and stores are independent and can be interchanged.

Summary of 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 ILP 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 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) register pressure. 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 `addi`, 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 element to 1/4.

A second limit to unrolling is the resulting growth in code size. 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 its advantages because it leads to 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 superscalar and VLIW processors explored later in this chapter.

3.3 Key ILP Concepts in Modern Superscalar Processors

Superscalar processors are processors that with ideal CPI of less than 1 that discover and exploit ILP using hardware techniques. Their high performance even with moderately optimized programs has made them the dominant processor choice in the server, desktop, laptop, and cellphone domains. However, their limitations and design complexity have led chip vendors to combine superscalar ideas with techniques focusing on data-level parallelism (see SIMD in Chapter 4) and TLP (see multicore in Chapter 5).

Processors for popular instruction set architectures (ISAs) such as x86, Arm, and RISC-V must faithfully implement the sequential semantics of these ISAs. The execution of a program must be functionally equivalent to using a simple processor that executes instructions in the original program order with no overlap. Functional equivalency covers the values produced in register and memory locations, the exceptions raised, and the order in which the values and exceptions produced become visible. As we will see soon, many superscalar processors maintain functional equivalency even though they execute instructions in the order dictated by data dependences and not in the original program order. These designs are called *out-of-order (OOO)* superscalar processors. Since most modern superscalar processors are OOO pipelines, the terms superscalar and OOO superscalar are assumed by many to be synonymous. Later in this chapter, we will also describe low-power, in-order superscalar designs that execute instructions in the order defined in the program.

Fundamental compiler optimizations—such as redundant code elimination, strength reduction, and register allocation—are beneficial for superscalar processors as well as scalar processors. However, these processors rely less on advanced compiler techniques, such as loop unrolling, as they feature hardware

mechanisms that apply similar optimizations within a window of tens to a few hundreds of instructions. New superscalar processors will deliver performance gains for existing binary programs that were not recompiled and reoptimized to match their pipeline. This benefit has been a key factor in the commercial success of superscalar architectures.

Deeper Pipelining and Multiple Issue

Superscalar processors improve performance over the simple pipelines we have seen so far by targeting two factors in the CPU execution time formula (CPU time = IC • CPI • CCT):

- *Deeper pipelining*: The simple processor in Appendix C uses a five-stage pipeline. Superscalar processors use deeper pipelines, in the range of 10–20 stages in contemporary designs. The deeper pipelines allow for shorter clock cycle times, and clock frequencies are now in the 1–5 GHz range. While simple operations, such as adding two integer operands, still occupy a single pipeline stage, longer-latency operations—such as multiplication, floating-point operations, and instruction or data cache accesses—use multiple stages, typically two to four. However, roughly half of the stages in most pipelines are used for the complex control logic needed to discover and dynamically schedule ILP.
- *Multiple or wide issue*: The simple processor in Appendix C initiates one instruction per clock cycle; hence the best-case pipeline CPI is 1. Superscalar processors attempt to initiate multiple instructions per clock cycle, in the range of 2–16 instructions per cycle in contemporary designs. A processor that initiates 4 instructions per cycle has best-case CPI of ¼, or a best-case IPC of 4. These processors include multiple functional units to execute instructions in parallel, such as adders and multipliers.

  As an example, suppose a superscalar processor that is 4 instructions wide and 15 stages deep. This processor pipeline can overlap the execution 60 instructions and requires high ILP to deliver significant performance improvements over the simple pipelines.

  We will next review how the ILP challenges we presented in Section 3.1 have shaped the design of superscalar processors, before explaining each technique in detail in subsequent sections.

Control dependences → Branch prediction and speculative execution

Typical programs include a branch instruction every 4 to 7 instructions. The resulting control dependences severely limit the ILP available to exploit.

Consider the loop example from Section 3.1 that increments the elements of an array:

```
Loop:       fld f0,0(x1) //f0=array element
            fadd.d f4,f0,f2 //add scalar in f2
            fsd f4,0(x1) //store result
            addi x1,x1,-8 //decrement pointer 8 bytes
            bne x1,x2,Loop //branch x1≠x2
```

There is little parallelism to exploit within a single iteration of this loop. To overlap independent instructions across iterations, we must quickly resolve the `bne` branch at the end of the loop. As we will see later in this chapter, the branch delay in a 15-stage pipeline is typically 5—10 cycles. Since branches that terminate loops are frequently taken, each branch delay cycle leads to a missed opportunity to execute 4 instructions in pipeline that is 4 instructions wide, diminishing most performance benefits from the multiple-issue and deeper pipeline.

Loop unrolling by the compiler can alleviate this problem, as it reduces the frequency of branches. However, the right amount of unrolling depends on the processor design. While a 2-wide, 7-stage processor may perform well with the loop unrolled 4 times, a 4-wide, 15-stage processor may require the loop to be unrolled up to 16 times to expose enough parallelism. With many programs distributed in binary form, it is difficult to recompile and reoptimize for each processor pipeline. Moreover, loop unrolling does not help with branches for conditional statements (if-then-else), which can be common within loops.

Superscalar processors reduce the impact of control dependences using *dynamic branch prediction* and *speculative execution*. They implement hardware structures that maintain history on recent executions of branches and use them to predict if a branch is taken (*direction prediction*) and what will be the next instruction after a taken branch (*target prediction*). In the simple loop above, a few iterations and simple history structures are enough to conclude with high confidence that the branch is almost always taken, and its target is static throughout the program execution. Large and complex prediction structures are needed to maintain history for programs with thousands of branches and to capture the patterns that lead to predictable branch behavior across nested loops, direct and indirect function calls, and conditional statements, even if these patterns change during program execution. Moreover, the processor pipeline must handle multiple predicted branches in flight due to the deeper pipelines and wide issue.

Branch prediction takes place in parallel with fetching instructions. On every clock cycle, hardware must determine the program counter (PC) to use for instruction access in the next cycle. A first prediction must be made with the current PC as the only input, before knowing if it even points to a branch or the type of this branch instruction. Hence processors often use multilevel schemes that refine predictions as information about the fetched instructions becomes available during instruction decoding.

The instructions fetched after a predicted branched execute speculatively until both the direction and target prediction are validated when the branch instruction executes. If either prediction was wrong, a *recovery* mechanism is used to cancel

any instructions fetched and executed speculatively after the mispredicted branch. After recovery, execution resumes from the correct PC.

Branch prediction must be highly accurate for a superscalar processor to perform well. Let's assume that our 4-wide processor has 10 cycles of branch delay and achieves 95% prediction accuracy for the 5-instruction loop shown above. On each misprediction, the processor will fetch the wrong instructions for 10 cycles, an opportunity cost of 40 instructions for a 4-wide design. In order to get 40 instructions to execute as fast as other dependencies allow in a wide-issue and deep pipeline, the processor must correctly predict 7 branches in the row. The probability for that is $(0.95)^7 = 69.8\%$. The probability that the processor gets 200 instructions to execute without a misprediction is $(0.95)^{39} = 13.5\%$. Ignoring all other performance challenges, even 95% prediction accuracy places an upper bound on sustained performance of any superscalar processor and limits the width and depth of its pipeline.

Name dependences → Register renaming

With branch prediction enabling loop unrolling in hardware, superscalar processors can overlap instructions across multiple iterations. In our loop example, overlapping the loads (`fld`) from multiple iterations is key as these instructions have considerable latency, especially if some miss in the L1 cache. However, all load instructions have a name dependence on register `f0` (WAW). Figure 3.3 shows all the name dependences, WAW and WAR, between two iterations of the loop that prohibit any meaningful overlapping of instructions across iterations.

As we saw in Section 3.2, when the compiler unrolls the loop four times, it uses a different set of integer registers for each iteration, eliminating name dependences. *Register renaming* attempts to do the same in hardware with two major differences. First, hardware cannot assume that any of the other registers defined in the ISA are available to use. While our example code uses only registers `f0`, `f2`, `f4`, and `f6`, all other floating-point and integer registers may store data that are used in preceeding and subsequent parts of the program. Second, we may need

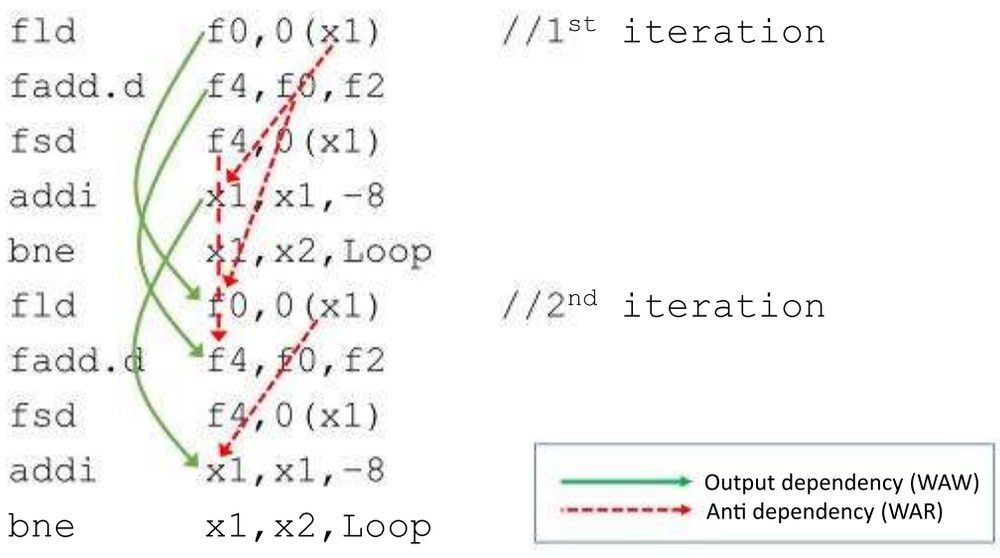


Figure 3.3 All the name dependences between instructions in two iterations of the unrolled loop. Name dependences can be output dependences (WAW) or antidependences (WAR).

more registers than those defined in the ISA. To unroll the loop 10 times for a wide and deep pipeline, we need 40 FP registers (4 per iteration) which is more than the 32 defined in RISC-V.

Hardware register renaming addresses these challenges by implementing hundreds of *physical registers* that are transparent to software and dynamically allocated to eliminate name dependences. Hardware maintains a mapping between the *architectural registers* referred to by instructions and the physical registers implemented by hardware. This mapping is used during instruction issue to determine which physical registers hold the instruction's input operands and is updated to indicate which physical register will store the instruction's output operand. As we discuss in Section 3.4, one challenge is to determine when the value stored in a physical register is no longer needed and the physical register can be reused by a future instruction.

Data dependences → OOO execution with dynamic scheduling

With branch prediction speculating on control dependencies and register renaming eliminating name dependences through registers, superscalar processors implement *dynamic scheduling* to execute fetched instructions in the order of data dependences, also known as the *data-flow order*. The hardware implements two functions. First, it tracks the availability of input operands (physical registers) for all pending instructions. When all its input operands are available, an instruction is *ready for execution*. In our loop example the `fsd` instruction becomes ready when the values of registers `f4` and `x1` are available. Second, the hardware selects a subset of the ready instructions for execution on the available functional units using criteria such as instruction age and instruction criticality.

Dynamic scheduling leads to OOO instruction execution. In our example loop, the load instructions (`fld`) for the first iteration may encounter cache misses and the dependent add (`fadd.d`) and store instructions (`fsd`) will not be ready for execution for tens of cycles. However, the `addi` instruction can execute as it is not data dependent on the load. This opportunity in turn will allow the load instruction in the second iteration to execute. If the second load encounters a cache hit, all FP instructions in the second iteration will likely complete before the corresponding instructions in the first iteration.

Dynamic scheduling is limited by the capacity of the hardware structures used for operand tracking, which is typically in the range of a few tens to a hundred instructions. Unlike static scheduling by the compiler that works well when the execution latencies of instructions are statically known, dynamic scheduling can handle variable execution latencies. This feature is important for scheduling in the presence of load instructions that read data from multilevel caches and may exhibit latencies ranging from 2 cycles to 100 cycles. Hardware will dynamically schedule around loads that miss and execute any subsequent, independent instructions. This optimization is a key performance advantage of modern superscalar processors.

Memory dependences → speculative memory disambiguation

Dynamic scheduling must also respect dependences through memory locations. Consider the following version of our example loop where the destination array is different from the input array and the traversal order of the two arrays differs:

```
Loop: fld f0,0(x1) //f0=array element
      fadd.d f4,f0,f2 //add scalar in f2
      fsd f4,40(x3) //store result in different array
      addi x1,x1,-8 //decrement x1 pointer by 8 bytes
      addi x3, x3, 8 // increment x3 pointer by 8 bytes
      bne x1, x2, Loop //branch x1≠x2
```

The key to high performance is to execute loads from multiple iterations as early as possible, since the remaining floating-point instructions are data dependent on loads that may incur high latency during cache accesses. For hardware to schedule the load of the $(i + 1)^{th}$ iteration ahead of the store of the i^{th} iteration, it must first check that they operate on different memory addresses. This check cannot always be done statically as the values of registers `x1` and `x3` may not be statically known and the compiler cannot determine if the input and output arrays are disjoint or overlap. Moreover, the memory address for the store may not be instantly available. The check can only take place after the execution of all previous instructions that affect the value of register `x3`, and after the store instruction is assigned to an address generation unit that calculates its effective address (`x3+40`). This constraint can delay when the next load can execute, limiting the amount of ILP the hardware can exploit. Even when the effective addresses of loads and stores are available, the hardware may not be able to safely reorder loads and stores without violating the memory consistency constraints in multiprocessor systems (see Chapter 5).

Superscalar processors overcome the uncertainty of memory dependences using *speculative memory disambiguation*. Similar to tracking branch history in order to speculate around control dependencies, the hardware tracks recent history for load and store instructions in order to learn which load instructions are likely to be dependent on older stores that have not yet calculated their address. Loads without likely dependences are speculatively reordered ahead of older stores. Once the addresses for all older stores are available, the hardware can verify if its decision to reorder a load was correct. If not, the hardware must recover from the incorrect speculation by reexecuting the load and any dependent instructions.

Speculation Recovery and Precise Exceptions → Reorder Buffer

Program execution on a superscalar processor must be functionally equivalent to using a simple processor with no instruction overlapping. Such a simple processor executes instructions only after all their dependences are resolved, so it does

not need to predict the outcome of branches or the order of load/store instructions. To maintain functional equivalency, a superscalar processor needs a recovery mechanism that cancels any instructions that were incorrectly executed after a mispredicted branch instruction or a mispredicted dependence between a load and a store instruction. The recovery mechanism should ensure that incorrectly executed instructions have no impact on the program's data flow (no updates to register or memory values) or control flow (no updates to the PC).

This equivalency requirement extends to any exceptions raised during the program execution, such as a page fault on a load instruction or arithmetic overflow on a floating-point instruction. In our simple processor, when an exception occurs and control is transferred to the operating system, all instructions preceding the offending instruction have completed their execution, while none of the instructions following the offending one have modified the processor state (registers or memory locations) or raised any exceptions. This property is known as *precise exceptions*. A superscalar processor uniformly predicts that instructions will not raise exceptions—a prediction that is generally very accurate. However, any instructions that come after an instruction that *could* raise an exception must be treated as speculative until the processor can verify that an exception will not occur.

Advanced superscalar processors have two properties that complicate precise exceptions. First, they reorder instructions. At the time the store instruction in the first iteration of our example loop raises a page fault exception, several load and floating-point instructions from subsequent iterations may have already executed and even raised exceptions of their own. Second, they speculate on both control and memory dependences. If the branch predictor is incorrect, some of the instructions that were incorrectly executed fully or partially before the misprediction is discovered and recovered may raise exceptions. If speculative memory disambiguation is wrong, some load instructions may return incorrect data, leading to incorrect calculations and incorrect exceptions.

Superscalar processors recover from mispeculation using a *reorder buffer (ROB)*. The ROB is a first-in, first-out (FIFO) structure that temporarily buffers instruction results and allows them to have a permanent impact on the program's data flow and control flow only in the original program order. Instructions are assigned entries in the ROB in program order. Instructions update their ROB entry as they complete their execution out of order, marking any potential exceptions detected during execution. On every clock cycle, the hardware checks the oldest instructions in the ROB and waits until their execution has completed before the instruction results are allowed to take permanent effect on registers (older values in physical registers are discarded) and memory locations (stores are allowed to overwrite memory locations) and the ROB entries are released. If the oldest instruction is a branch with an outcome that was mispredicted earlier in the pipeline, all following instructions are canceled by freeing their ROB entries and any other temporary buffers, such as the physical registers these instructions allocated for their results. Canceling all these instructions ensures that incorrect

branch speculation does not impact the program's data flow and control flow. Execution resumes from the correct branch target produced during the execution of the branch instruction. If the oldest instruction is a load that was incorrectly reordered ahead of older stores, we cancel the load itself and all following instructions. Execution resumes from the address of the load instruction. This ordering ensures that the memory dependence mispeculation does not impact the program's data flow and control flow.

The speculation recovery provided by the ROB also supports precise exceptions. If the oldest instruction in the ROB is marked to lead to an exceptional condition, such as a page fault or an arithmetic overflow, the oldest instruction and all instructions behind it are canceled. Execution resumes from the PC associated with the operating system handler that will process this type of exception. The hardware also updates any registers that help the operating system identify the instruction that caused the exception and the type of exception. Since exceptions are reported to the operating system only when an instruction reaches the head of the ROB, neither mispeculation nor instruction reordering can impact the order in which exceptions are reported.

Superscalar Processor Overview

Figure 3.4 summarizes how a modern superscalar processor combines multiple issue and deep pipelining with speculative and dynamic scheduling techniques. Such a processor can have tens to hundreds of instructions pending throughout its pipeline, including multiple predicted branches and multiple load/store instruction that experienced cache misses. The processor pipeline consists of three parts that process multiple instructions per cycle in a pipelined manner.

The processor front-end *fetches* and *decodes* multiple instructions per cycle. It operates in a speculative, in-order manner as it uses branch prediction to determine the PC used in each cycle, but it does not reorder instructions. After each instruction is decoded, its register operands are renamed, an ROB entry is allocated for its results, and the instruction is *issued* to the hardware structures for dynamic scheduling. If there is no free physical register, no free ROB entry, or no free scheduling entry, the front end stops fetching new instructions until resources are available. We will use the term *instruction bundle* to describe the group of instructions that are issued in parallel within a clock cycle. In most superscalar processors the instructions in a bundle are also fetched and decoded in parallel.

The execution engine uses dynamic scheduling to process multiple instructions per cycle speculatively and OOO at the speed allowed by data dependences. An instruction is *dispatched* for execution, when all its input operands are ready, and a proper functional unit is available. When execution completes, the instruction outcomes, including any exceptions, are buffered in the ROB entry. The scheduling hardware is notified as some dependent instructions may now be ready to execute. Note that the instructions within a bundle will likely be dispatched and complete execution on different clock cycles. The hardware will reorder and overlap the execution of instructions from multiple bundles.

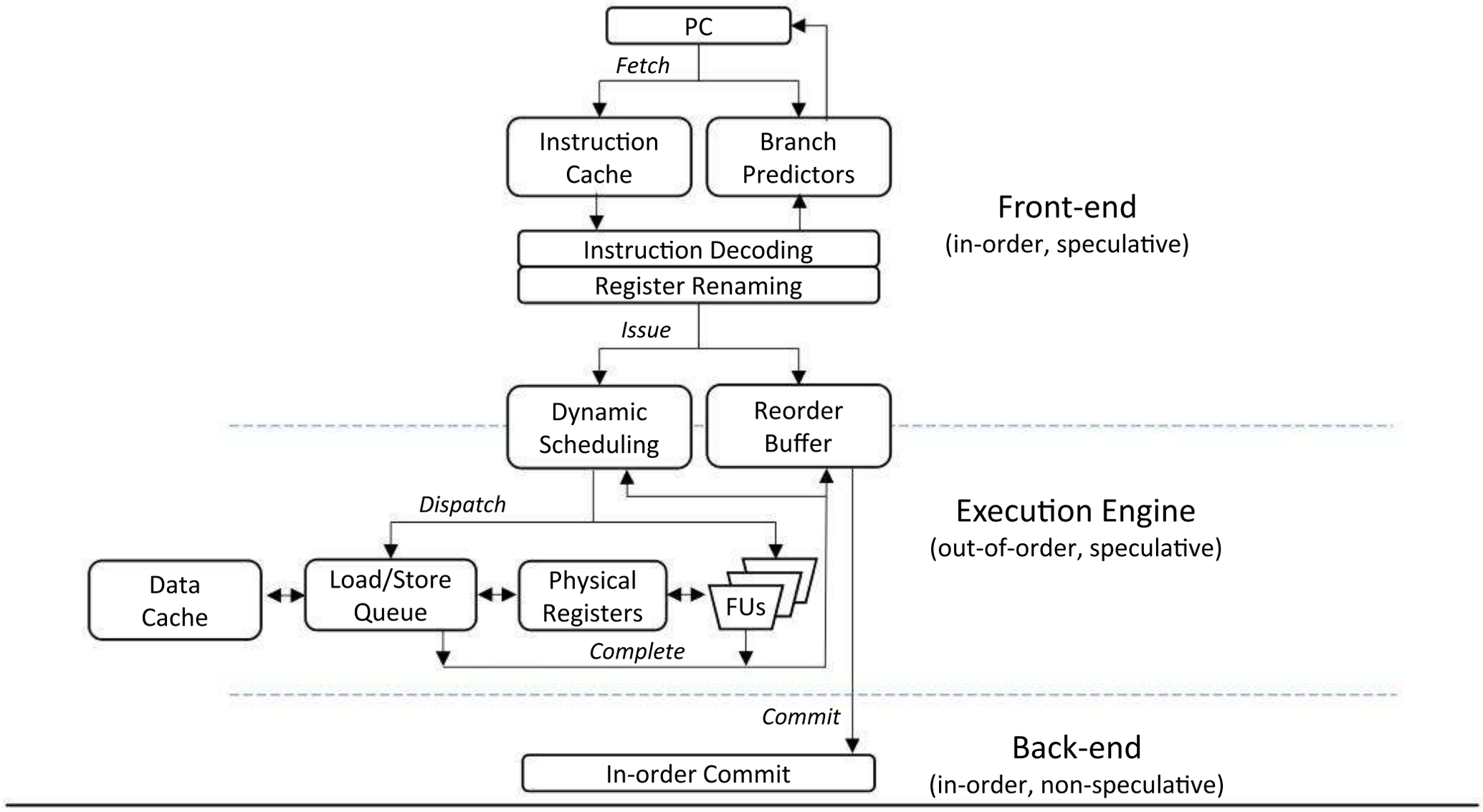


Figure 3.4 A high-level overview of a modern superscalar and out-of-order processor pipeline. The front-end, execution engine, and back-end are processing multiple instructions per cycle. Several of the hardware blocks shown here, such as the caches or some functional units, may occupy multiple pipeline stages.

The processor back-end uses the ROB to update the processor state, registers, and memory locations, in program order and to recover from any mispeculations or exceptions. An instruction *commits* (or *retires*) and its result becomes architecturally visible only after it becomes the oldest instruction in the ROB. If the oldest instruction led to an exception or misprediction, the ROB is used to undo any instructions past the offending one: release any physical registers they allocated, free their ROB entries, and resume execution from the correct PC. Some of these canceled instructions may have completed execution, while others may still be waiting for their operands.

Now that we understand the overall operation of a modern OOO processor, the following sections will examine in depth the implementation of the key hardware structures in its pipeline. As we will discuss further in Section 3.8, a major source of complexity is processing multiple instructions per cycle in every pipeline stage without violating any dependences.

3.4 Reducing Branch Costs with Advanced Branch Prediction

The need to enforce control dependences through frequent branches makes branch prediction essential to exploit ILP. In Appendix C we examine simple

branch predictors that rely either on compile-time information or on the observed dynamic behavior of a single branch in isolation. As the number of instructions in flight has increased with deeper pipelines and more issues per clock cycle, the importance of more accurate branch prediction has grown. In this section, we examine hardware techniques for improving dynamic prediction accuracy, covering both direction prediction (taken or not taken) and target prediction. This section makes extensive use of the simple 2-bit predictor covered in Section C.2, and it is critical that the reader understand the operation of that predictor before proceeding.

Correlating Branch Predictors

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

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

Here is the RISC-V code that we would typically generate for this code fragment, assuming that `aa` and `bb` are assigned to registers `x1` and `x2`:

```
    addi x3,x1,-2
    bnez x3,L1 //branch b1 (aa!=2)
    add x1,x0,x0 //aa=0
L1: addi x3,x2,-2
    bnez x3,L2 //branch b2 (bb!=2)
    add x2,x0,x0 //bb=0
L2: sub x3,x1,x2 //x3=aa-bb
    beqz x3,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 neither branches b1 nor b2 are taken (i.e., if the conditions both evaluate to true and `aa` and `bb` are both assigned 0), then b3 will be taken, because `aa` and `bb` are clearly equal. A predictor that uses the behavior of only 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 (T) or not taken (NT). The branch-prediction buffer can then be indexed using a concatenation of the low-order bits from the branch instruction address (PC) 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. By combining the local and global information by concatenation or a simple hash function, we can index the predictor table with the result and get a prediction as fast as we could for the standard 2-bit predictor, as we will do very shortly.

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

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

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

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

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

Figure 3.5 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.

Perhaps the best-known example of a correlating predictor is McFarling's gshare predictor. In gshare the index is formed by combining the address of the branch and the most recent conditional branch outcomes using an exclusive-OR (XOR), which essentially acts as a hash of the branch address

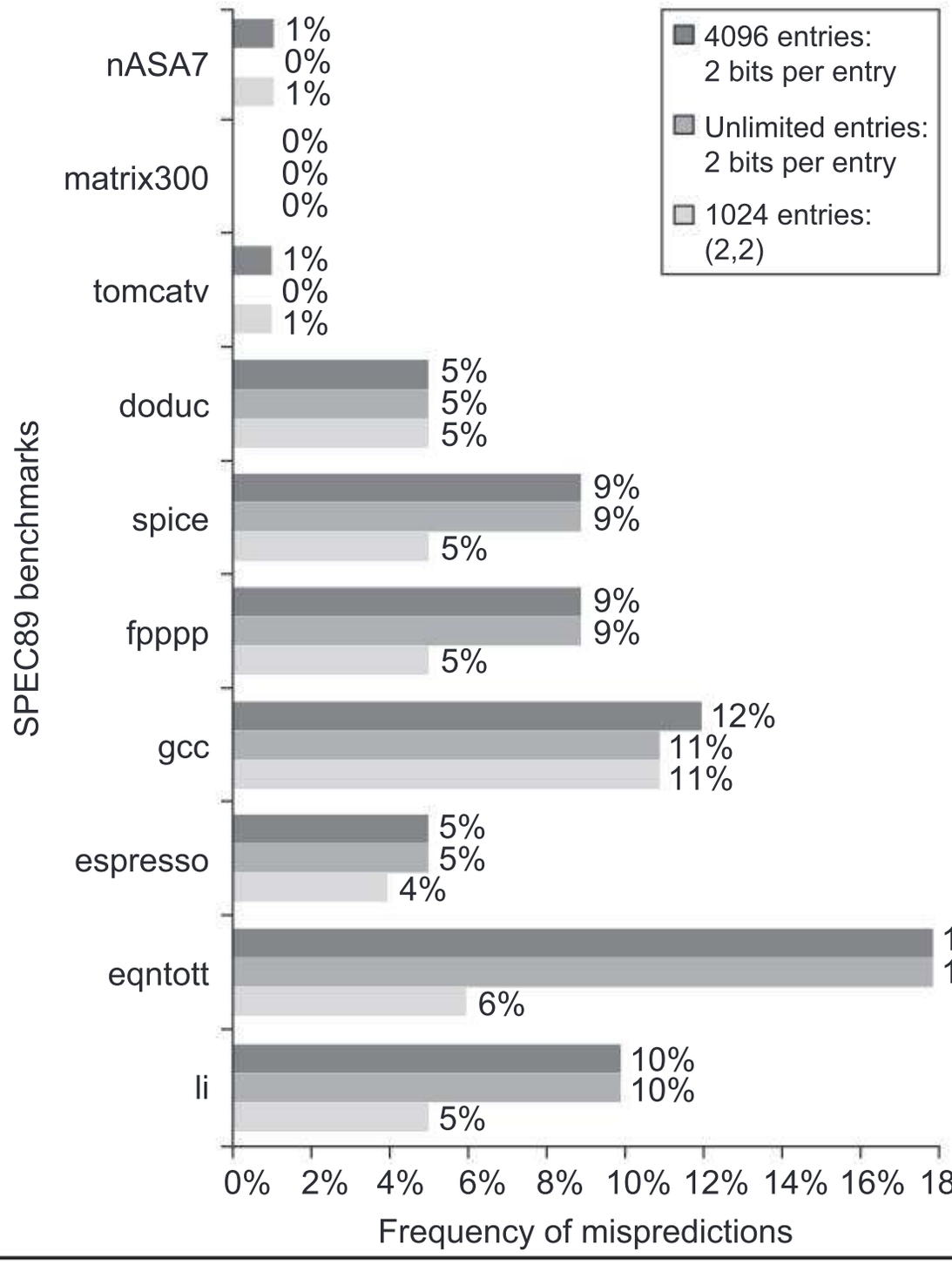


Figure 3.5 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

and the branch history. The hashed result is used to index a prediction array of 2-bit counters, as shown in Figure 3.6. The gshare predictor works remarkably well for a simple predictor and is often used as the baseline for comparison with more sophisticated predictors. Predictors that combine local branch information and global branch history are also called *alloyed predictors or hybrid predictors*.

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. Adding global history could help remedy this situation. *Tournament predictors* take this insight to the next level, by using multiple predictors,

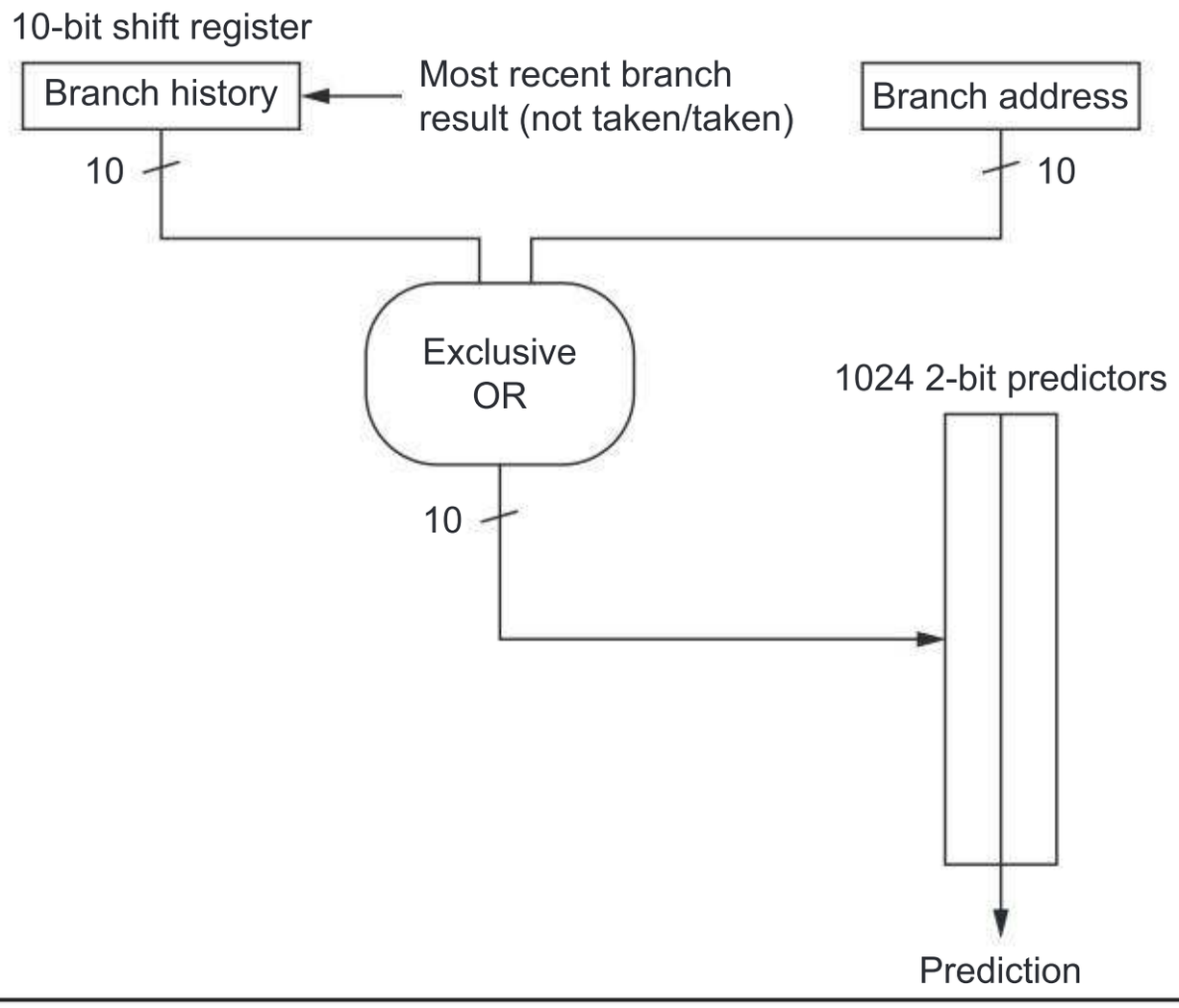


Figure 3.6 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

usually a global predictor and a local predictor, and choosing between them with a selector, as shown in Figure 3.7. A *global predictor* uses the most recent branch history to index the predictor, while a *local predictor* uses the address of the branch as the index. Tournament predictors are another form of hybrid or alloyed predictors.

Tournament predictors can achieve better accuracy at medium sizes (8K–32K bits) and effectively use very large numbers of prediction bits. 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 time-varying 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, several AMD processors have used tournament-style predictors.

Figure 3.8 looks at the performance of three different predictors (a local 2-bit predictor, a correlating predictor, and a tournament predictor) for different numbers of bits using SPEC89 as the benchmark. The local predictor reaches its limit first. The correlating predictor shows a significant improvement, and the tournament predictor generates a slightly better performance. For more recent

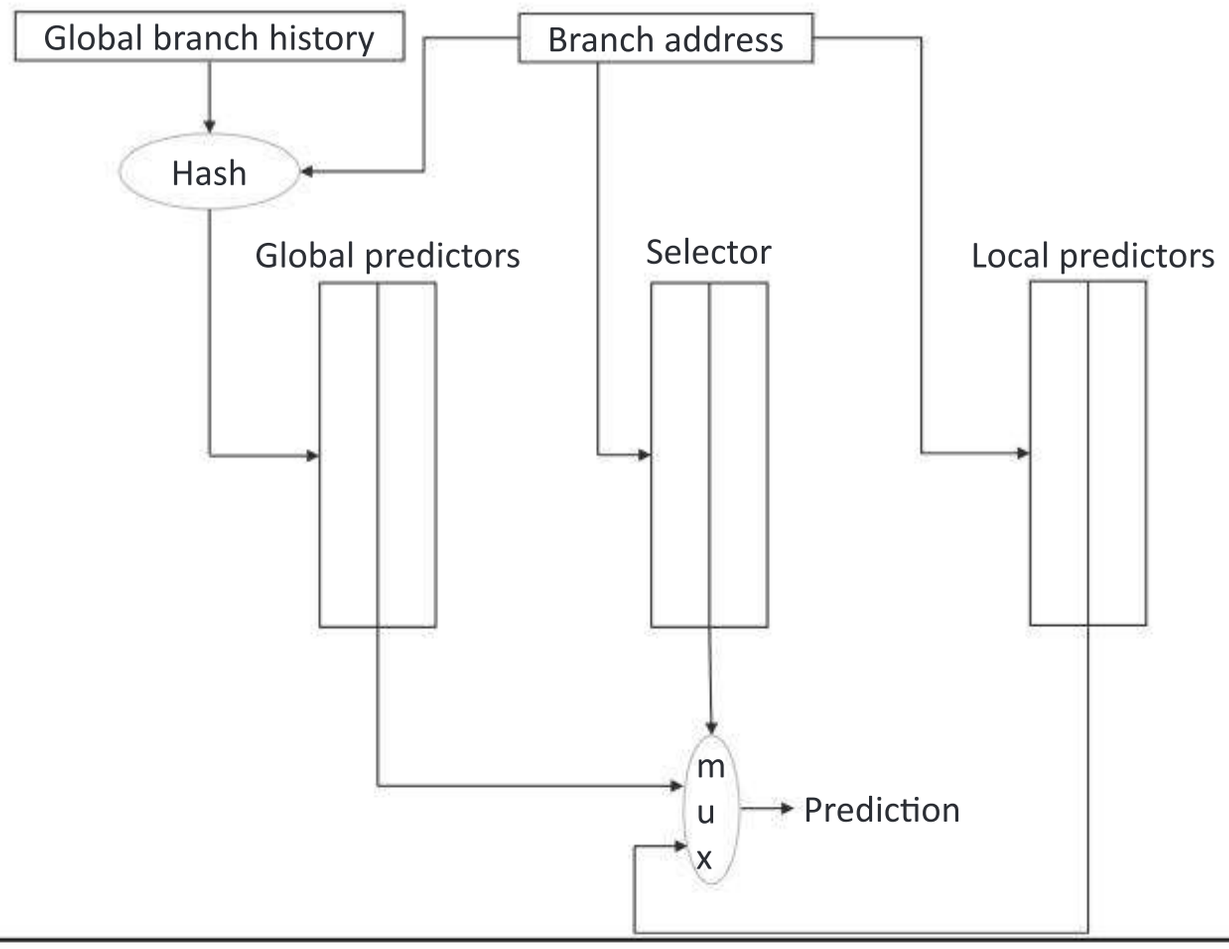


Figure 3.7 A tournament predictor using the branch address to index a set of 2-bit selection counters, which choose between a local and a global predictor. In this case the index to the selector table is the current branch address. The two tables are also 2-bit predictors that are indexed by the global history and branch address, respectively. The selector acts like a 2-bit predictor, changing the preferred predictor for a branch address when two mispredicts occur in a row. The number of bits of the branch address used to index the selector table and the local predictor table is equal to the length of the global branch history used to index the global prediction table. Note that misprediction is a bit tricky because we need to change both the selector table and either the global or local predictor.

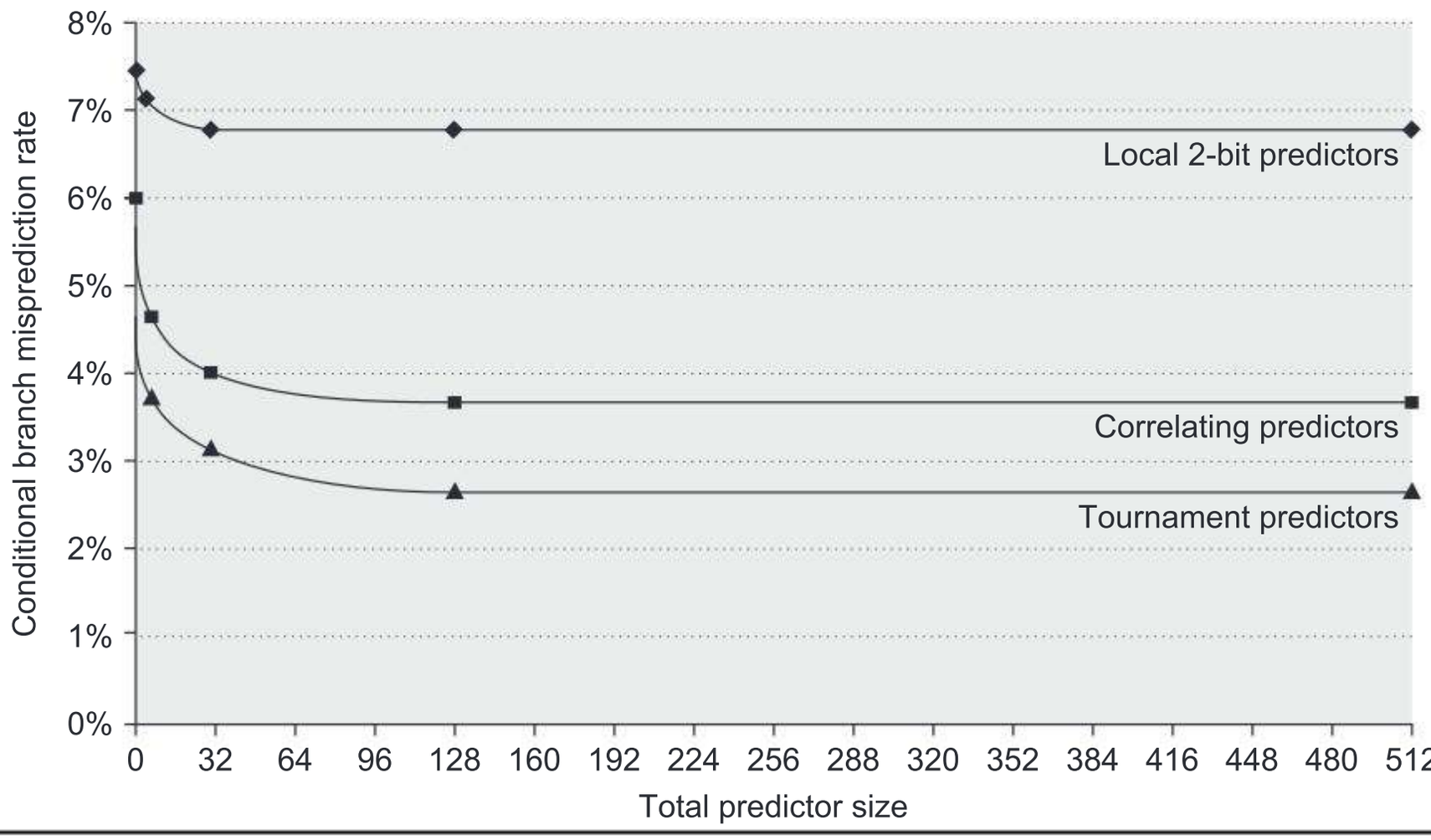


Figure 3.8 The misprediction rate for three different predictors on SPEC89 versus the size of the predictor in kilobits. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

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 evaluated in Figure 3.8 consists of a two-level predictor as well. 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 is taken 10 or more times in a row, the entry in the local history table will be all 1s. If the branch is alternately taken and untaken, the history entry consists of alternating 0s and 1s. This 10-bit history allows patterns of up to 10 branches to be discovered and predicted. The selected entry from the local history table is used to index a table of 1K entries consisting of 3-bit saturating counters, which provide the local prediction. This combination, which uses a total of 29K bits, leads to high accuracy in branch prediction while requiring fewer bits than a single level table with the same prediction accuracy. Two-level local predictors are particularly effective with loops that have small iteration counts. For a loop with 3 iterations, the local history allows us to identify the case of three taken occurrences (TTT) and train a 2-bit or 3-bit saturating counter to predict that the next branch occurrence will not be taken. Hence frequently executing low-iteration loops can be predicted with 100% accuracy.

Tagged Hybrid Predictors

The main shortcoming of predictors that maintain local and global history stems from the difficulty in determining the right history length. If the history is too short, the predictor may not be able to capture some repeating patterns. For example, 4-bit local history cannot capture accurately a loop that iterates 5 times. If the history is too long, training the predictor to accurately capture a pattern takes longer as more entries in the table of saturating counters are used. For example, if we use 3-bits of global history for the code example from eqntott above, we need to train two saturating counters to correctly predict the correlation between B1, B2, and B3, the counters that correspond to global histories NNN and TNN. The more counters we need to accurately capture a pattern, the more likely it is to run into *aliasing*, where hashing collisions lead to multiple branches using the same counter to track history. Aliasing is destructive if the aliased branches have opposite outcomes and their executions are frequently interleaved.

The best-performing branch prediction schemes as of 2022 target these shortcomings by combining multiple predictors that track whether a prediction is likely to be associated with the current branch. One important class of predictors is loosely based on an algorithm for statistical compression called PPM (prediction by partial matching). PPM (see Jiménez and Lin, 2001), like a branch prediction algorithm, attempts to predict future behavior based on history. This class of branch predictors, which we call *tagged hybrid predictors* (see Seznec and Michaud, 2006), employs a series of global predictors indexed with different length histories.

For example, as shown in Figure 3.9, a five-component tagged hybrid predictor has five prediction tables: P(0), P(1),…P(4), where P(i) is accessed using a hash of the PC and the history of the most recent i branches (kept in a shift register, h, just as in gshare). The use of multiple history lengths to index separate predictors is the first critical difference. The second critical difference is the use of tags in tables P(1) through P(4) to reduce aliasing. The tags can be short because 100% matches are not required: a small tag of 4–8 bits appears to gain most of the advantage. A prediction from P(1),…P(4) is used only if the tags match the hash of the branch address and global branch history. Note that in contrast to caches, branch predictor tags are only used to reduce the probability of aliasing, but not for correctness. With a short tag, there still exists a small chance for aliasing; however, this only affects prediction accuracy, not correctness. Each of the predictors in P(0…n) can be a standard 2-bit counter. In practice, a 3-bit counter, which requires three mispredictions to change a prediction, gives slightly better results than a 2-bit counter.

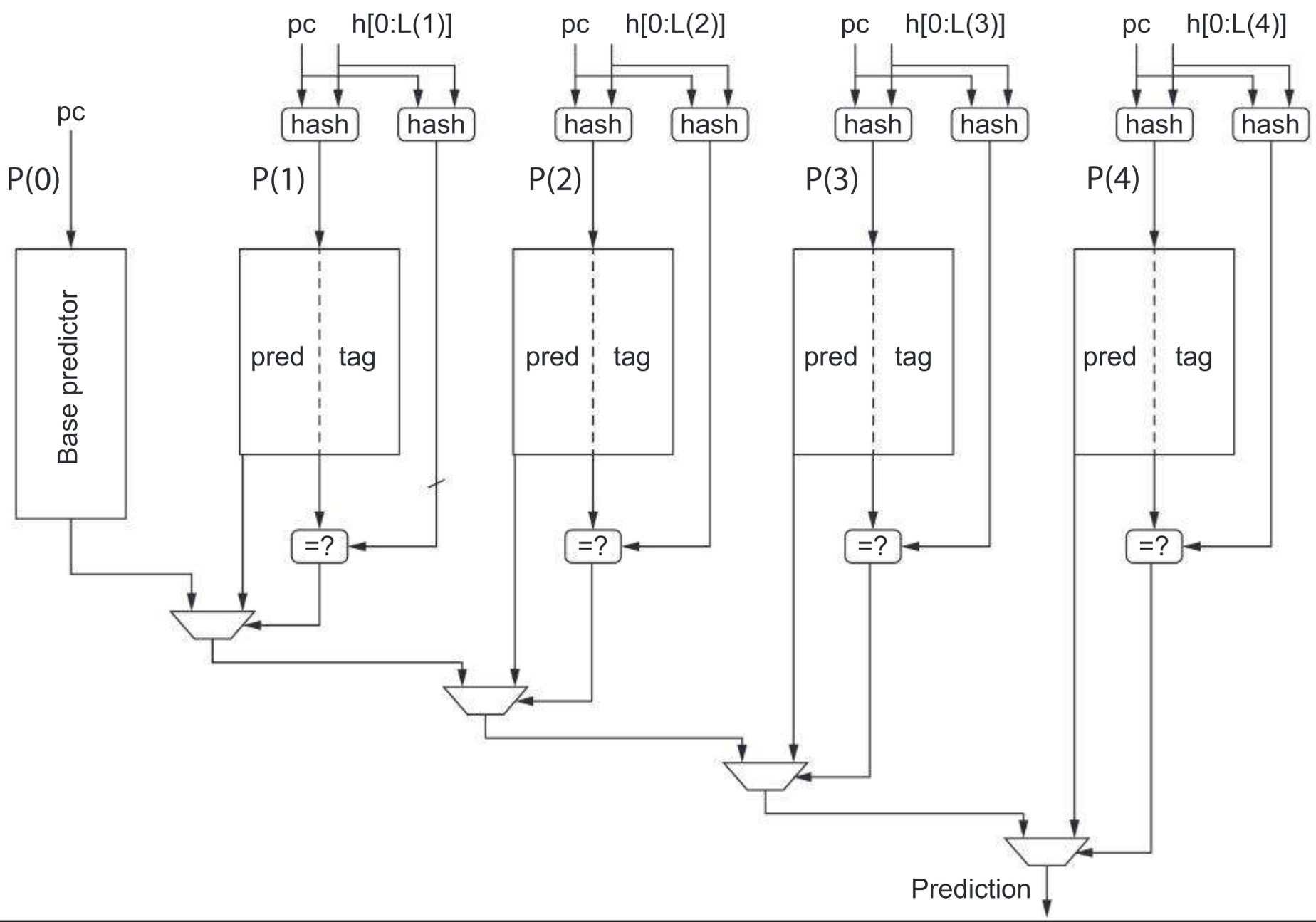


Figure 3.9 A five-component tagged hybrid predictor has five separate prediction tables, indexed by a hash of the branch address and a segment of recent branch history of length 0–4 labeled "h" in this figure. The hash can be as simple as an exclusive-OR, as in gshare. Each predictor is a 2-bit (or possibly 3-bit) predictor. The tags are typically 4–8 bits. The chosen prediction is the one with the longest history where the tags also match.

The prediction for a given branch is the predictor with the longest branch history that also has matching tags. P(0) always matches because it uses no tags and becomes the default prediction if none of P(1) through P(*n*) match. The tagged hybrid version of this predictor also includes a 2-bit use field in each of the history-indexed predictors. The use field indicates whether a prediction was recently used and therefore likely to be more accurate; the use field can be periodically reset in all entries so that old predictions are cleared. Many more details are involved in implementing this style of predictor, especially how to handle mispredictions. The search space for the optimal predictor is also very large because the number of predictors, the exact history used for indexing, and the size of each predictor are all variable.

Tagged hybrid predictors (sometimes called TAGE—TAgged GEometic—predictors) and the earlier PPM-based predictors have been the winners in recent international branch-prediction competitions. Such predictors outperform gshare and the tournament predictors with modest amounts of memory (32–64 KiB), and in addition, this class of predictors seems able to effectively use larger prediction caches to deliver improved prediction accuracy.

Another issue for larger predictors is how to initialize the predictor. It could be initialized randomly, in which case it will take a fair amount of execution time to fill the predictor with useful predictions. Some predictors (including many recent predictors) include a valid bit, indicating whether an entry in the predictor has been set or is in the "unused state." In the latter case, rather than use a random prediction, we could use some method to initialize that prediction entry. For example, some instruction sets contain a bit that indicates whether an associated branch is expected to be taken or not. In the days before dynamic branch prediction, such hint bits *were* the prediction; in recent processors that hint bit can be used to set the initial prediction. We could also set the initial prediction on the basis of the branch direction: forward-going branches are initialized as not taken, while backward-going branches, which are likely to be loop branches, are initialized as taken. For programs with shorter running times and processors with larger predictors, this initial setting can have a measurable impact on prediction performance.

Figure 3.10 shows that a hybrid tagged predictor significantly outperforms gshare, especially for the less predictable programs like SPECint and server applications. In this figure performance is measured as mispredicts per thousand instructions; assuming a branch frequency of 20%–25%, gshare has a mispredict rate (per branch) of 2.7%–3.4% for the multimedia benchmarks, while the tagged hybrid predictor has a misprediction rate of 1.8%–2.2%, or roughly one-third fewer mispredicts. Compared to gshare, tagged hybrid predictors are more complex to implement and are probably slightly slower because of the need to check multiple tags and choose a prediction result. Nonetheless, for deeply pipelined processors with large penalties for branch misprediction, the increased accuracy outweighs those disadvantages. Thus many designers of higher-end processors have opted to include tagged hybrid predictors in their newest implementations.

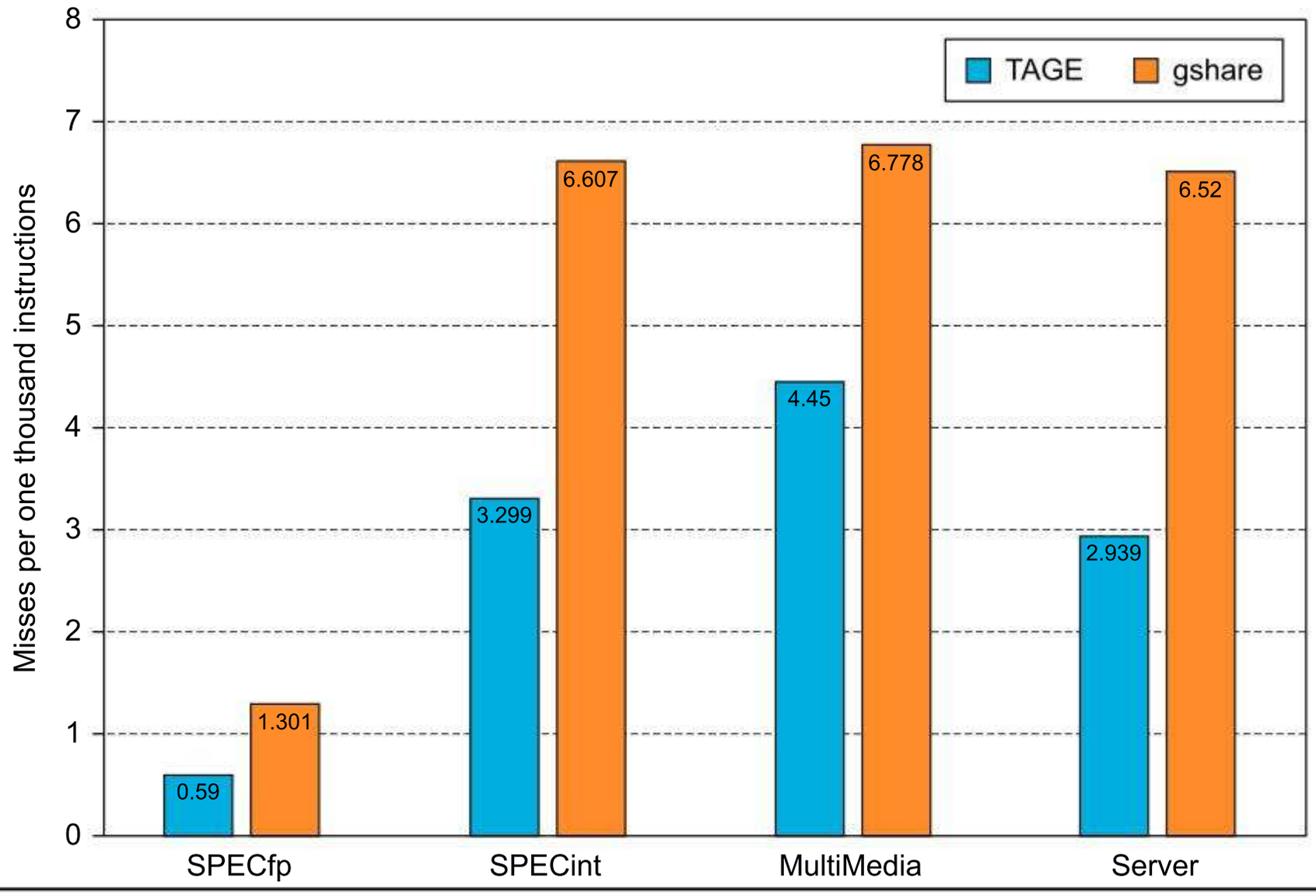


Figure 3.10 A comparison of the misprediction rate (measured as mispredicts per 1000 instructions executed) for tagged hybrid versus gshare. Both predictors use the same total number of bits, although tagged hybrid uses some of that storage for tags, while gshare contains no tags. The benchmarks consist of traces from SPECfp and SPECint, a series of multimedia and server benchmarks. The latter two behave more like SPECint.

Moreover, multilevel and multistage designs provide a way around the speed – size and complexity trade-off for branch predictors.

The Evolution of the Branch Predictor in Intel Processors

Because of the combination of deep pipelining and multiple issues per clock, superscalar processors have many instructions in-flight at once, up to several hundred in recent designs. This makes branch prediction critical, and it has been an area where Intel has been making constant improvements. Perhaps because of the performance-critical nature of the branch predictor, Intel has tended to keep the details of its branch predictors highly secret. Even for older processors such as the Core i7 920 introduced in 2008, they have released only limited amounts of information. In this section, we briefly describe what is known and compare the performance of predictors of the Core i7 920 processor, which uses the Nehalem microarchitecture, with those in the Intel Core i7 6700 processor, released in 2016 and using the Skylake microarchitecture.

The Core i7 920 used 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 2-bit predictor, which is introduced in Appendix C (and used in the preceding tournament predictor); (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 predicts return addresses.

Although even less is known about the predictors in the Core i7 6700 processor, there is good reason to believe that Intel is employing a tagged hybrid predictor. One advantage of such a predictor is that it combines the functions of all three second-level predictors in the earlier i7. The tagged hybrid predictor with different history lengths subsumes the loop exit predictor as well as the local and global history predictor. A separate return address predictor is still employed.

As in other cases, speculation causes some challenges in evaluating the predictor because a mispredicted branch can 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 mispeculation). Figure 3.11 shows these data

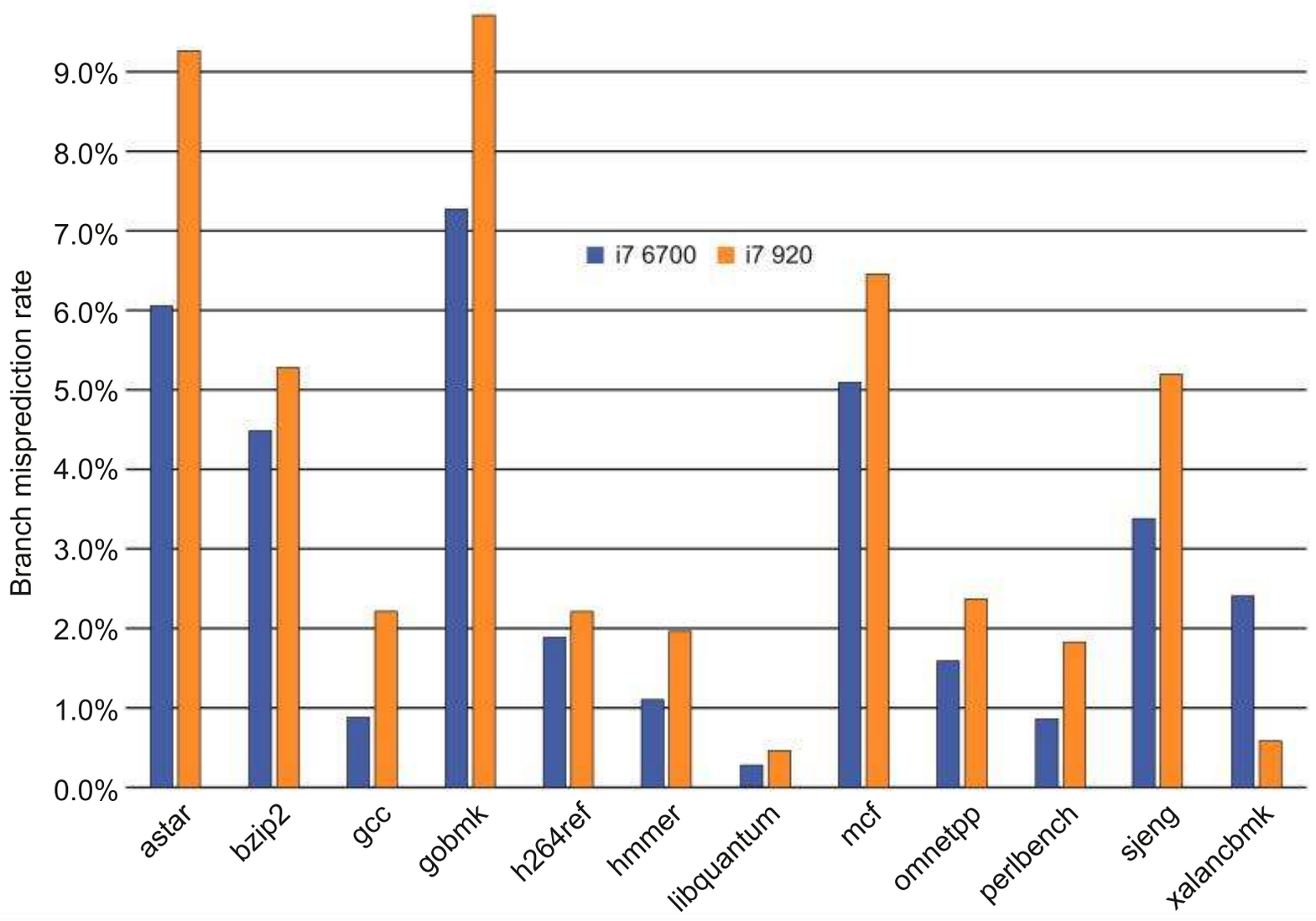


Figure 3.11 The misprediction rate for the integer SPECCPU2006 benchmarks on the Intel Core i7 920 and 6700. The misprediction rate is computed as the ratio of completed branches that are mispredicted versus all completed branches. This could understate the misprediction rate somewhat because if a branch is mispredicted and led to another mispredicted branch (which should not have been executed), it will be counted as only one misprediction. On average, the i7 920 mispredicts branches 1.3 times as often as the i7 6700.

for SPECPUint2006 benchmarks. These benchmarks are considerably larger than SPEC89 or SPEC2000, with the result being that the misprediction rates are higher than those in Figure 3.8 even with a more powerful combination of predictors. Because branch misprediction leads to ineffective speculation, it contributes to the wasted work, as we will see later in this chapter.

Branch-Target Buffers

The branch predictors we have seen so far allow us to speculate if the PC currently used to fetch instructions points to a taken or not taken branch. If the as-yet-undecoded instruction is a branch predicted to be taken, we also need to predict the branch target address to redirect fetch and have a branch penalty of zero. This prediction is made using a *branch-target buffer (BTB)* or *branch-target cache*, a cache that stores the predicted address for the next instruction after a branch. Figure 3.12 shows a BTB. Each entry includes a predicted target, as well as a tag with the PC of the branch that the entry refers to. BTB entries are tagged so that the processor avoids using the target address for the wrong branch in the case

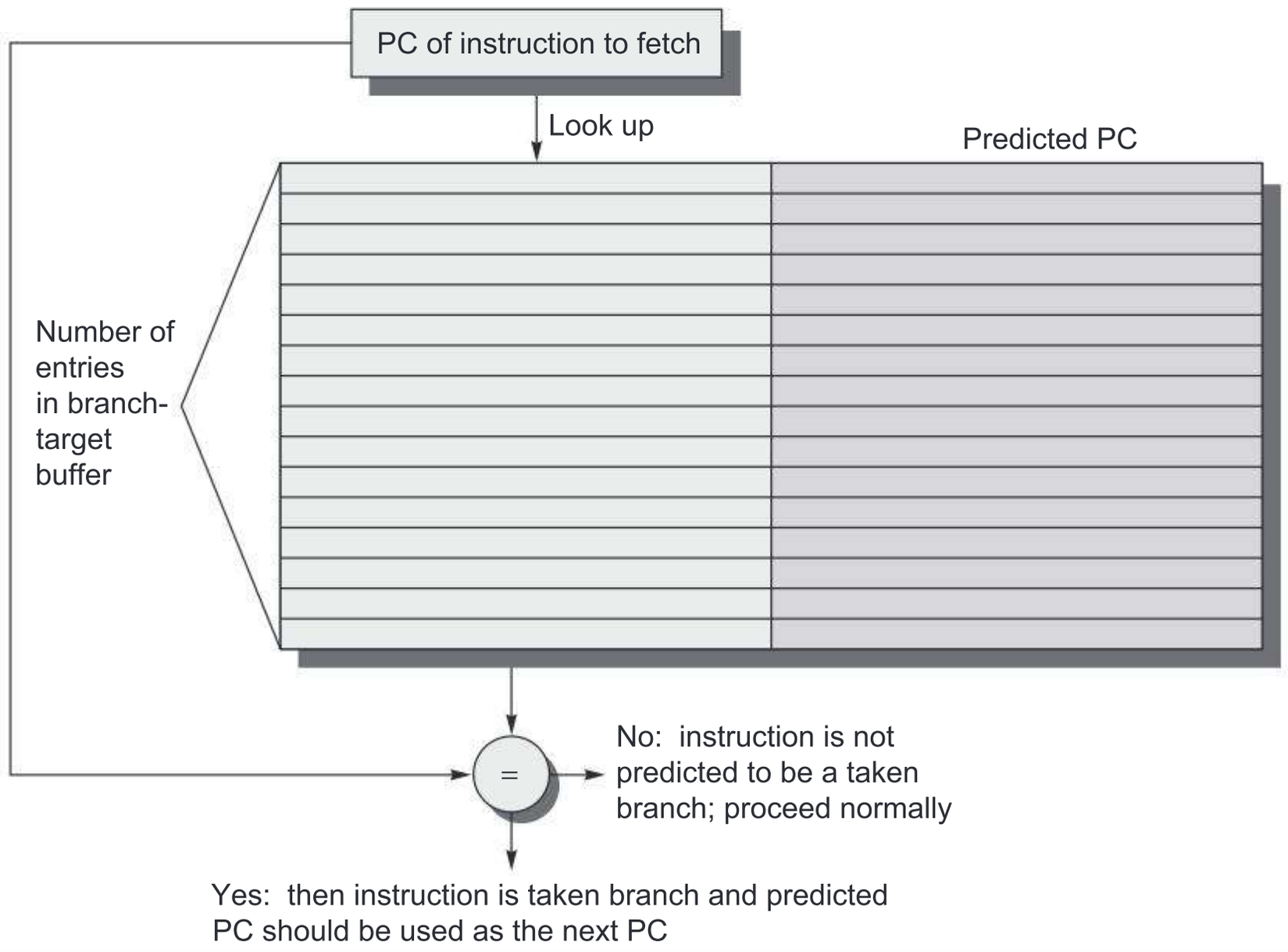


Figure 3.12 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and 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.

of BTB conflicts, which will lead to worse performance. BTB entries include additional information such as a valid bit, the type of the branch, and several bits for implementing a replacement policy.

The BTB is accessed in parallel with the branch predictors. If the branch is predicted to be nontaken, then the BTB lookup result is ignored. If the branch is predicted taken and the BTB entry tag matches the PC of the fetched instruction (BTB hit), the predicted PC is used in the following cycles. If the branch is predicted taken but the BTB returns a miss, then the processor must stop fetching until the current instruction is fetched and decoded in order to calculate its target address. The targets of frequently executing taken branches are inserted into the BTB, replacing the targets for branches that are not recently used or are now frequently not taken. We need to store only the predicted-taken branches in the

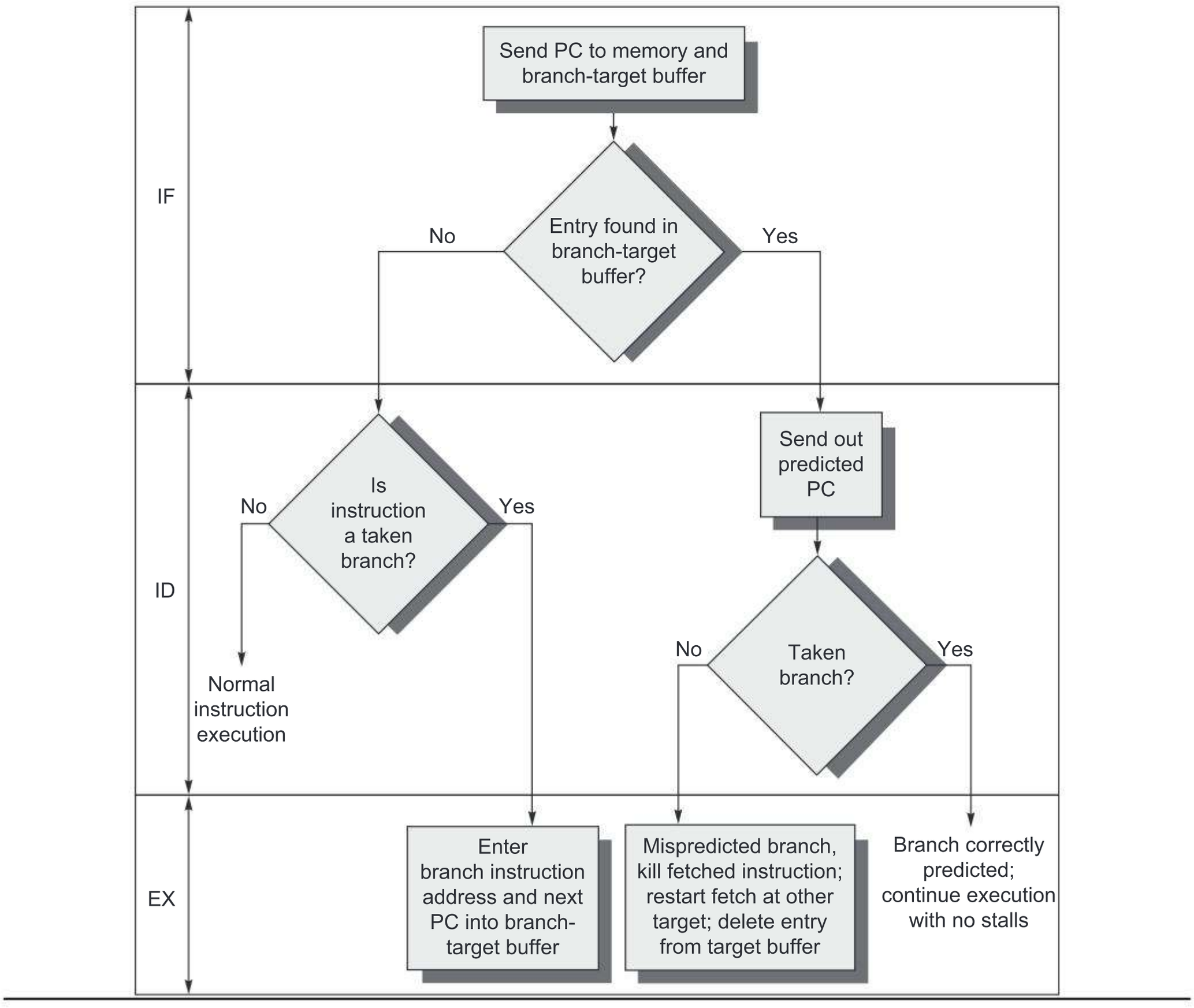


Figure 3.13 The steps involved in handling an instruction with a branch-target buffer.

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

BTB because an untaken branch should simply fetch the next sequential instruction, as if it were not a branch.

The BTB hit rate for taken branches is critical to performance and can be improved using techniques similar to those for improving caches in Chapter 2. Conflict misses can be reduced using associative BTBs. The trade-off between BTB hit time and BTB capacity misses can be addressed using multilevel BTB structures, which are particularly useful for processor pipelines that require 5 to 10 cycles before instruction fetch and effective target calculation.

Figure 3.13 shows the steps when using a BTB for a simple five-stage pipeline. As we can see in this figure, there will be no branch delay for a taken branch if the direction prediction is correct and a branch-target entry is found in the buffer. Otherwise, there will be a penalty of at least two clock cycles.

To evaluate how well a BTB works, we first must determine the penalties in all possible cases. Figure 3.14 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 in Figure 3.14. 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.

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

$$\text{Probability (branch not in buffer but actually taken)} = 10\%$$
$$\text{Branch penalty} = (0.09 + 0.10) \times 2$$
$$\text{Branch penalty} = 0.38$$

The improvement from dynamic branch prediction will grow as the pipeline length, and thus the branch delay grows; in addition, better predictors will yield a greater 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 BTB 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 BTB access to take longer than the time between successive instruction fetches, possibly allowing a larger or multilevel BTB. 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. As we will see, the Cortex A-53 uses a single-entry branch target cache that stores the predicted target instructions.

Consider a BTB that stores 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 BTB signals a hit and indicates that the branch is unconditional, the pipeline can simply substitute the instruction from the BTB 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.

Specialized Branch Predictors: Predicting Procedure Returns, Indirect Jumps, and Loop Branches

As we try to increase the opportunity for parallelism by speculation, we face the challenge of predicting indirect jumps, that is, jumps whose destination address varies at runtime. High-level language programs will generate such jumps for indirect procedure calls (such as C++ virtual function calls), select or case statements, and FORTRAN-computed gotos, although many indirect jumps simply 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.

Although procedure returns can be predicted with a BTB, 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, most processors use a small buffer of return addresses operating as a stack. This *return address stack (RAS)* 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 3.15 shows the performance of such a return buffer with 0–16 elements for a number of the SPEC CPU95 benchmarks. All recent high-performance superscalar processors implement an RAS.

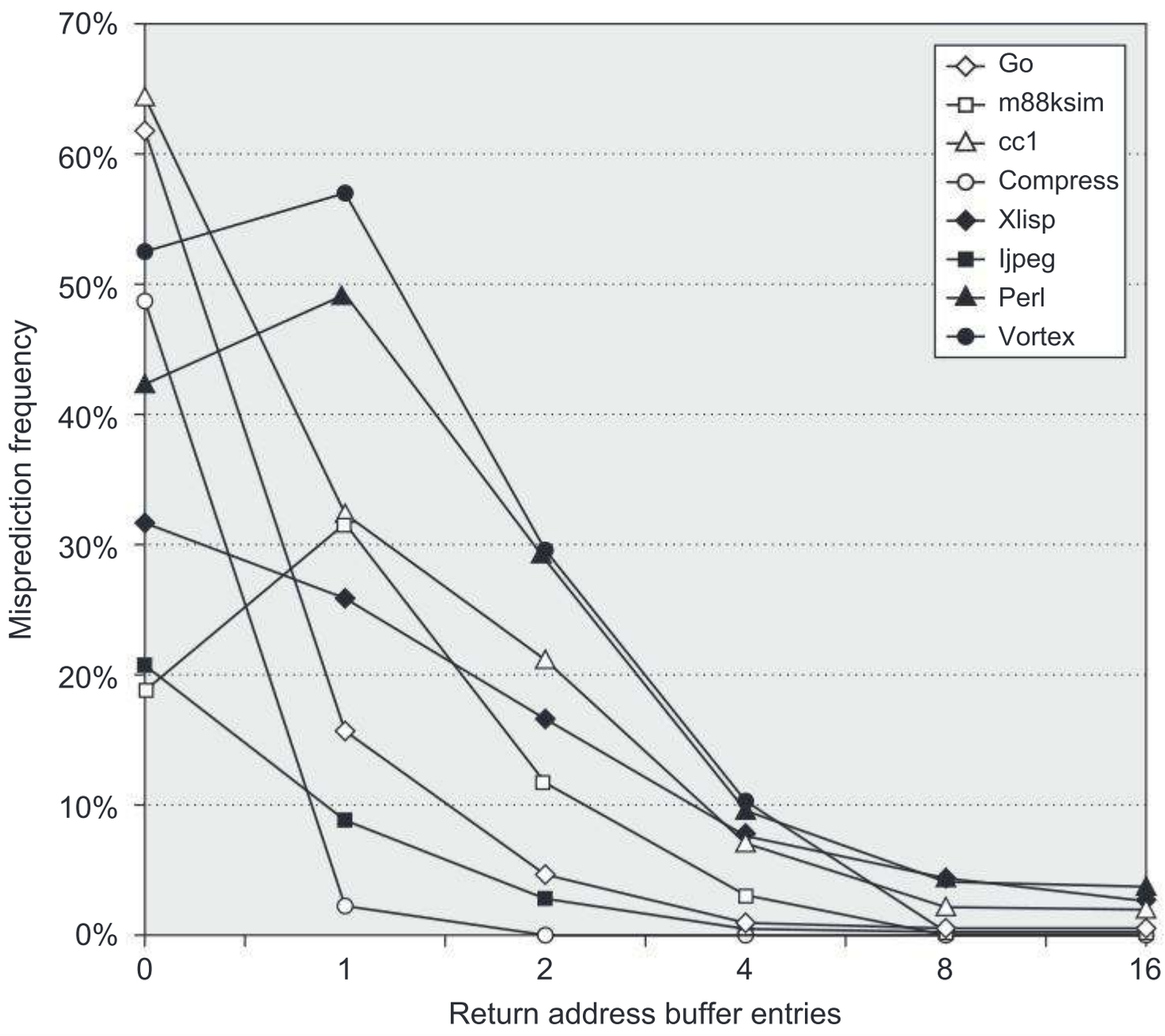


Figure 3.15 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Because call depths are typically not 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.

Indirect jumps also occur for various function calls and control transfers. Predicting the targets of such branches is not as simple as in a procedure return. Some processors have opted to add specialized predictors for all indirect jumps, whereas others rely on a branch target buffer.

Although a simple predictor like gshare does a good job of predicting many conditional branches, it is not tailored to predicting loop branches, especially for long running loops. As we observed earlier, the Intel Core i7 920 (Skylake microarchitecture) used a specialized loop branch predictor. With the emergence of tagged hybrid predictors, which are as good at predicting loop branches, some recent designers have opted to put the resources into larger tagged hybrid predictors rather than a separate loop branch predictor.

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, often referred to as a decoupled front-end. Essentially, this amounts to recognizing that characterizing instruction fetch as a simple single pipe stage is no longer valid given the complexities of multiple issue. 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. Branch predictors and BTBs handle multiple branches per fetch bundle.
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.

3.5 Overcoming Name Dependences with Register Renaming

Register renaming allows superscalar processors to use a large number of physical registers to eliminate the risk of WAR and WAW hazards from name dependences. We will now examine in detail how renaming works using one possible implementation, before discussing the trade-offs in alternative implementations. For simplicity, we discuss the hardware needed for integer register renaming. Similar structures are used to rename floating-point registers.

Figure 3.16 shows the hardware structures necessary for register renaming using a *unified register file (URF)*. This register file implements a large set of physical registers (p0 – p127 in Figure 3.16) used to hold both the values for the 32 architecturally visible registers, as well as temporary values for instructions that have not yet committed. There are two hardware structures that manage the use of these physical registers. The *renaming map* is a table that identifies the physical register number that currently corresponds to the architectural register specified by an instruction as input operands ($x_i \rightarrow p_i$). The renaming map is indexed and updated as instructions are issued in the front-end of the processor pipeline. Hence the mappings stored in renaming map are speculative as some of these instructions may be canceled later due to a mispeculation or exception. The *architectural map* is a table that identifies the physical register number that corresponds to each

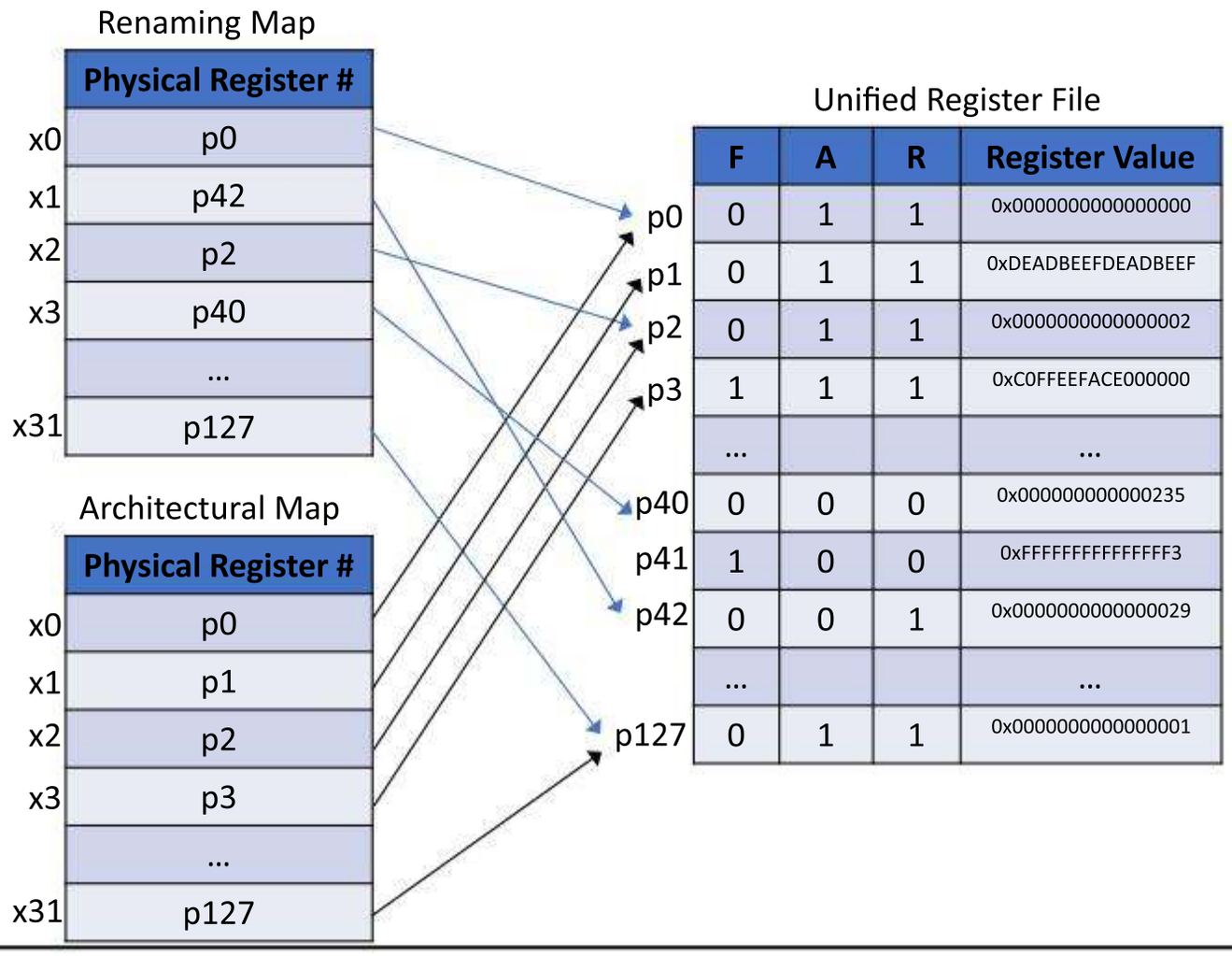


Figure 3.16 The hardware structures for register renaming with a unified register file (URF). The three metadata bits in each URF entry indicate if a physical register is free (F), architectural (A), or ready (R). As shown, physical registers p0—p3 are in architectural state. Physical registers p40 and p42 are in used states, not ready and ready, respectively, as their values are produced by pending instructions. Physical register p41 is free.

architectural register based on the instructions that have committed. The architectural map is updated in the pipeline back-end and is used for exception and misprediction recovery.

Figure 3.17 describes the four possible states of a physical register: free, used and not ready, used and ready, and architectural. The free state indicates that this physical register is not in use and can be allocated by a new instruction. The two used states indicate that the register has been allocated by a pending instruction. The not ready state signifies that the instruction has not yet completed its execution and the value is not ready for use by subsequent dependent instructions. The ready state indicates that register value is ready for us as the producing instruction has completed its execution but has not committed yet. The architectural state indicates that the physical register stores the result of an instruction that completed its execution and has been committed. An entry in the architectural map points to this physical register. Architectural registers are always ready. At reset, the first 32 physical registers are in the architectural state, while the rest are free. In Figure 3.16 the register state for each physical register is encoded using three metadata bits (Free, Architectural, Ready) with the following encoding: Free (1, 0, 0), Used and Not Ready (0, 0, 0), Used and Ready (0, 0, 1), Architectural (0, 1, 1).

The bulk of renaming work is performed in the issue stage after instruction decoding. Nevertheless, physical registers transition between states as instructions go through execution stages.

- **Issue (in order)**: When an instruction is issued, it looks up its input operands in the renaming map. This table provides the addresses of the physical registers that currently store the latest values for the architectural registers read by this instruction. The physical register numbers and their state are passed forward to the hardware that implements dynamic scheduling (see Section 3.5). In parallel, the instruction allocates a new physical register (p_j) for the new value of its output register x_i. The renaming map entry for x_i is updated to $x_i \rightarrow p_j$. The state of the newly allocated physical register is set to used but not ready. Note that some instructions, such as stores and branches, do not produce a new register value and so they do not allocate a new physical register.
- **Dispatch (out of order)**: When all physical registers that store the instruction's input operands are in the architectural state or the used and ready state, the instruction is ready for execution and can be dispatched. We describe the scheduling hardware that tracks input operand state in Section 3.5. The state of physical register p_j for the instruction output remains unchanged (used and not ready).
- **Completion (out of order):** When a functional unit completes the execution of the instruction, it stores the new value into the allocated physical register p_j and changes the physical register state to used and ready. Dependent instructions can now use this value.

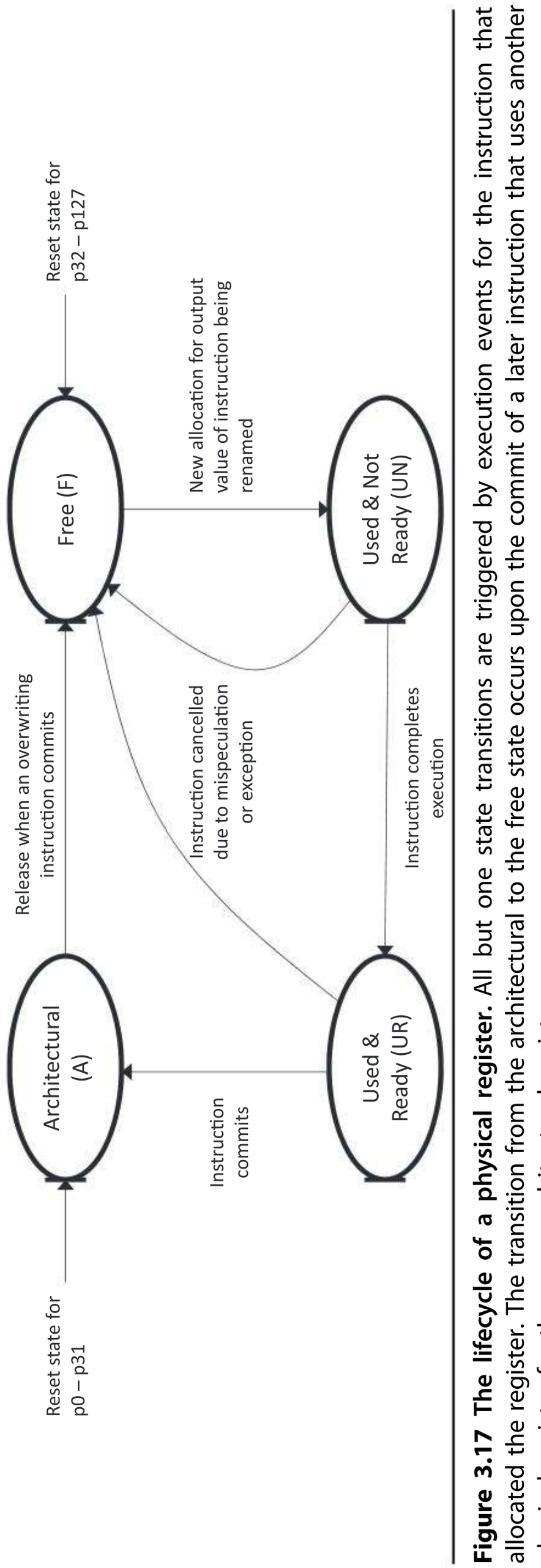


Figure 3.17 The lifecycle of a physical register. All but one state transitions are triggered by execution events for the instruction that allocated the register. The transition from the architectural to the free state occurs upon the commit of a later instruction that uses another physical register for the same architectural register.

- **Commit (in order):** When an instruction reaches the head of the ROB and has not generated exceptions or mispredictions, it commits its results to the architectural state. The state of physical register p_j becomes architectural. We also update the architectural map entry for register x_i. If the entry previously associates $x_i \rightarrow p_k$, we update the entry to indicate $x_i \rightarrow p_j$ and set the state of physical register p_k to free. It is no longer possible that the value stored in p_k is needed as the instruction that overwrites it has just committed.
- **Recovery (in order):** If the oldest instruction in the ROB is associated with an exception or misprediction, we must cancel all pending instructions and recover precise architectural state as if these instructions were never partially processed. For the renaming state, this involves two fast steps. First, all physical registers in the two used states are marked as free. Second, the state of the renaming map is discarded and initialized to the state of the architectural map with a bulk copy operation.

Renaming logic is complex because superscalar processors rename all instructions within a bundle in a single clock cycle. The first implication is that mapping tables must be multiported structures. If a processor dispatches N instructions per cycle, the renaming map must support 2N read ports to lookup the current mapping of all input operands and N write ports to update the mapping for all output operands. If a processor commits M instructions per cycle, the architectural map needs M write ports for the corresponding updates. The three metadata bits for physical registers must also support multiple accesses per cycle. Since they are implemented as bitmasks, the cost of multiple ports for parallel updates is low.

The second implication of renaming multiple instructions per cycle is that the renaming logic must detect and respect any data dependences within the instruction bundle. If a dependence does not exist within the bundle, the renaming map is used to determine the physical register that holds, or will hold, the result on which instruction depends. If an instruction is data dependent on an earlier instruction in the same bundle, then the physical register allocated by the earlier instruction should be used as an input operand. Let's review an example of this issue using the following loop that increments all elements of an integer array:

```
Loop: ld x2,0(x1) //x2=array element
      addi x2,x2,1 //increment x2
      sd x2,0(x1) //store result
      addi x1,x1,8 //increment pointer
      bne x1,x3,Loop //branch if not last
```

Figure 3.18 shows how a processor renames the instructions for two iterations of this loop at the rate of five instructions per cycle. Each instruction bundle includes five instructions. In the first clock cycle, three physical registers are allocated for instruction results: `p51` to rename `x2` for `ld`, `p52` to rename `x2` for the first `addi`, and `p53` to rename `x1` for the second `addi`. At the end of the

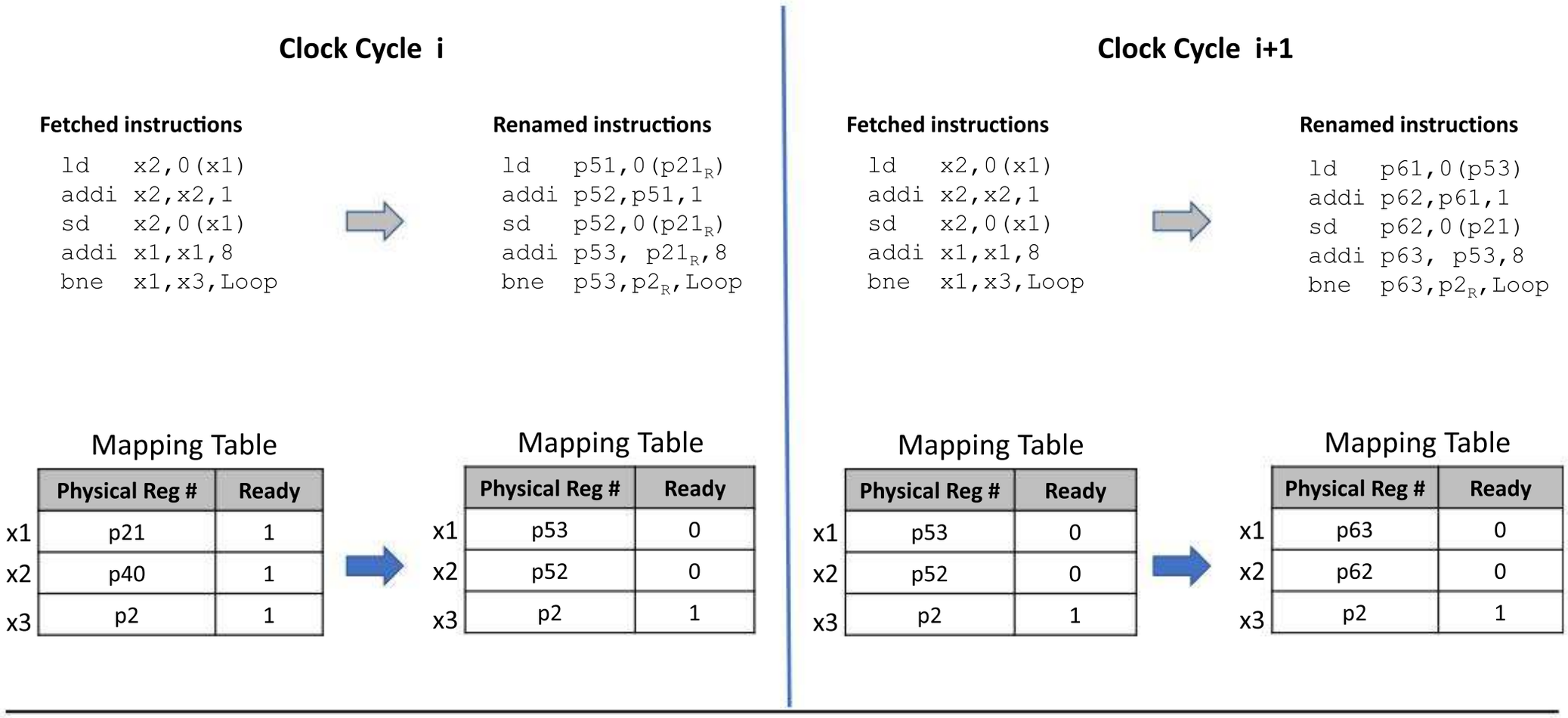


Figure 3.18 Register renaming for two iterations of a loop at the rate of five instructions per cycle. For simplicity, we show only three relevant entries from the mapping table, as well as the ready bits associated with the physical registers each table entry points to. Renaming happens in parallel for 5 instructions in each bundle but must observe data dependences within each bundle. For instance, the `bne` instruction is data dependent on the preceding `addi` instruction for register `x1`. The output of the renaming stage is used for dynamic scheduling. To assist with scheduling, physical registers are marked as ready (subscript R) if it is the case.

cycle, the mapping table reflects the mapping for `x2` generated for the youngest instruction of the two (`p52`). There are also three data dependences within the bundle. The first one involves register x2 produced by `ld` and consumed by the subsequent `addi`. The `addi` ignores the pointer `p40` returned for `x2` by the renaming table. Instead, it uses `p51` which is allocated concurrently by the `ld`. The renaming logic handles similarly the true dependences between `sd` and `addi` for `x2`, and between `bne` and `addi` for `x1`. In the second clock cycle, the renaming logic handles similarly a bundle of instructions from the second iteration of the loop, using new physical registers for the new values produced by these instructions (`p61`, `p62`, and `p63`).

An additional source of complexity is tracking when it is safe to free (deallocate) a physical register. A physical register p_k used for architectural register x_i can be freed after a subsequent, overwriting instruction allocates a new physical register for x_i and there are no instructions waiting to be scheduled that have not yet read p_k. Moreover, we need to be sure no instruction will raise exception before the overwriting instruction can commit. Hence most processors opt for freeing physical registers later, when the overwriting instruction commits, trading off simplicity for longer occupancy for physical registers.

Alternative Implementations for Register Renaming

The are several alternative implementations for register renaming.

No architectural map: It is possible to implement renaming with a URF using only the renaming map. Without an architectural map, precise register state can be recovered in the following way when mispredictions or exceptions occur. At dispatch, each instruction reads and records in the ROB the old mapping for its destination register (p_k) before it overwrites it with the number of the newly allocated physical register (p_j). If the instruction commits, this old mapping is dropped. If there is an exception or misprediction, we traverse the ROB backward, from younger to older instructions, and restore to the renaming map the old mapping included in each ROB entry. The disadvantage of this approach is that the recovery latency is proportional to the ROB size.

Separate physical and architectural registers: Instead of a URF with 128 physical locations, we can implement two register files: one for architectural state (*ARF*) with 32 entries (x0 – x31) and one for temporary renaming registers (*RRF*). Instructions allocate RRF registers upon dispatch and update them at completion. At commit time, the new output value is copied from the physical register in the RRF to the architectural register in the ARF and the renaming table is updated to indicate this. Recovery from exceptions or mispredictions is fast and involves marking as free all RRF registers and reinitializing the renaming map entries to point to the ARF. In fact, one can implement the ARF and the renaming map as a single structure with 32 entries. Each ARF entry includes a pointer to the RRF that indicates the physical register allocated by the latest pending instruction to overwrite this architectural register. No separate architectural map is needed.

Renaming registers in the ROB: If we separate architectural and renaming registers as explained above, we can integrate the renaming registers into the ROB. Each ROB entry provides space for one register value, where the output of the instruction is buffered until the instruction commits. Upon commit, the value is copied from the ROB to the ARF. If there is an exception or misprediction, the temporary buffer is freed along with the ROB entry. The disadvantage of this approach is that it adds further complexity to the already multiported ROB.

Renaming table in the ROB: Some designs have gone one step further by integrating the mapping table in the ROB as well. Upon dispatch, a new instruction must search the ROB for the youngest instruction that may have written to each architectural register it wants to read. If a matching ROB entry is found, this entry provides the input value to the new instruction, when it becomes ready. Otherwise, values are sources from the architectural register file. This renaming approach eliminates the two-step process for reading an input operand, first access the renaming map and then access a register file. However, it requires that the ROB do an associative search for multiple instructions and multiple input operands on each clock cycle. The search process must also prioritize younger instructions, in other words, those furthest away from the head of the ROB.

3.6 Overcoming Data Hazards with Dynamic Scheduling

Dynamic scheduling allows superscalar processors to exploit ILP by executing instructions as soon as their data dependences are resolved, regardless of their program order. It is also essential in the presence of cache misses or other high-latency operations that would otherwise introduce long pipeline stalls. Previous editions of this textbook introduced first the Tomasulo algorithm for dynamic execution, developed in 1967 and implemented in the floating-point unit of the IBM 360/91 mainframe computer. Instead, we will directly describe modern implementations of dynamic scheduling that build upon branch prediction, speculative execution, and register renaming.

Dynamic Scheduling: The idea

Dynamic scheduling involves two steps. First, hardware must track the status of the data dependencies for each pending instruction and determine when it is ready to execute (*instruction wakeup)*. Second, hardware must select some subset of the ready instructions and initiate their execution (*instruction dispatch*). Even though instructions are dispatched and executed out of order, the ROB ensures that it is possible to recover from incorrect speculation and that precise state is recovered if an instruction generates an exception.

Instruction wakeup and dispatch can be implemented using a centralized hardware scheduler (also known as unified scheduler) or a distributed set of smaller scheduling structures. Recent Intel processors such as Skylake and Golden Cove have used a centralized approach for integer and floating-point instructions, while recent AMD processors such as Milan and Genoa have used a distributed approach. We will first describe a centralized scheduler and then discuss the trade-offs with a distributed design. In research literature the centralized scheduler is sometimes referred to as the *instruction window*, *instruction buffer*, or *instruction queue*, even though it does not operate in a first-in, first-out manner.

Figure 3.19 presents the fields in each entry of the centralized scheduler (top) and its overall structure (bottom). Each entry has a busy field to indicate if it is occupied by an unscheduled instruction. The following three fields are used when the instruction starts executing (opcode, destination physical register, ROB entry to update). Next, the entry tracks the state of the two input registers j and k: if they are needed to execute the instruction (V_j and V_k), the indices of the physical registers that will provide their values (Reg_j and Reg_k), and if the values are ready (R_j and R_k). The last field of the entry indicates if the instruction is ready for execution and is set when all valid input registers become ready. An input physical register is ready if it was ready at issue time or a functional unit notifies the scheduler that it completed the execution of the instruction that writes this physical register. Functional units use broadcast buses to notify the instruction scheduler on each clock cycle about the physical registers written by completed

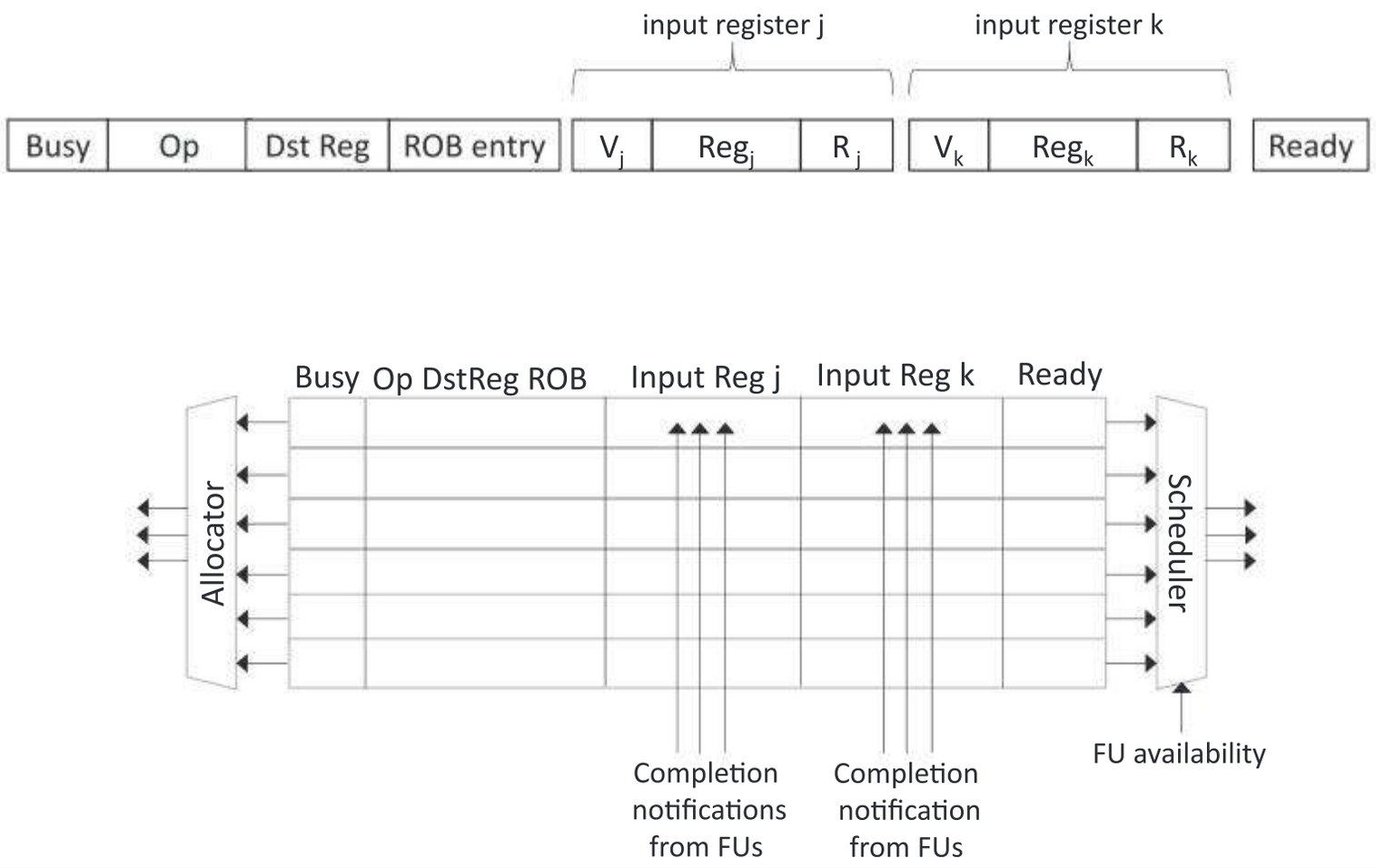


Figure 3.19 The entry format (top) and structure (bottom) of a centralized instruction scheduler.

instructions. Each scheduler entry includes comparators that constantly compare the numbers of the two input physical registers in the entry (Reg_j and Reg_k), to the physical registers that the functional units just updated. If matches are discovered, either or both registers are marked as ready (R_j and/or R_k set to true).

Figure 3.20 presents the organization of the ROB. Each entry tracks the status of a single instruction. It tracks if the execution of the instruction has completed (`Ready`), if the instruction produced a register output (`DestV`), the numbers of the architectural and physical registers written by the instruction (`Dst AReg` and `PReg`), the instruction address (PC), and information that encodes if the instruction caused a misprediction or exception, including the type of the misprediction or exception. The ROB is managed as a circular buffer using head and tail pointers.

Figure 3.21 shows the block diagram of the processor execution engine and back-end, including the instruction window and the ROB. In this figure, solid lines indicate communication of data values or addresses, while dashed line indicates communication of instructions, control signals, and notifications. The block diagram also presents the basic structure of the load-store unit that we will discuss in Section 3.7.

Let's review the steps involved with instruction execution after renaming:

1. *Issue (in-order):* All instructions allocate an entry in the ROB. Arithmetic, logical, and branch operations allocate an entry in the scheduler. Load/store operations are sent to the *load/store queue* for memory address disambiguation

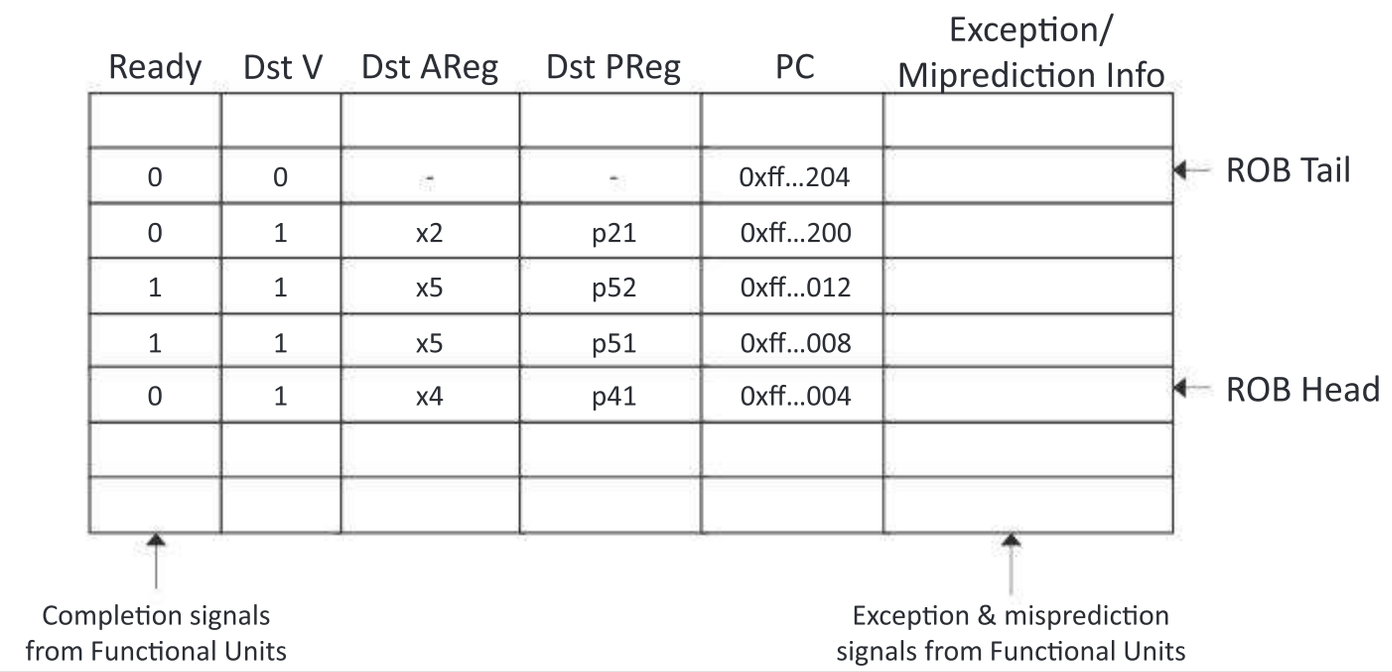


Figure 3.20 The structure of the reorder buffer (ROB). The head and tail pointers are used to manage its entries as a circular buffer.

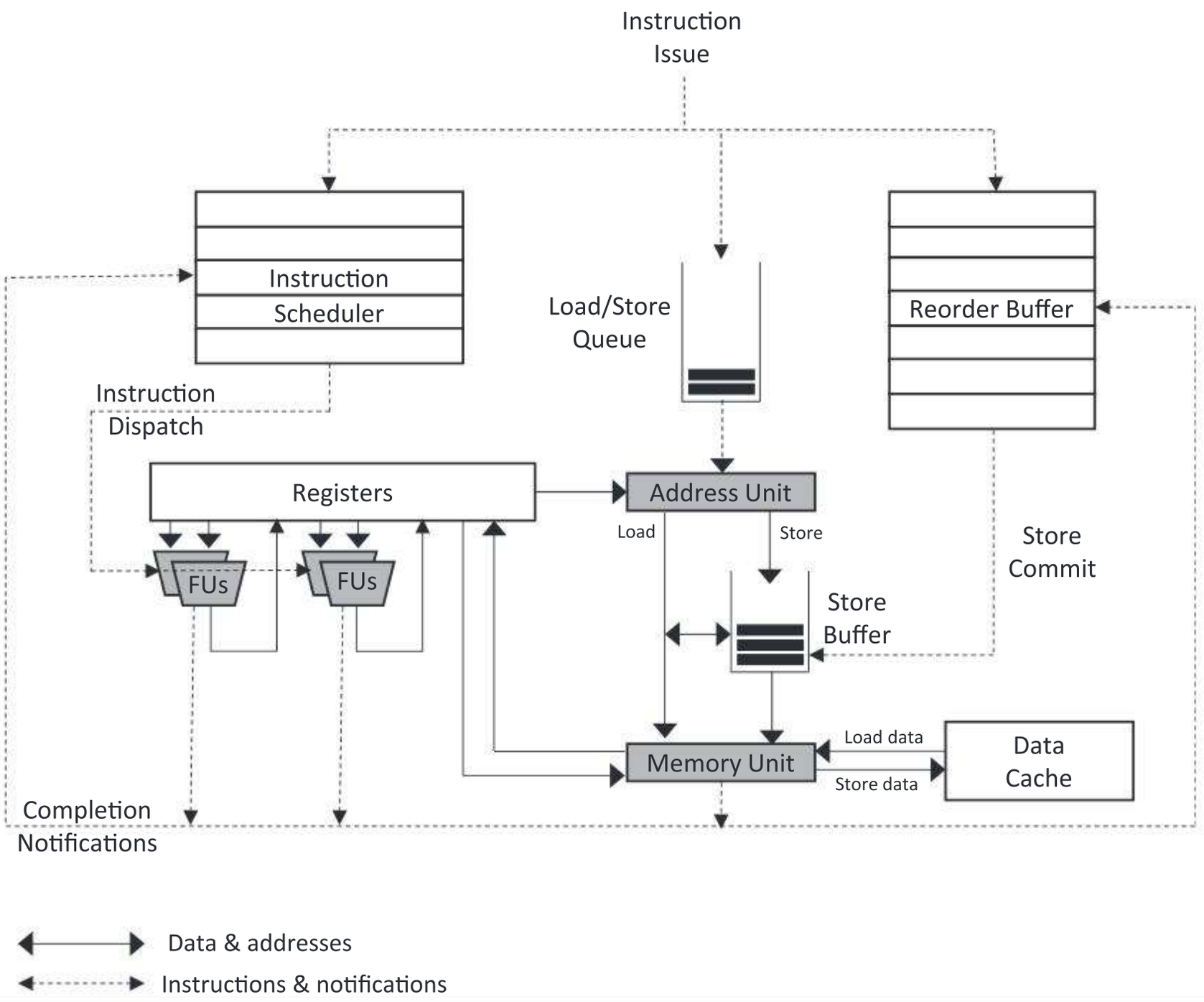


Figure 3.21 The basic structure of the execution engine and back-end of a dynamically scheduled superscalar processor with a reorder buffer for speculation recovery. Instructions are issued from the front-end to the reorder buffer and the instruction scheduler. The instruction scheduler tracks readiness of input operands and can determine when data hazards are resolved for each instruction to be issued either to a functional unit (addition, subtraction, multiplication, division, etc.) or the memory unit (loads, stores). When instructions complete execution, they update the value of their physical register and notify the instruction scheduler (to make dependent instructions ready) and the reorder buffer.

and scheduling (see Section 3.7). If the ROB, the hardware scheduler, or the load/store queue is full, we stall the processor front-end until space is available.

2. *Dispatch (out-of-order):* On every clock cycle, the scheduler selects a subset of ready instructions for execution. The selection circuit considers the number and type of available functional units, as well as the type and age of ready instructions. Since instructions must commit in order, it is preferable to schedule the oldest of the ready instructions first. Selected instructions are removed from the instruction window and sent to the matching functional units. For pipelined functional units we can issue one instruction per cycle.
3. *Completion (out-of-order):* A functional unit executes an issued instruction by first reading its input data from the URF. When the arithmetic operation completes, the functional unit writes back the result to the destination physical register. It also generates completion notifications for the ROB and the scheduler. The ROB uses this notification to mark the corresponding instruction ready and to potentially record any detected exceptions or mispredictions. The scheduler is notified that the destination physical register is now ready in order to perform *instruction wakeup*. Every scheduler entry compares the destination register number to its input register numbers to determine if either or both input registers became ready. If both input registers in a scheduler entry are now ready, the instruction is now ready and will be considered for scheduling.
4. *Commit or recovery (in-order):* An instruction can commit when it reaches the head of the ROB and it has completed its execution without generating any exceptions. An additional commit condition is that branch instructions should not be mispredicted and load instructions should not be involved with incorrect speculation during memory disambiguation. For most instructions, commit means updating the architectural mapping for its destination and removing the instruction from the ROB. Committing a store is similar except that we also notify the store buffer that it is now safe to update memory. When a branch or load that led to mispeculation or an instruction that causes an exception reaches the head of the ROB, the ROB and the whole processor pipeline is flushed and execution restarts at the correct PC (the right branch target, the load address, or the PC for the exception handler).

Each of the four steps handles multiple instructions per cycle. For example, a pipeline may be sized to issue 4 instructions per cycle, dispatch 6 instructions per cycle, and commit 4 instructions per cycle. It is common to have the dispatch width wider than the issue and commit width of the pipeline. Assuming there are enough functional units available, this allows to quickly process instructions after a critical dependence has been resolved. For example, a load that misses in the L1 and L2 caches may cause many dependent instructions to wait in the scheduler until the load completes. The wider dispatch allows fast processing of this backlog when the load completes.

Dynamic Scheduling: Examples

Example Assume a dynamically scheduled, superscalar processor with three functional units for floating-point loads/stores, floating-point add/subtract, and floating-point multiply/divide. The latencies for the functional units are given in Figure 3.2: add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles. In the following renamed code segment with the original code shown in the comments, pf6 denotes a floating-point physical register, while p2 denotes an integer physical register. Show what the status of the instruction window, ROB, and floating-point physical registers looks like when the `fmul.d` is ready to go to commit.

```
fld       pf6,0(p2)       // fld      f6, 0(x2)
fld       pf2,0(p3)       // fld      f2, 0(x3)
fmul.d    pf0, pf2,pf4    // fmul.d   f0, f2, f4
fsub.d    pf8, pf2,pf6    // fsub.d   f8, f2, f6
fdiv.d    pf10,pf0,pf6    // fdiv.d   f0, f0, f6
fadd.d    pf16,pf8,pf2    // fadd.d   f6, f8, f2
```

Answer Figure 3.22 shows the result for the ROB, the scheduler, and the floating-point physical registers. Notice that although the `fsub.d` instruction has completed execution, it does not commit until the `fmul.d` commits. The `fadd.d` can execute and complete ahead of the `fdiv.d` instruction because register `f6` has been renamed and the WAR hazard is avoided. The `fdiv.d` instruction will start execution in the following cycle, as the floating-point multiply/divide unit just became available.

The preceding example illustrates the key feature of superscalar processors with dynamic scheduling and a ROB: no instruction after the earliest uncompleted instruction (`fmul.d`) is allowed to commit. Hence it is possible to maintain a precise exception model.

For example, if the `fmul.d` instruction caused a floating-point exception, we could simply wait until it reached the head of the ROB and raise the exception, flushing any other pending instructions from the ROB. Because instruction commit happens in order, this yields a precise exception. In contrast, if we committed the outcome of instructions as soon as they completed their execution, the `fsub.d` and `fadd.d` instructions could both complete before the `fmul.d` raised its exception. The values in registers `f8` and `f6`, the destinations of the `fsub.d` and `fadd.d` instructions, would be overwritten, in which case the interrupt would be imprecise.

Reorder buffer					
Entry	Busy	Instruction	Status	Ready	Destination
1	No	`fld pf6, 0(p2)`	Committed	Yes	`pf6 (f6)`
2	No	`fld pf2, 0(p3)`	Committed	Yes	`pf2 (f2)`
3	Yes	`fmul.d pf0, pf2, pf4`	Completed	Yes	`pf0 (f0)`
4	Yes	`fsub.d pf8, pf2, pf6`	Completed	Yes	`pf8 (f8)`
5	Yes	`fdiv.d pf10, pf0, pf6`	Dispatched	No	`pf0 (f10)`
6	Yes	`fadd.d pf16, pf8, pf2`	Completed	Yes	`pf16 (f6)`

Instruction Scheduler											
Entry	Busy	Op	Dst	ROB	V_j	Reg_j	R_j	V_j	Reg_j	R_j	Ready
1	No	`fld`	`pf6`	1	-	-	-	-	-	-	No
2	No	`fld`	`pf2`	2	-	-	-	-	-	-	No
3	No	`fmul.d`	`pf0`	3	-	-	-	-	-	-	No
4	No	`fsub.d`	`pf8`	4	-	-	-	-	-	-	No
5	Yes	`fdiv.d`	`Pf0`	5	Yes	`pf0`	Yes	Yes	`Pf6`	No	Yes
6	No	`fadd.d`	`pf16`	6	-	-	-	-	-	-	No

Floating-point Rename Registers							
Reg	`pf0`	`pf2`	`pf4`	`pf6`	`pf8`	`pf10`	`pf12`
Status	`Used/Ready`	`Arch`	`Arch`	`Arch`	`Used/Ready`	`Used/Not ready`	`Used/Ready`
Value	`Mem[p3]+X`	`Mem[p3]`	`X`	`Mem[p2]`	`Mem[p3]-Mem[p2]`	`Y`	`2*Mem[p3]-Mem[p2]`

Figure 3.22 At the time the `fmul.d` is ready to commit, only the two `fld` instructions have committed, although several others have completed execution. The `fmul.d` is at the head of the ROB, and the two `fld` instructions are there only to ease understanding. The `fsub.d` and `fadd.d` instructions will not commit until the `fmul.d` instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The `fdiv.d` is now ready to be dispatched to an execution unit as it uses the value just calculated by the `fmul.d (pf0)`. The instruction scheduler will select it for execution shortly. The status, busy, and instruction fields in the ROB are shown for illustration purposes. We do not show the entries for the load/store queue, but these entries are kept in order.

Some users and architects have decided that imprecise floating-point exceptions are acceptable in high-performance processors because 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 because the program must transparently resume execution after handling such an exception.

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

Example Consider the following code example with a loop:

```
Loop: fld f0,0(x1)
      fmul.d f4,f0,f2
      fsd f4,0(x1)
      addi x1,x1,-8
      bne x1,x2,Loop //branches if x1≠x2
```

Assume that we have predicted the branch as taken and have dispatched all the instructions in the loop twice. Let's also assume that the `fld` and `fmul.d` from

the first iteration have committed and all other instructions have been dispatched and have completed their execution. Normally, the store would wait in the ROB for both the effective address operand (`x1` in this example) and the value (`f4` in this example) to be computed. Because we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued. What is the status of the reorder buffer and the physical registers at this point?

Answer Figure 3.23 shows the result for the reorder buffer and the integer and floating-point physical registers.

Because store values are not yet written in memory and past register values are preserved in physical registers until an instruction commits, the processor can 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 3.23. The instructions prior to the branch will commit when they reach the head of the ROB. When the branch reaches the head of the ROB, the ROB is cleared, the register renaming map is restored from the architectural map, and the processor begins fetching instructions from the correct branch target.

Many speculative processors try to recover as early as possible once they discover that a branch is mispredicted. This involves clearing the ROB entries for all instructions after the mispredicted branch, allowing those that are before the branch in the ROB to continue. Recovering the right renaming state before

Reorder buffer

Entry	Busy	Instruction	Status	Ready	Destination
1	No	`fld f0, 0(x1)`	Committed	Yes	`pf10 (f0)`
2	No	`fmul.d f4, f0, f2`	Committed	Yes	`pf14 (f4)`
3	Yes	`fsd f4, 0(x1)`	Completed	Yes	–
4	Yes	`addi x1, x1, -8`	Completed	Yes	`p11 (x1)`
5	Yes	`bne x1, x2, Loop`	Completed	Yes	–
6	Yes	`fld f0, 0(x1)`	Completed	Yes	`pf20 (f0)`
7	Yes	`fmul.d f4, f0, f2`	Completed	Yes	`pf24 (f4)`
8	Yes	`fsd f4, 0(x1)`	Completed	Yes	–
9	Yes	`addi x1, x1, -8`	Completed	Yes	`p21 (x1)`
10	Yes	`bne x1, x2, Loop`	Completed	Yes	–

Floating-point Physical Registers

Reg	pf0	pf2	pf4	pf10	pf14	pf20	pf24
Status	Free	Arch	Free	Used/Ready	Used /Ready	Used /Ready	Used /Ready
Value	X	Y	Z	Mem[A]	Mem[A]+Y	Mem[A-8]	Mem[A-8]+Y

Integer Physical Registers

Reg	p1	p2	p11	p21
Status	Arch	Arch	Used /Ready	Used /Ready
Value	A	B	A-8	A-16

Figure 3.23 Only the `fld` and `fmul.d` instructions have committed, although all the others have completed execution. Thus no instruction scheduler entries are busy and none are shown. The remaining instructions will be committed as quickly as possible. The first two reorder buffers are empty, but are shown for completeness.

restarting instruction at the correct branch successor is more difficult. This is often done by taking a complete checkpoint of the renaming map when a branch is issued. If the branch is later discovered to be mispredicted, the corresponding checkpoint is restored. The amount of hardware available for checkpointing renaming state limits the number of pending branches in the pipeline. In speculative processors, performance is more sensitive to branch prediction because the impact of a misprediction is typically high. Thus all aspects of handling branches—prediction accuracy, latency of misprediction detection, and misprediction recovery time—increase in importance.

Exceptions are handled by not raising the exception until the instruction is ready to commit. If an instruction generated an exception during its execution, the exception is recorded in the ROB. If a branch misprediction is detected for a branch older than this instruction, 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, it is not as critical.

Dynamic Scheduling: The Details

Figure 3.24 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. A store updates memory only when it reaches the head of the ROB, which ensures that memory is not updated until an instruction is no longer speculative. Figure 3.24 has one significant simplification for stores, which is unneeded in practice. Figure 3.24 requires stores to wait for both input register values before address calculation. In reality, however, the value to be stored need not be available until *just before* the store commits and updates memory.

The speed of some of the steps in Figure 3.24 is critical to the performance of OOO processors. Suppose, for instance, that we have two simple arithmetic instructions with a data dependence: an add followed by a subtract. In our simple five-stage pipeline in Appendix C, these two dependent instructions will execute in back-to-back cycles without any stalls using forwarding logic. For the superscalar processor we described in this section, once the add instruction completes execution, the following steps must occur for subtract to execute: 1) the instruction scheduler entry for subtract must be updated to indicate the instruction is ready (wakeup); 2) the scheduler must select subtract for execution (issue); 3) and the input register values for subtract must be read from the URF. If every step takes a whole clock cycle, these two instructions will execute three cycles apart, implying that a higher amount of ILP is needed to keep the functional units busy. Many processor designs cut back on the latency for execution of dependent instructions by first combining instruction wakeup and scheduling in a single clock cycle. This is possible as the notification of completion can happen early

Current Step	Wait until	Next Step and Bookkeeping
Issue for all instructions	ROB (b) available AND (LSQ (ls) available for load/store OR SCHED (w) available for other instructions)	ROB[b].Ready=No; ROB[b].Op=Opcode; ROB[b].DstV= rd_v; ROB[b].DstAreg=rd_arch; ROB[b].DstPreg=rd; ROB[b].PC = PC;
Issue for non loads/stores		SCHED[w].Busy=Yes; SCHED [w].Op=Opcode; SCHED [w].DstReg=rd_phy; SCHED[w].ROB=b; SCHED [w].Vj=Yes; SCHED[w].Regj=rs; SCHED[w].Rj=URF[rs].Ready SCHED [w].Vk=Yes; SCHED[w].Regk=rt; SCHED[w].Rk=URF[rt].Ready SCHED [w].Ready = (SCHED[w].Rj AND SCHED[w].Rk)
Issue for loads		LSQ[ls].Busy=Yes; LSQ[ls].Op=Opcode; LSQ[ls].DstReg=rd_phy; SCHED[w].ROB=b; LSQ[w].Vk=Yes; LSQ[ls].Regk=rt; LSQ[ls].Rk=URF[rt].Ready; LSQ[w].Imm = Imm If (LSQ[ls].Rk) LSQ[ls].Readuy= True;
Issue for stores		LSQ[ls].Busy=Yes; LSQ[ls].Op=Opcode; SCHED[w].ROB=b; LSQ[w].Vj=Yes; LSQ[ls].Regj=rd; LSQ[ls].Rj=URF[rd].Ready; LSQ[w].Vk=Yes; LSQ[ls].Regk=rt; LSQ[ls].Rk=URF[rt].Ready; LSQ[w].Imm = Imm; If (LSQ[ls].Rk AND LSQ[ls].Rj) LSQ[ls].Ready = True;
Instruction Dispatch	SCHED[w].Ready	Select instruction; send to functional unit
Execution non loads/stores	Instruction dispatched	Op = SCHED[w].Op; A = SCHED[w].Regj; B=SCHED[w].Regk; D = SCHED[w].DstReg; URF[D] = URF[A] Op URF[B]
Load/store Step 1	LSQ[ls].Ready and oldest load/store instruction	Addr = URF[LSQ[ls].Regk]+Imm;
Load Step 2	All earlier stores have a different address	URF[rd]=Mem[Addr]; URF[rd].Status=Ready;
Store Step 2	Store is committed	Mem[Addr] = URF[LSQ[ls].Regk];

Notify instruction completion	Execution done	ROB[b].Ready=True; ROB.Exception=exception;
		∀ w in SCHED {
		If (SCHED[w].Regk==dst) SCHED[w].Rk = True;
		If (SCHED[w].Regj==dst) SCHED[w].Rj = True;
		If (SCHED[w].Rk AND SCHED[w].Rj) SCHED[w]=Ready;
		}
		∀ ls in LSQ {
		If (LSQ[ls].Regk==dst) LSQ[w].Rk = True;
		If (LSQ[ls].Regj==dst) LSQ[w].Rj = True;
		If (LSQ[ls].Op=load AND LSQ[w].Rj) LSQ[ls]=Ready;
		If (LSQ[ls].Op=store AND LSQ[w].Rj AND LSQ[ls].Rk) LSQ[w]=Ready;
		}

Commit	Instruction is at the head of the ROB (entry h) and ROB[h].Ready=Yes	If (ROB[h].Op == Branch)
		If (branch is mispredicted) /* misprediction case*/
		{clear ROB; clear SCHED; clear LSQ; reset RenamingMap; fetch branch dest;}
		Else if (ROB[h].exception = True) /* exception case */
		{clear ROB; clear SCHED; clear LSQ; reset RenamingMap; raise exception;}
		else {
		if (ROB[h].Op == Store) {notify LSQ;} /* store case*/
		ROB[h].busy=NO; /* clear entry */
		If (ROB[h].DstV) /* update status of physical and arch registers*/
		{a=ROB[h].DstAreg; p=ROB[h].DstPreg;
		URF[ArchMap[a]].Status=Free;
		URF[p].Status = Architectural;
		ArchMap[a] = p;}

Figure 3.24 Steps in instruction execution and what is required for each step. For the issuing instruction, `rd` is the destination, `rs` and `rt` are the sources, `w` is the instruction scheduler entry allocated, `b` is the assigned ROB entry, and `h` is the head entry of the ROB. `SCHED` is the instruction scheduler data structure. `ROB` is the reorder buffer data structure. URF is the unified register file data structure. `LSQ` is the load store unit data structure. `RenamingMap` and `ArchMap` are the renaming and architectural map data structures.

in the clock cycle, in parallel with the execution of the add instruction. This technique generalizes for any instruction with fixed latency, where we can notify completion early to mitigate latency of wakeup and scheduling for any dependent instructions. Next, it is possible to avoid the extra latency of reading inputs from URF by reading ready physical registers during dispatch and storing them in the instruction scheduler. The add result can be passed between functional units using additional forwarding paths. While such optimizations are possible and reduce pipeline length and CPI, they increase complexity and create a trade-off with the clock cycle time for the processor.

A common way to reduce the increasing complexity of a centralized instruction scheduler is to use a distributed scheme. An extreme design point would be to associate a scheduler of a few entries (4−10) with each functional unit. At issue time, each instruction is assigned to the scheduler of a specific functional unit, which tracks when the instruction is ready to execute. These distributed schedulers are often referred to as *reservation stations*. A small reservation station with 10 entries is significantly simpler and faster for wakeup and dispatch than a centralized scheduler with more than 100 entries. For instance, it is faster to select the oldest ready instruction out of 10 rather than out of 100. It is also easier to select a single instruction to be dispatched per cycle in the local functional unit rather than 4 to 8 instructions in a centralized scheduler. In a large scheduler the simplest way to infer age is by the position on the instruction entry. This requires that, as some instructions are dispatched, remaining instructions are compacted so that older instructions appear at the bottom of the scheduler while empty entries for newly issued instructions are at the top. In a small reservation station, we can compare tags that encode instruction age regardless of position. It is also easier to implement compaction with 10 entries rather than 100. The downside of distributed reservation stations is that their entries may not be evenly utilized. If the mix of instructions in the program deviates from the common case assumed by the processor design when sizing each structure, one functional unit may have a full reservation station, while another functional unit may be an empty one. A full reservation station will stall instruction issue. The centralized scheduler can use any entry for an instruction destined for any functional unit.

Recent AMD processors use of a hybrid design. For example, the Milan processor pipeline uses a distributed design with 4 schedulers for integer instructions and 2 schedulers for floating-point instructions. Each of the schedulers can be used for any integer and floating-point instruction, respectively, and can dispatch instructions to any integer and floating-point functional unit, respectively. Hence they avoid the underutilization problem with per functional unit reservation stations. Each of the 6 distributed scheduling structures is smaller than a centralized scheduler and must dispatch fewer instructions per cycle. The one additional complexity of this distributed design is that the 4 integer and 2 floating point schedulers must coordinate to avoid dispatching instructions to the same functional unit. Because of the trade-off between size and clock frequency for compacting reservation stations, some recent processors utilize a noncompacting reservation station design with an age matrix. The age matrix does not provide a total instruction order but guarantees that we can dispatch the oldest instruction plus any other N-1 instruction for an N-wide scheduler.

3.7 Overcoming Memory Dependences with Dynamic Disambiguation

The previous section focused on dynamic scheduling in the presence of data dependences through registers. Let's now look at data dependences through

memory, in other words, dependences between loads and stores through memory locations. These dependences are significantly more difficult to detect and manage. While input and output register names are fully encoded within instructions and dependence checks can be done early in the pipeline, this is not the case for memory addresses. Load and store instructions must be processed by an address generation unit before their effective memory addresses are known. Moreover, the address calculation may be stalled due to register dependences with older arithmetic or load instructions that manipulate pointer values.

The simple but slow solution would be to execute all load and store instructions in program order. The oldest memory instruction waits to clear any data dependences through registers, before it calculates its address that then is sent to the cache hierarchy. Unfortunately, this simple solution would be detrimental to the performance of a multiple-issue processor because stores cannot be allowed to overwrite data in caches before they commit. Despite the use of branch prediction and register renaming, the loads for subsequent iterations of a loop would not access the data cache before the stores of the current iteration commit. A processor using this approach cannot exploit ILP across loop iterations.

Figure 3.21 suggests a better-performing alternative called *load bypassing*. Load and stores calculate their effective addresses as soon as data dependencies allow and stores are not dispatched to the cache hierarchy until they commit. However, a load instruction is allowed to bypass older stores if there is no data dependence: all older store instructions have calculated their effective addresses which are different than the effective address for the load. Load bypassing requires that store buffer entries include comparators to facilitate a fast comparison between the load address and the addresses of all pending stores. This optimization allows loads from further iterations of a loop to access caches early, hide the latency of any misses, and resolve the dependence on any subsequent arithmetic instructions, even though older stores have not committed yet.

An additional optimization is *store forwarding*. If address comparison shows that a load is data dependent on an older store, we allow the load instruction to retrieve its data from the input physical register for the store, without accessing the memory hierarchy. Similar to the case of register dependences between more than two instructions, the load must retrieve the data from the youngest store it depends on. Store forwarding is an important optimization for x86 processors. The small number of architectural registers (8 integer registers) forces compilers to frequently spill variables from registers to memory, only to reload these variables a few instructions later. Hence, it is common for superscalar x86 processors to overlap the execution of multiple pairs of spill (store) and restore (load) instructions. Store forwarding accelerates the dependent load and any subsequent, dependent instructions.

Speculative Memory Disambiguation

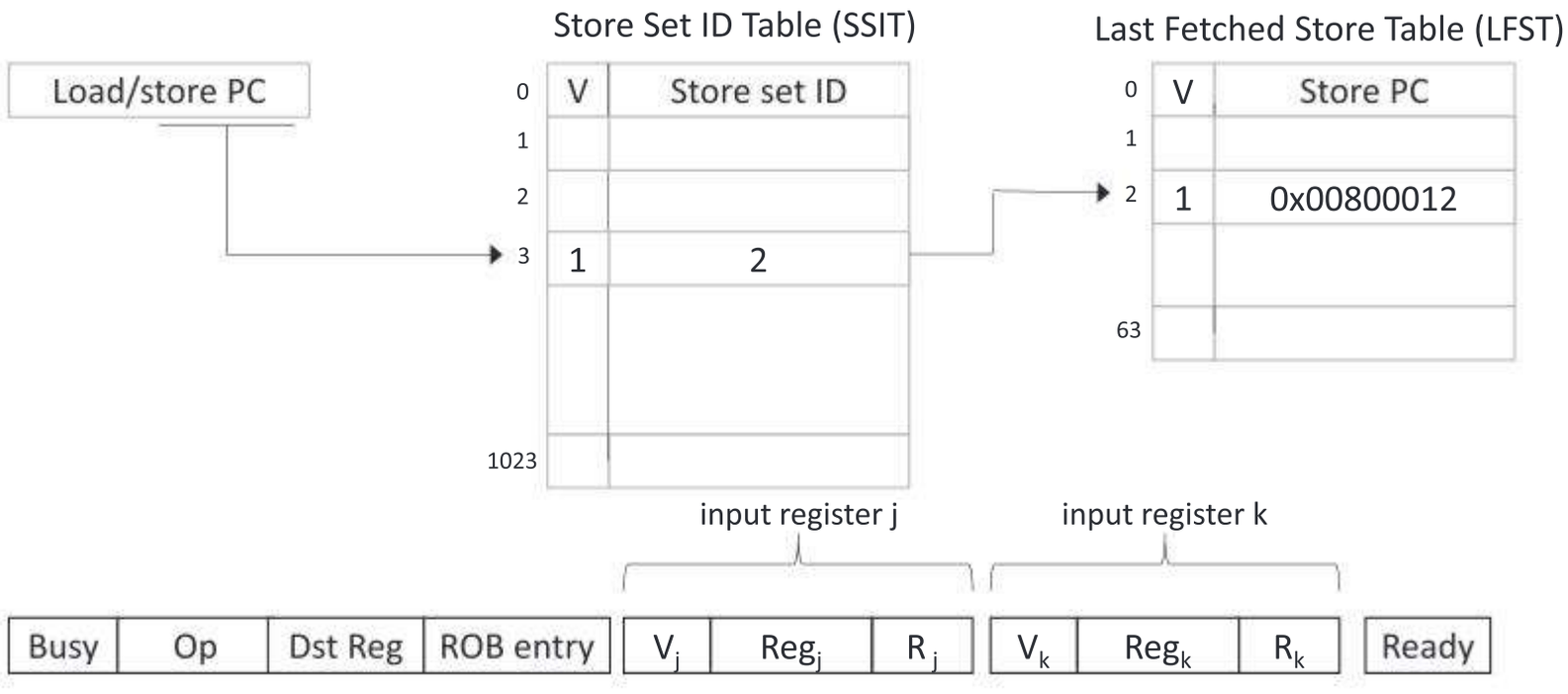
The requirement that loads cannot bypass older stores before all addresses are available for dependence checks is still a significant performance impediment.

This has led to speculative memory disambiguation: loads are allowed to bypass older stores that have not yet calculated their memory address, speculating that there is no dependence. When the missing store addresses become available, the actual dependence checks are done. If speculation was incorrect, the load and all dependent instructions that have used its result need to be reexecuted. Since selective instruction reexecution is difficult in complex pipelines, the recovery mechanism often used is to flush the whole pipeline after the load instruction, as in the case of mispredicted branches or exceptions.

Several studies have quantified that naively speculating on the lack of memory dependences can be worse than not speculating at all [Moshovos et al., 1997, Chrysos and Emer 1998]. This has led all recent OOO processors to implement a scheme for speculative but selective memory disambiguation. Similar to learning patterns for branch prediction, the processor learns which loads are likely to be dependent on pending stores. These loads are not allowed to execute early if older stores have not calculated their addresses yet. The processor can learn these patterns for loads and stores in loops by naively speculating on memory dependences when no other information is available. If a memory dependence is later discovered, the PC for the load and store instructions involved with the dependence can be stored in hardware structures. When the load is executed again, these structures can be queried to help determine if the load is likely dependent on a pending store that has not calculated its address yet.

We will describe in more detail the *store sets* scheme proposed by Chrysos and Emer in 1998, as an example design for speculative but selective memory disambiguation. The store sets scheme focuses on capturing information on loads and stores that were recently involved in memory dependence mispeculations. It can capture concisely that a load may depend on a group (set) of stores or that group of loads may depend on a single store. Figure 3.25 shows the two hardware structures required. The *store set ID table (SSIT)* tracks the store set ID for loads and stores that were recently involved in a memory dependence mispeculation. The SSIT is indexed with a few bits from the PC of load and store instructions and all its entries are initialized to invalid. The store set ID is a short integer (e.g., 0−63). The *last fetched store table (LFST)* tracks the instruction address (PC) for the last fetched store instruction in each store set.

When the hardware discovers a memory dependence mispeculation, it allocates a store set identifier and inserts that identifier in the SSIT entry for both the load and the store involved in the mispeculation. If one of the two instructions already belongs to a store set (valid SSIT), its existing store set ID is used instead. If both instructions belong to different store sets, the sets are merged. When a store is dispatched, it looks up its SSIT entry. If it is valid, then it marks its instruction address (PC) in the store set's LFST entry. The LFST entry is reset when the store calculates its address. A load checks the two structures when it has calculated its address and it is ready to be dispatched to the memory hierarchy. A load is not allowed to proceed if its SSIT is valid and the corresponding



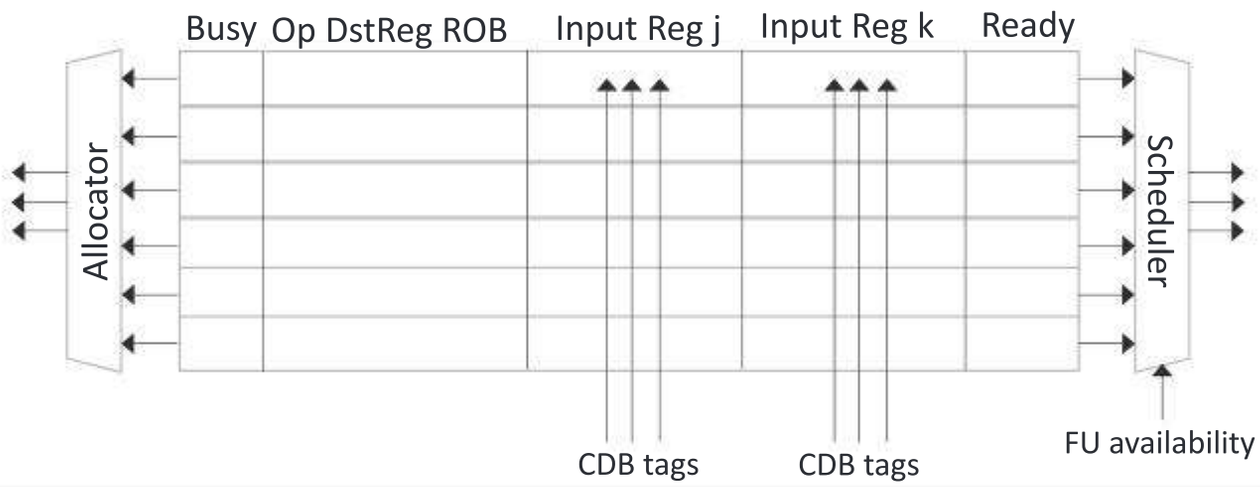


Figure 3.25 The hardware needed for the store set scheme for speculative but selective memory disambiguation [Chrysos and Emer, 1998]. The original paper suggested a store set ID table (SSIT) with 1K to 4K entries and a last fetched store table (LFST) with 64 to 128 entries. Depending on the implementation of the rest of the processor, an LFST entry may buffer the instruction address (PC), the ROB entry number, or the load/store queue entry number for the last fetched store.

LFST entry identifies a pending store that has not yet calculated its address. In all other cases (invalid SSIT entry, or valid SSIT entry pointing to an empty LFST entry), the load is allowed to speculatively issue ahead of older stores that do not have their addresses yet. An invalid SSIT entry suggests that this load has not been recently involved with dependence mispeculations. A valid SSIT entry pointing to an empty LSFT entry suggests that all stores that this load may be dependent upon have completed their address calculation.

The store set scheme gradually captures sets of dependent load and store instructions that have caused dependence mispeculations. Its accuracy is limited by several factors. First, different loads and store instructions may map to a single SSIT entry, causing unnecessary merging of store sets. Second, if a dependence mispeculation for a certain load/store pair is infrequent, it may be better for performance to not track it in the store set structures. Finally, store sets can only grow over time, making the processor increasingly less aggressive about load

bypassing. A simple way to address these issues is to reset the two structures every few millions of instructions and allow the hardware to rediscover the most critical dependences to preserve when speculating.

As we will see when we discuss memory consistency in Chapter 5, multiprocessor systems introduce further ordering constraints for loads and stores even after their memory addresses are known. In some cases, it can be incorrect to reorder even loads. Speculative memory disambiguation allows high performance in the presence of such constraints. The hardware can learn when reordering leads to memory consistency violations and reorder memory accesses for higher performance only when it is unlikely to cause a violation.

Speculative memory disambiguation is a simple and restricted form of *value prediction,* which attempts to predict the value that will be produced by an instruction. Value prediction could, if it were highly accurate, eliminate data flow restrictions and achieve higher rates of ILP. Despite many researchers focusing on value prediction in the past 20 years in dozens of papers, the results have not been sufficiently attractive to justify general value prediction in commercial processors thus far.

3.8 Advanced Issues in Modern Superscalar Processors

The Challenge of Multiple Issues per Cycle

Issuing multiple instructions per clock in a dynamically scheduled processor is complex for the simple reason that the instructions within a bundle may depend on one another. All the steps that we described for instruction execution, and in particular the issue, dispatch, and commit steps, must take into account the dependences between instructions that execute in parallel. The complexities of renaming multiple instructions per clock were covered in Section 3.5. Here we expand on the impact on the rest of the pipeline.

Two different approaches have been used to manage multiple instructions per clock in a superscalar processor. One approach is to run this step in half a clock cycle so that two instructions can be processed in one clock cycle in a sequential manner. This approach cannot be easily extended to handle four or more instructions per clock, unfortunately. A key observation is that we cannot simply pipeline away the problem. If instruction issue takes multiple clocks, for example, we must be able to update renaming tables so that a dependent instruction issuing on the next clock cycle can use the updated information. The second alternative is to build the logic necessary to handle two or more instructions at once, including any possible dependences between the instructions. This is the approach used by modern superscalar processors in key stages of their deep pipelines.

The issue step is one of the most fundamental bottlenecks in wide issue processors. To correctly issue N instructions in a bundle, we must:

- Allocate and update N entries in the ROB. If less than N entries are available, only a subset of the instructions are issued, the oldest ones in the bundle.
- Allocate and update N entries in the instruction scheduler and load-store queues. If less than N entries are available, only a subset of the instructions are issued, the oldest ones in the bundle.
- Allocate and update up to N physical registers. If less than N registers are available, only a subset of the instructions are issued, the oldest ones in the bundle.
- Correctly rename all input operands for N instructions in the bundle, taking into account the data and name dependencies between instructions in the bundle (see Section 3.5).

Because the number of possibilities for dependences grows as the square of the number of instructions in the bundle, the issue step is a likely bottleneck for clock frequency and the reason that few processors have attempted to go beyond eight instructions per cycle. Figure 3.18 in Section 3.5 provides an example of the complexity of renaming multiple instructions per cycle. and makes it clear why issue rates for high-volume processors have grown from 3–4 to only 4–8 in the past 20 years.

Dispatching multiple instructions per cycle is simpler as, by that point in the pipeline, data and name dependences are resolved and instructions can be managed independently for the most part. Semiconductor scaling has also made it easy to have a large number of functional units to handle the ready instructions in every clock cycle, hence several processors that can dispatch more instructions per cycle than their issue width. This helps quickly handle a burst of ready instructions after a load that experiences a cache miss. Similarly committing multiple instructions from the ROB is not particularly complicated as long as the commit logic ensures that instructions behind the first one to report an exception or misprediction are canceled.

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, dynamically scheduled processor with speculation:

```
Loop: ld x2,0(x1) //x2=array element
      addi x2,x2,1 //increment x2
      sd x2,0(x1) //store result
      addi x1,x1,8 //increment pointer
      bne x2,x3,Loop //branch if not last
```

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. Assume that up to two instructions of any type can commit per clock.

Answer Figures 3.26 show the performance for a two-issue, dynamically scheduled processor with speculation. In this case, where the branch at the end of the loop can be a critical ILP limiter, speculation helps significantly. By speculating branches, following instructions can execute, but not commit, before early branches are resolved, allowing for fast execution. For example, the second load can complete execution ahead of the older branch. If the processor did not use speculation, resolving branches would be the key performance bottleneck.

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.

Iteration Number	Instruction	Issue at clock cycle number	Dispatch at clock cycle number	Complete at clock cycle number	Commits at clock cycle	Comment
1	`ld p2, 0(p1)`	1	2	4	5	First instruction
1	`addi p22, p2, 1`	1	4	5	6	Wait for ld
1	`sd p22, 0(p1)`	2	5		6	Wait for addi
1	`addi p21, p1, 8`	2	2	3	7	Commit in order
1	`bne p21, p3 Loop`	3	3	4	7	Wait for addi
2	`ld p32, 0(p21)`	4	5	7	8	No execution delay
2	`addi p42, p32, 1`	4	7	8	9	Wait for ld
2	`sd p42, 0(p21)`	5	8		9	Wait for addi
2	`addi p31, p21, 8`	5	6	7	10	Commit in order
2	`bne p31, p3 Loop`	6	7	8	10	Wait for addi
3	`ld p52, 0(p31)`	7	8	10	11	Earliest possible
3	`addi p62, p52, 1`	7	10	11	12	Wait for ld
3	`sd p62, 0(p31)`	8	11		12	Wait for addi
3	`addi p41, p31, 8`	8	9	10	13	Executes earlier
3	`bne p41, p3 Loop`	9	10	11	13	Wait for addi

Figure 3.26 The time of dispatch, issue, execution completion, and commit of a dual issue version of our pipeline with speculation. Note that the `ld` following the `bne` can start execution early as the branch is predicted taken. We assume that loads require one cycle for address generation and one cycle for cache access (all hits); stores can be buffered as needed; and instructions can be scheduled for execution in the same cycle an older dependent instruction completes its execution.

To maintain most of the advantage while minimizing the disadvantages, many pipelines with speculation will allow only low-cost exceptional events (such as cache misses) to be handled in speculative mode. If an expensive exceptional event occurs, such as 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 became more aggressive in their use of speculation, 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

The wide-issue and deeply pipelined processors we considered in this chapter overlap the execution of tens of instructions. This means that at any point in time, it is possible to have multiple pending speculative branches. Three more characteristics increase this possibility: (1) the high branch frequency in many programs, (2) the clustering of branches in some code segments, and (3) the long delays necessary to resolve branches in a functional unit. Database programs and other less structured integer computations often exhibit the first two 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 pipelines.

The primary complication of speculating through multiple branches is deciding when to update branch prediction state, such as history registers, prediction counters, and the target prediction structures. If there is only one speculative branch pending, we can wait for the branch to get resolved before we update the prediction state. Note that we need to update the predictor state on both correct and incorrect predictions. With multiple speculative branches, there is a trade-off. If we wait until a branch is resolved before we update the prediction tables, the history of this branch will not be available when we predict the next few branch instructions. If we update prediction state as soon as a branch is predicted to be taken or not taken, the state now contains information that may not be accurate. Several processor designs handle this trade-off by updating prediction state early and by correcting these updates in some or all prediction structures if the branch is resolved in the opposite direction than the original prediction. For example, some processors maintain two RASs, one updated speculative when a function call instruction is fetched, and one updated only when function call instructions are no longer speculative.

So far, all processors predict a single branch per clock cycle. A single PC value is used to fetch a bundle of instructions from the cache and to index into the prediction structures for the next PC. The prediction structures track the information for the first taken branch within the bundle, if any. Predicting through multiple

taken branches per clock cycle would introduce significant complexity and power consumption and it is unlikely that the performance benefits would justify the cost. The low-hanging fruit is loops with small loop bodies that can be effectively captured with a loop cache, a common feature in many modern processors.

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 because whenever speculation is wrong, it consumes excess energy in two ways:

1. Instructions that are speculated and whose results are not needed generate 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 power consumption as it introduces several new hardware structures to manage instruction execution, such as branch predictors, the renaming tables, the instruction window, and the ROB. These large structures are in constant use and contribute to increased power consumption, both dynamic and static. But, if speculation lowers the execution time by more than it increases the average power consumption, then the total energy consumed may be less.

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 are executed, it is unlikely that speculation will improve running time by a comparable amount. Figure 3.27 shows the fraction of instructions that are executed from mispeculation for a subset of the SPEC2000 benchmarks using a sophisticated branch predictor. As we can see, this fraction of executed mispeculated instructions 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, and the end of Dennard scaling makes imperfect speculation more problematic. Designers could avoid speculation, try to reduce the mispeculation, or think about new approaches, such as only speculating on branches that are known to be highly predictable.

What Limits Superscalar Processors

The hardware techniques that we discussed in this chapter are particularly effective in exploiting ILP. Virtually all medium to high-performance processors today are based on wide issue, deep pipelining, branch prediction, register renaming, and dynamic scheduling. Nevertheless, in the past decade we have observed a slowdown in the rate of improvement of superscalar processors. Superscalar features are improving in an evolutionary manner, while several key characteristics,

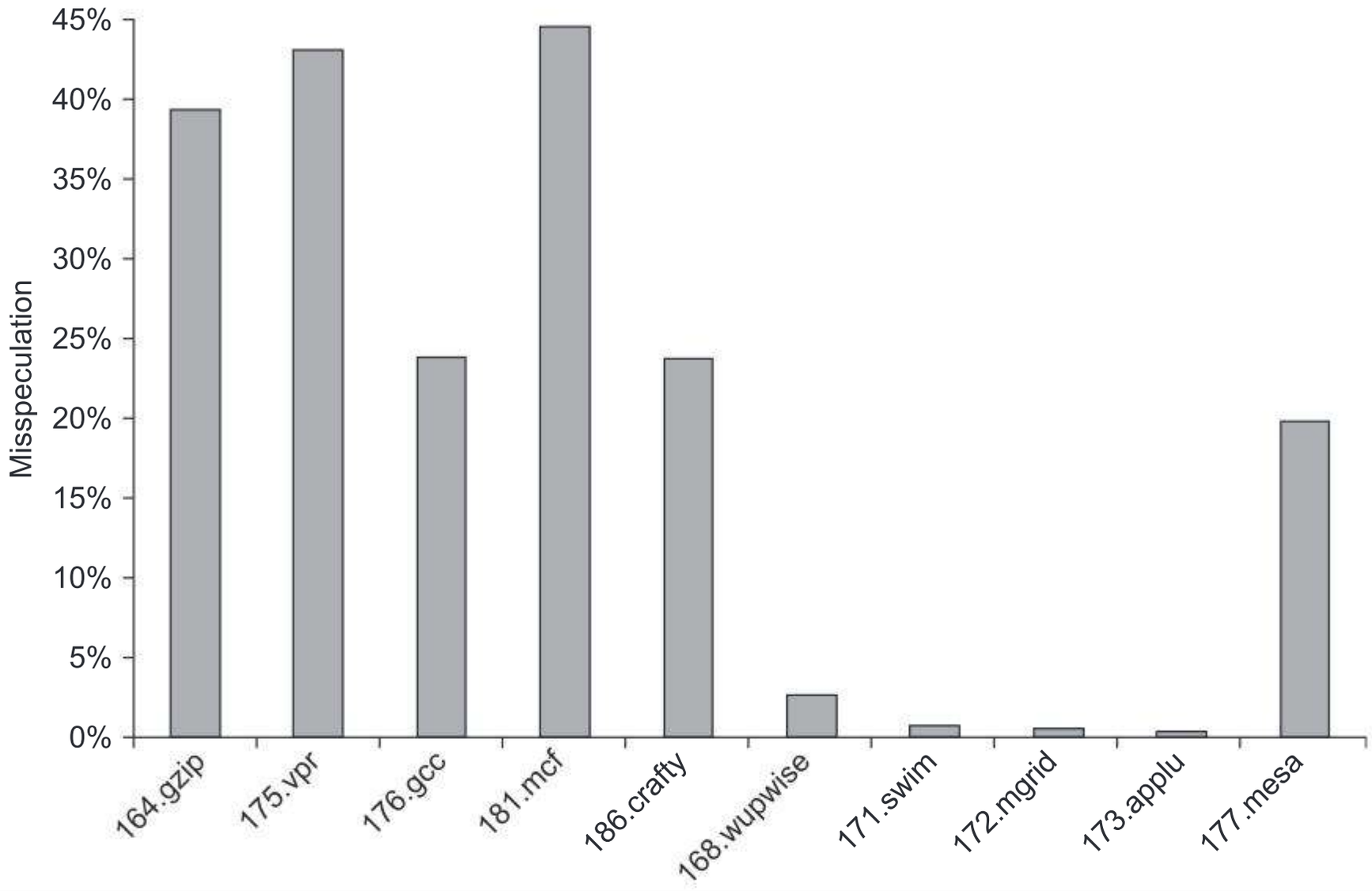


Figure 3.27 The fraction of instructions that are executed as a result of mispeculation is typically much higher for integer programs (the first five) versus FP programs (the last five).

such as issue width, pipeline depth, and even clock frequency, have been rather constant for years now. This suggests that there are limitations to the effectiveness of superscalar processors.

We have already discussed two key limitations. First, processors that are very wide (4 instructions or more per cycle) and very deeply pipelined are complex to design and power hungry. The complexity grows with the square of issue width for some stages. To alleviate that, processor vendors are now pursuing higher performance gains by combining superscalar techniques with vector and multicore techniques for data-level and thread-level parallelism, respectively. These two forms of parallelism are more regular or coarser grain than ILP and require less complex hardware to support as we will see in Chapters 4 and 5. Second, the performance of superscalar processors is gated by the accuracy of branch prediction. Even a few percentage points of misprediction rate are enough to make wider and deeper pipelines ineffective.

A third limitation is due to cache misses that are served by off-chip caches and main memory. Karkhanis and Smith provided an early analysis of the impact of long cache misses that may be pending for hundreds of clock cycles [Karkhanis & Smith 2002]. Interestingly, the dependent instructions that are stalled in the instruction window while a long miss is pending are not a primary issue. The few tens of dependent instructions fit in the large instruction windows of modern processors without blocking independent instructions from issuing when ready. In

contrast, the ROB blockage is an important issue as no younger instruction can commit while the miss is pending. This has motivated research on techniques to build superscalar processors with thousands of ROB entries using checkpointing techniques, which have not been commercially applied thus far. Finally, long misses often feed into data-dependent branches, leading to inaccurate branch prediction.

3.9 Exploiting ILP Using Multiple Issue and Static Scheduling

The preceding sections present hardware techniques to exploit ILP using multiple-issue processors. Even though programs are compiled assuming sequential execution, one instruction at the time without any overlapping, the hardware uses branch prediction, register renaming, and dynamic scheduling to discover and exploit ILP. 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.

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. Because 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.

Because 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 have seen earlier in this chapter, the growth in overhead is a major factor limiting wider issue processors.

Let's consider a VLIW processor with instructions that contain five operations, including one integer operation (which could also be a branch), two floating-point operations, and two memory references. The instruction would have a set of fields for each functional unit—perhaps 16–24 bits per unit, yielding an instruction length of between 80 and 120 bits. By comparison, the Intel Itanium 1 and 2 contain 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, because moving code across branches is expensive.

In Appendix H, we 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 RISC-V code) for such a processor. Unroll as many times as necessary to eliminate any stalls.

Answer Figure 3.28 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 for the unrolled and scheduled loop. 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 made the approach less efficient. The technical problems were the increase in code size and the limitations of the 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 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,

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
`fld f0,0(x1)`	`fld f6,-8(x1)`			
`fld f10,-16(x1)`	`fld f14,-24(x1)`			
`fld f18,-32(x1)`	`fld f22,-40(x1)`	`fadd.d f4,f0,f2`	`fadd.d f8,f6,f2`	
`fld f26,-48(x1)`		`fadd.d f12,f0,f2`	`fadd.d f16,f14,f2`	
		`fadd.d f20,f18,f2`	`fadd.d f24,f22,f2`	
`fsd f4,0(x1)`	`fsd f8,-8(x1)`	`fadd.d f28,f26,f24`		
`fsd f12,-16(x1)`	`fsd f16,-24(x1)`			`addi x1,x1,-56`
`fsd f20,24(x1)`	`fsd f24,16(x1)`			
`fsd f28,8(x1)`				`bne x1,x2,Loop`

Figure 3.28 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming correct branch prediction. 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 RISC-V would normally use in this loop. The preceding VLIW code sequence requires at least eight FP registers, whereas the same code sequence for the base RISC-V processor can use as few as two FP registers or as many as five when unrolled and scheduled.

we show other techniques, as well as document the significant code expansion seen in 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 because all the functional units had to be kept synchronized. Although a compiler might have been able to schedule the deterministic functional units to prevent stalls, predicting which data accesses would encounter a cache stall and scheduling them were very difficult to do. Thus caches needed to be blocking and causing *all* the functional units to stall. As the issue rate and number of memory references became large, this synchronization restriction became 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 general-purpose VLIWs or those that run third-party software. 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. In the domain-specific architectures, which we examine in Chapter 7,

this problem is not serious because applications are written specifically for an architectural configuration.

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

Multiple Issue in Perspective

Figure 3.29 summarizes the basic approaches to multiple issue and their distinguishing characteristics and shows processors that use each approach. Although statically scheduled superscalar processors issue a varying rather than a fixed number of instructions per clock, they are actually closer in concept to VLIWs because 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. Several low-power processors, such as the Arm A53 we describe later in this chapter, are statically scheduled superscalar processors with issue width of two. Beyond that width, most designers choose to implement either a VLIW or a dynamically scheduled and speculative superscalar. With the EPIC architecture canceled in 2020, dynamic and speculative superscalar processors are the dominant approach for server, desktop, laptop, and even cell-phone systems. As a result, the term superscalar processor is commonly used to mean a dynamically scheduled, speculative superscalar design. Several chips combine both dynamic and statically scheduled superscalar processors in what is known as the big.LITTLE arrangement, leaving it to software to decide which processors to use in order to optimize for performance or low power depending on the use case. VLIW processors are still popular in signal processing applications. We will also see them again in Chapter 7 when we discuss domain-specific accelerators.

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 the former's ability to extract some parallelism from less structured code and to easily cache all forms of data. For these reasons, multiple-issue approaches have become the primary method for taking advantage of ILP, and vectors have become primarily an extension to these processors.

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Static superscalar	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Dynamic superscalar	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (dynamic and speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Xeon and Core, AMD EPYC and Ryzen, ARM Neoverse and many Cortex-A, Apple M1
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	Mostly static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Intel Itanium

Figure 3.29 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.

3.10 Cross-Cutting Issues

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, of these approaches are listed here:

- 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. This disambiguation allows us to move loads past stores at runtime. Support for speculation around memory references can also help overcome the conservatism of the compiler, but unless speculation is selective, 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, where the misprediction rates for dynamic predictors are usually less than one-half of those for static predictors. 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 into 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 one 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 because 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 described on page 244, 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.

Raising protection exceptions for these addresses would be functionally incorrect and likely cancel the benefits from speculative execution. As we discussed in Sections 3.5 and 3.6, superscalar processors defer exceptions until instructions are ready to commit, in other words, until they are no longer speculative. But as we will see in Section 3.12, even just executing loads to invalid addresses that will be later canceled can be the source of significant security vulnerabilities.

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. Thus, these processors must be matched with nonblocking caches.

In reality, the penalty of a miss that goes to DRAM is so large that speculated misses are handled only when the next level is on-chip cache (L2 or L3). 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 a number of simultaneous memory accesses.

3.11 Multithreading: Exploiting Thread-Level Parallelism to Improve Single Core Throughput

The topic we cover in this section, multithreading, is truly a cross-cutting topic, because it has relevance to pipelining and superscalars, to GPUs (Chapter 4), and to multiprocessors (Chapter 5). A software *thread* is like a process in that it has state and a current PC, but threads typically share the address space of a single process, allowing a thread to easily access data of other threads within the same process. Multithreading is a hardware technique whereby multiple threads within a process or multiple processes share a processor pipeline without requiring an intervening process switch by the operating system. The ability to switch between

threads rapidly or to overlap instructions between threads is what enables multithreading to be used to hide pipeline and memory latencies.

In the next chapter, we will see how multithreading provides the same advantages in GPUs. Finally, Chapter 5 will explore the combination of multithreading and multiprocessing. These topics are closely interwoven because multithreading is a primary technique for exposing more parallelism to the hardware. In a strict sense, multithreading uses thread-level or process-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.

Because attempts to cover long memory stalls with more ILP have limited effectiveness and significant cost, 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 because 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 process-level 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 across multiple processor pipelines. 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 PC. As we will see in Chapter 5, many recent processors both incorporate 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 and a separate PC 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 by the operating system, which typically requires hundreds to thousands of processor cycles, or a thread switch in a user-level, cooperative threading library, which also costs tens to hundreds of clock cycles.

There are three main hardware approaches to multithreading: fine-grained, coarse-grained, and simultaneous. *Fine-grained multithreading* switches between active threads on each clock cycle, causing the execution of instructions from

multiple threads to be interleaved across pipeline stages. This interleaving is often done in a round-robin fashion, skipping any inactive 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 because 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 because a thread that is ready to execute without stalls will be delayed by instructions from other threads. It trades an increase in overall, multithreaded throughput for a loss in the performance of a single thread (as measured by latency). The processor must also be carefully designed. Since the pipeline may overlap instructions from multiple threads, any control logic forwarding or stalling must consider the thread identifier in addition to register numbers.

Fine-grain multithreading was first used commercially in the Denelcor HEP and Tera MTA supercomputers. The SPARC T1 through T5 processors (originally made by Sun, then by Oracle, and now by Fujitsu) use fine-grained multithreading. These processors were targeted at multithreaded workloads such as transaction processing and web services. The T1 supported 8 cores per processor and 4 threads per core, while the T5 supported 16 cores and 128 threads per core. Later versions (T2—T5) also supported 4—8 processor sockets. The NVIDIA GPUs, which we look at in the next chapter, also make use of fine-grained multithreading.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such cache misses that lead to off-chip memory or cache accesses. Because instructions from other threads will be issued only when a thread encounters a costly stall, coarse-grained multithreading relieves the need to have thread-switching be essentially free and is much less likely to slow down the execution of any one thread. It also simplifies the processor design as the pipeline is executing instructions by a single thread at any point in time. Instructions for other active threads may be prefetched but are not executed until the costly stall occurs.

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 processor 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). SMT 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 TLP 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. In practice, SMT weaves together the ILP across multiple threads to utilize the pipeline as well as possible.

Figure 3.30 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 SMT

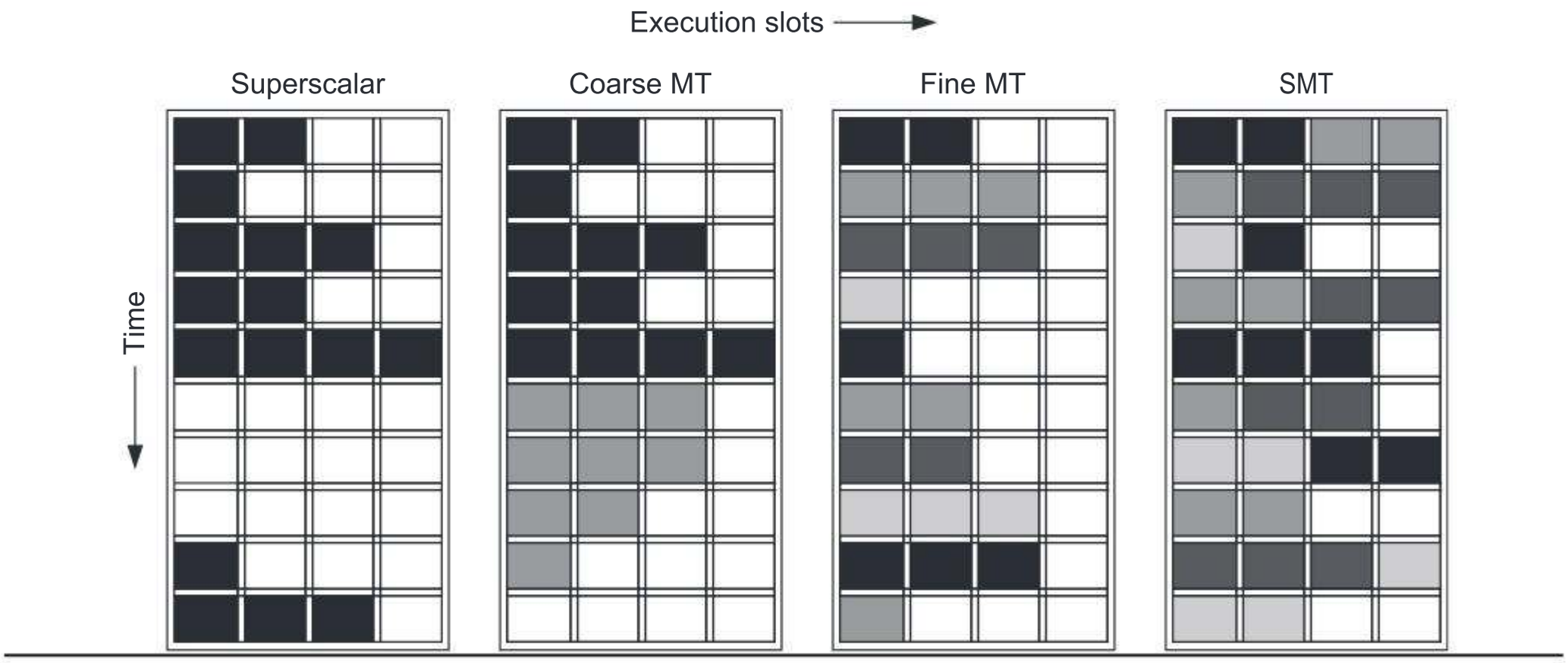


Figure 3.30 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (*white*) box indicates that 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 Power10 processors use SMT. The T2 and Power10 have 8 threads, while the Intel i7 has 2. 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 superscalar processor 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 pipeline can be frequently left idle.

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 occurs only 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 issue only 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 speculative, superscalar processor, the result is SMT. In most 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. On each cycle, the instruction scheduler picks ready instructions to execute across all threads. Although Figure 3.30 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.

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

A key design decision for an SMT processor is how to allocate pipeline resources such as the ROB, instruction scheduler, or BTB entries across threads. One approach is to statically partition resources between threads, which provides performance consistency and fairness at the expense of limiting single-thread performance and overall hardware utilization. The alternative is to dynamically share entries as needed given the ILP of each running thread, which optimizes for the opposite trade-off. SMT processors use both approaches across different blocks of their pipeline. Certain blocks, such as the RAS, are typically replicated since they are small to begin with and sharing them can have a large performance impact.

On each clock cycle, the front-end of an SMT processor must decide which thread to fetch instructions for. A particularly effective approach is to fetch instructions for the thread with the fewest instructions pending in the rest of the pipeline. This approach balances performance and fairness. If a thread exhibits high ILP for a while and its instructions execute and commit fast without stalls, this thread will be prioritized for instruction fetch. However, the same policy will also be fair to slower threads; after a few cycles of fetching instructions for the fast thread, the pending instruction count of any slower thread will drop and its fetch priority will rise.

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 5 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.

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 for x86 and Arm offer only two 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. IBM Power processors are the only ones that implement 4 to 8 SMT threads.

Esmaeilzadeh et al., (2011) did an extensive and insightful set of measurements that examined both the performance and energy benefits of using SMT in a single i7 920 core running a set of multithreaded applications. The Intel i7 920 supported SMT with two threads per core, as does the recent i7 6700. The changes between the i7 920 and the 6700 are relatively small and are unlikely to significantly change the results as shown in this section.

The benchmarks used 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 3.31. Figure 3.32 shows the ratios of performance and energy efficiency for these benchmarks when run on one core of a i7 920 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

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 3.31 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 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.

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 3.32).

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 pjbb2005 and tradebeans, the average 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 support in an aggressive speculative processor can improve performance in an energy-efficient fashion. In 2011, the balance between offering multiple simpler cores and fewer, more sophisticated cores

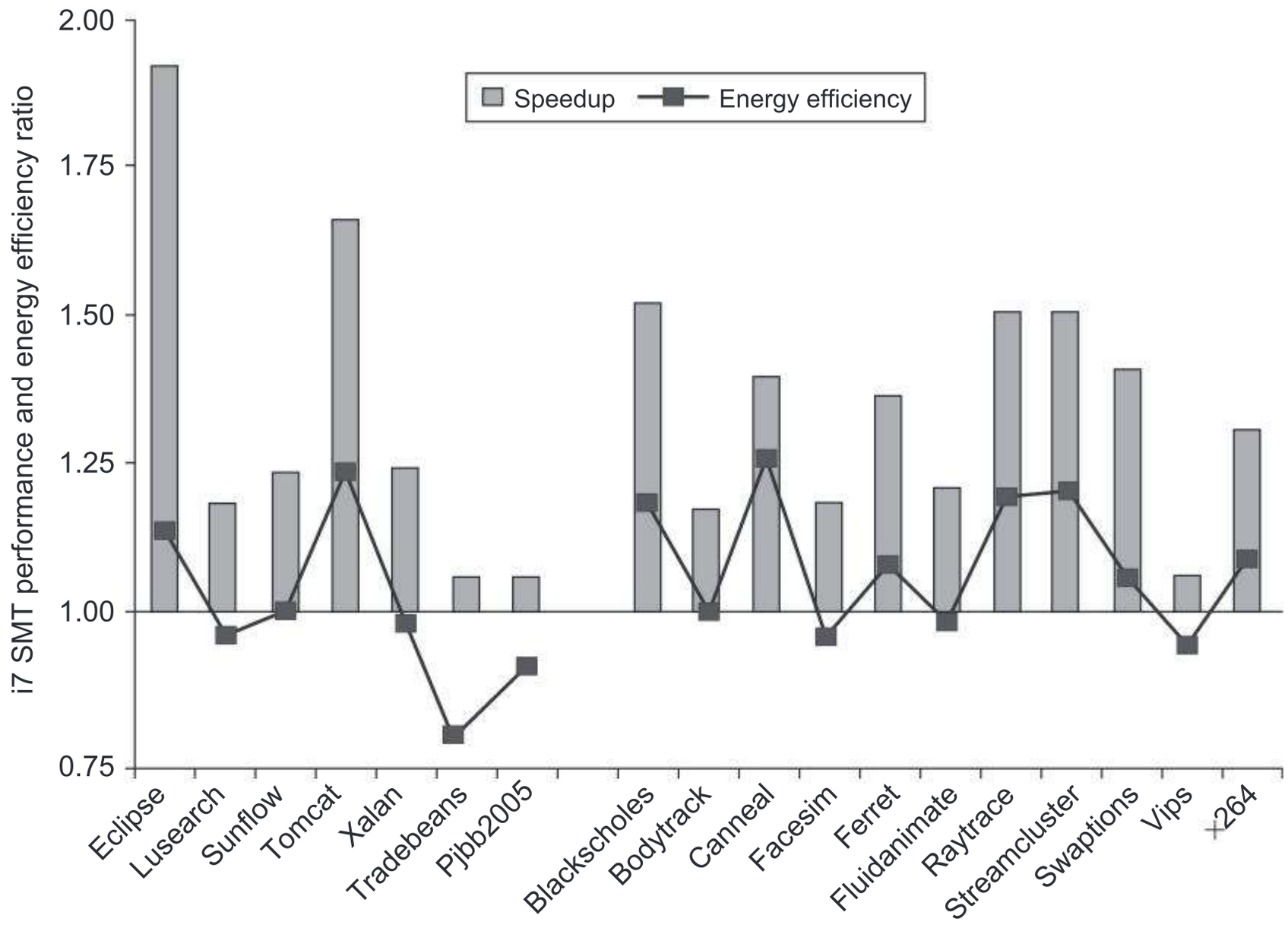


Figure 3.32 The speedup from using multithreading on one core on an i7 processor averages 1.28 for the Java benchmarks and 1.31 for the PARSEC benchmarks (using an unweighted harmonic mean, which implies a workload where the total time spent executing each benchmark in the single-threaded base set was 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 issue. Turbo Boost is off in all cases. These data were collected and analyzed by Esmaeilzadeh 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.

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) showed 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 originally designed for the netbook and PMD market, now focused on low-end PCs).

3.12 Microarchitecture Side-Channel Attacks

The speculative execution and multithreading techniques described in this chapter have enabled wide-issue processors to deliver good performance across a wide range of workloads. However, their use has also introduced a new class of

security vulnerabilities, called *microarchitecture side-channel attacks* or *transient execution attacks*. Starting with the introduction of the Meltdown [Lipp et al., 2018] and Spectre [Kocher et al., 2019] attacks, there has been a steady stream of new types and new variants of side-channel attacks that exploit performance features of superscalar processors to access and leak secret data.

Microarchitecture side-channel attacks are similar to the prime-probe attacks we saw in Chapter 2. What makes microarchitecture attacks particularly worrisome is that they allow the spy to control the execution of instruction sequences between the prime and probe phases. As we will explain shortly, speculation allows these instructions to directly access secret data, bypassing security mechanisms such as software bounds checks, virtual memory protection, virtual machine isolation, and hardware isolation for trusted execution. Hence, they can quickly and accurately generate a mark in a cache to be observed during probing. To make matters worse, an SMT processor allows the spy process to share the processor pipeline with the victim process. This creates two problems. First, the spy can use multiple caching structures to establish a high bandwidth side channel, including first-level caches or the BTB. Second, since the spy runs in parallel with the victim, the spy can quickly probe the side channel for the leaked information before it is overwritten by regular execution activity (e.g., cache refills). The combination of speculation and SMTs allows for side-channel attacks that can potentially surpass all previously known attacks in accuracy and effective bandwidth. With numerous services running on shared infrastructure such as public cloud frameworks that use superscalar and SMT processors, the security implications of microarchitecture side-channel attacks can be severe.

We will now look at how speculation assists with side-channel attacks using the following pseudocode that summarizes the original Meltdown attack [Lipp et al., 2018]. A spy that runs this code on a shared core can potentially observe the value of any memory address used by the operating system kernel.

```
---------------------------------
Allocate user_memory[];              // prime phase
Flush_caches();
Generate_exception();                // speculate and leak
Value = kernel_array[Address];
Index = (Value & 1) * 256;           // probe phase
Value = user_array[Index];
---------------------------------
```

The spy primes the system by (1) allocating the array `user_memory` and (2) ensuring that none of its elements are cached in the processor's first-level cache. Next, (3) it issues an instruction that will generate an exception, for example, an arithmetic exception (divide by zero) or memory protection error through a load instruction. If the processor executed one instruction at the time, none of the following steps (4–6) would ever execute. However, a speculative processor will continue executing the following code until the instruction generating the exception reaches the head of the ROB. Hence, it is very likely that all following instructions will fully or partially execute before the exception is raised and the

results of the instructions in steps 4 to 6 are discarded. The next instruction (4) accesses a memory location in the kernel that it should not have access to, for example, by adding or subtracting a large offset from a pointer in the spy's memory address space. Instruction (5) generates an index value, 0 or 256, based on the least significant bit of the kernel location read and instruction (6) uses the index to read an element of the `user_memory` array. While instructions (4–6) will be canceled when the exception reaches the head of the ROB, the impact of instruction (6) on the processor's first-level cache will remain. A portion of array `user_memory` is now cached. After recovering from the exception or running in parallel on another thread on the same SMT processor, the spy can probe the cache by timing the access to elements of array `user_memory`. If element 0 loads fast, indicating a cache hit, the least significant bit of the kernel memory location is 0. If element 256 loads fast, the least significant bit of the kernel memory location is 1. The spy can repeat these steps to gradually read big portions of the kernel memory and all the secrets it contains, bypassing virtual memory protection. The Meltdown paper explains each step in detail and illustrates implementations for the x86 instruction set [Lipp et al., 2018].

The structure of most microarchitecture side-channel attacks is similar to that of Meltdown. Between priming and probing, the spy exploits speculation to execute (but not commit) instructions that access and leak secret data. The secret data is used to modify the state of a hardware performance mechanism that will be soon probed to infer some bits of the secret data. Recently developed attacks have used a wide range of performance mechanisms (caches, BTBs, etc.) or speculation events (delayed exception reporting, branch prediction, memory disambiguation) in superscalar processors. In many attacks, including Spectre, the spy succeeds by calling into the operating system or the victim process through available software interfaces with carefully selected inputs that cause the victim's own code to speculatively access and leak the secret data. Using an SMT thread on the same core, the spy process probes for the leaked data while its call is served by the victim process. We refer readers to a survey paper by Guangyuan Hu et al., for a detailed review and taxonomy of microarchitecture side-channel attacks [Hu et al., 2021].

Microarchitecture side-channel attacks are difficult to defend against without incurring performance loss. A first defense option is to recompile and modify the code that potentially operates on secret data. The compiler can introduce fence instructions that disallow speculative execution until a condition has been resolved. Alternatively, the compiler can introduce a dependency between an instruction that exploits speculation and a load that potentially leaks secret data, a technique known as speculative load hardening. The disadvantage of this defense approach is that it relies on recompilation, which is not always possible, and introduces a performance-security trade-off when choosing how liberally to apply these defenses on potentially vulnerable code. A second option is to limit access to shared hardware structures that can be used to communicate secrets. In some cases, SMT multithreading has been turned off to disallow a spy process to easily observe the wide range of shared structures on an SMT processor that also runs

the victim process. Cache partitioning technology that isolates the cache space used by different processes can also reduce the effectiveness of some attacks. Finally, there are several proposals to modify processor pipelines to limit access or delay the use of potentially sensitive data under speculation, or to undo the impact of mispeculated instructions on hardware performance mechanisms. The survey by Guangyuan Hu et al., also reviews the main categories of defenses for microarchitecture side-channel attacks [Hu et al., 2021]. At this point, there is no set of defenses that are guaranteed to work against all possible attacks.

Microarchitecture side-channel attacks have raised awareness that existing ISAs do not have sufficient specification of correct behaviors in order to reason about security attacks. ISAs define the functional behavior of programmable processors in a timing-independent manner. Hence, it is possible to build superscalar processors that correctly implement the ISA but are vulnerable to security attacks. This weakness is the case with speculative superscalar processors today. Our field is missing the abstractions and tools to model and formally analyze how the timing behavior of hardware implementations interacts with correct execution with respect to security and privacy. At the time of writing, there are several research efforts to address this challenge.

3.13 Putting It All Together: The Arm Cortex-A53 and the Intel Golden Cove Processors

In this section, we explore the design of two multiple issue processors. First, we discuss the Arm Cortex-A53 core, which is often used as the high-efficiency core in chips for cellphones and tablets that pack several high-performance (big) and high-efficiency (little) cores. Next, we discuss the Intel Golden Cove core, which is the high-performance (p-core) in the Alder Lake line of client CPUs and the Sapphire Rapids line of server CPUs.

The Arm Cortex-A53

The A53 is a dual-issue, statically scheduled superscalar with dynamic issue, which allows the processor to issue two instructions per clock. Figure 3.33 shows the basic pipeline structure of the pipeline. For nonbranch, integer instructions, there are eight stages: F1, F2, D1, D2, D3/ISS, EX1, EX2, and WB, as described in the caption. The pipeline is in order, so an instruction can be issued for execution only when its results are available and when proceeding instructions have initiated execution as well. Thus, if the next two instructions are dependent, both can proceed to the appropriate execution pipeline, but they will be serialized when they get to the beginning of that pipeline. When the scoreboard-based issue logic indicates that the result from the first instruction is available, the second instruction can issue.

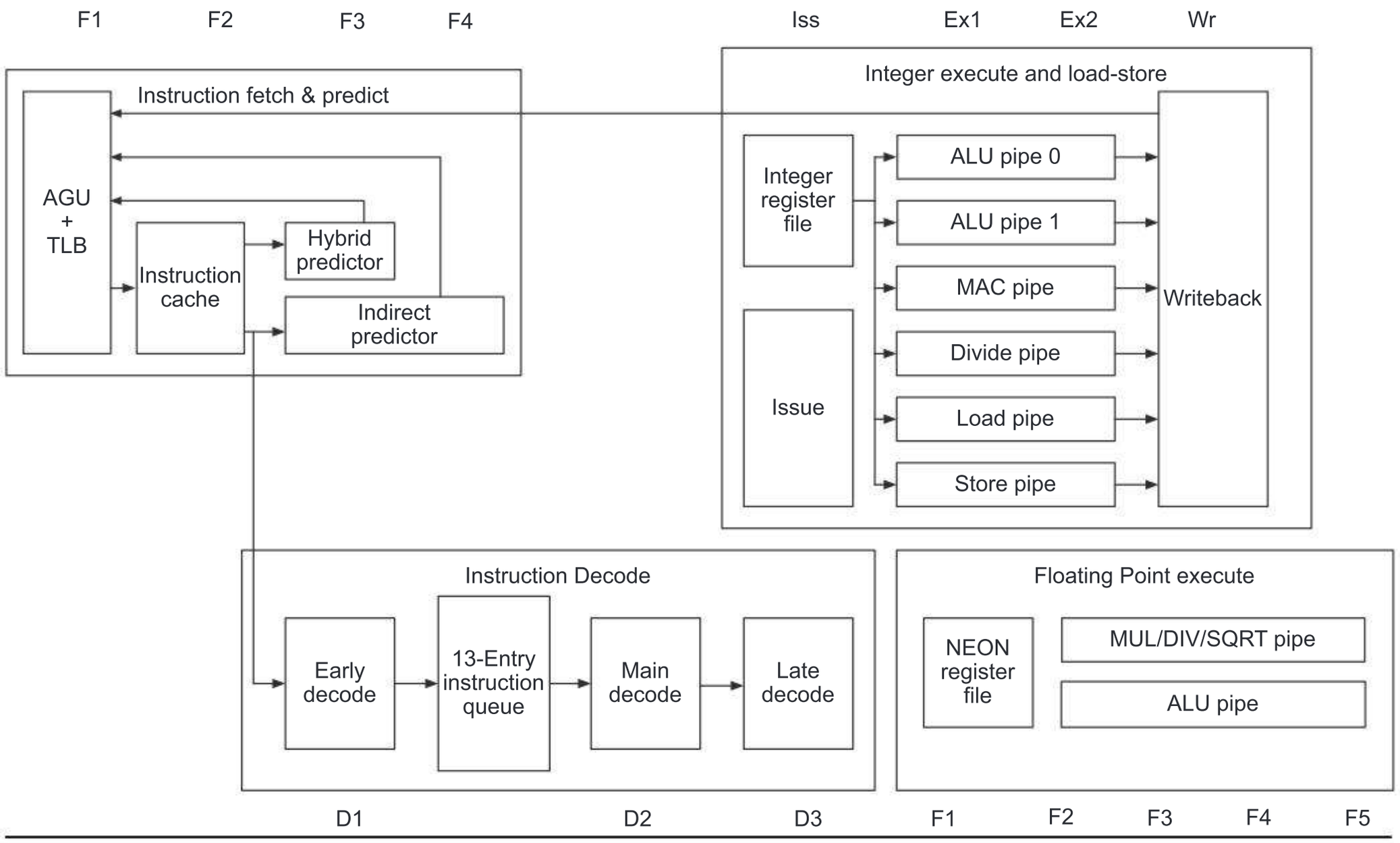


Figure 3.33 The basic structure of the A53 integer pipeline is 8 stages: F1 and F2 fetch the instruction, D1 and D2 do the basic decoding, and D3 decodes some more complex instructions and is overlapped with the first stage of the execution pipeline (ISS). After ISS, the Ex1, EX2, and WB stages complete the integer pipeline. Branches use four different predictors, depending on the type. The floating-point execution pipeline is 5 cycles deep, in addition to the 5 cycles needed for fetch and decode, yielding 10 stages in total.

The four cycles of instruction fetch include an address generation unit that produces the next PC either by incrementing the last PC or from one of four predictors:

1. A single-entry branch target cache containing two instruction cache fetches (the next two instructions following the branch, assuming the prediction is correct). This target cache is checked during the first fetch cycle. If it hits, then the next two instructions are supplied from the target cache. In case of a hit and a correct prediction the branch is executed with no delay cycles.
2. A 3072-entry hybrid predictor, used for all instructions that do not hit in the branch target cache, and operating during F3. Branches handled by this predictor incur a 2-cycle delay.

3. A 256-entry indirect branch predictor that operates during F4; branches predicted by this predictor incur a 3-cycle delay when predicted correctly.
4. An 8-deep return stack, operating during F4 and incurring a three-cycle delay.

Branch decisions are made in ALU pipe 0, resulting in a branch misprediction penalty of 8 cycles. Figure 3.34 shows the misprediction rate for SPECint2006. The amount of work that is wasted depends on both the misprediction rate and the issue rate sustained during the time that the mispredicted branch was followed. As Figure 3.35 shows, wasted work generally follows the misprediction rate, though it may be larger or occasionally shorter.

Performance of the A53 Pipeline

The A53 has an ideal CPI of 0.5 because of its dual-issue structure. Pipeline stalls can arise from three sources:

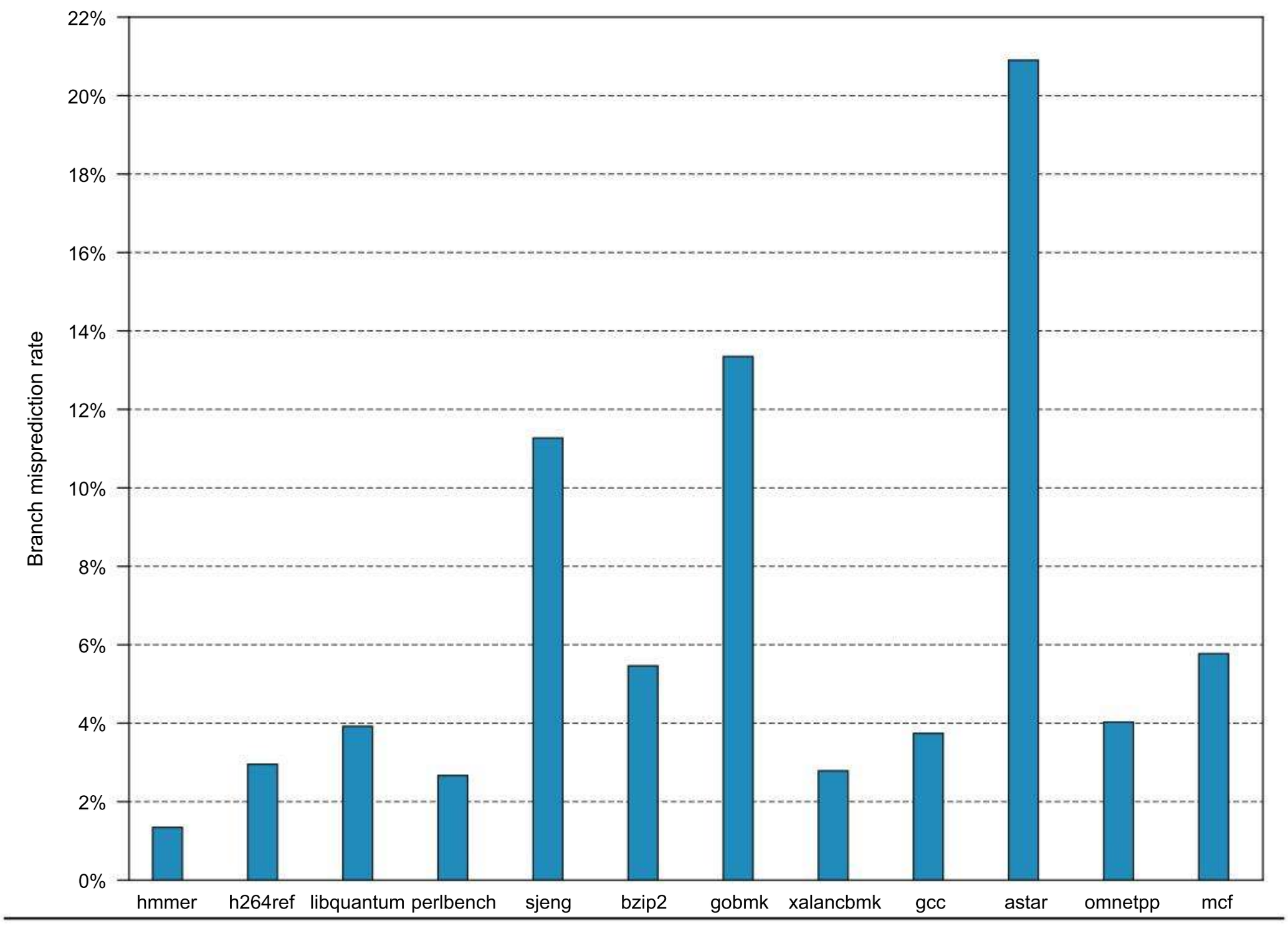


Figure 3.34 Misprediction rate of the A53 branch predictor for SPECint2006.

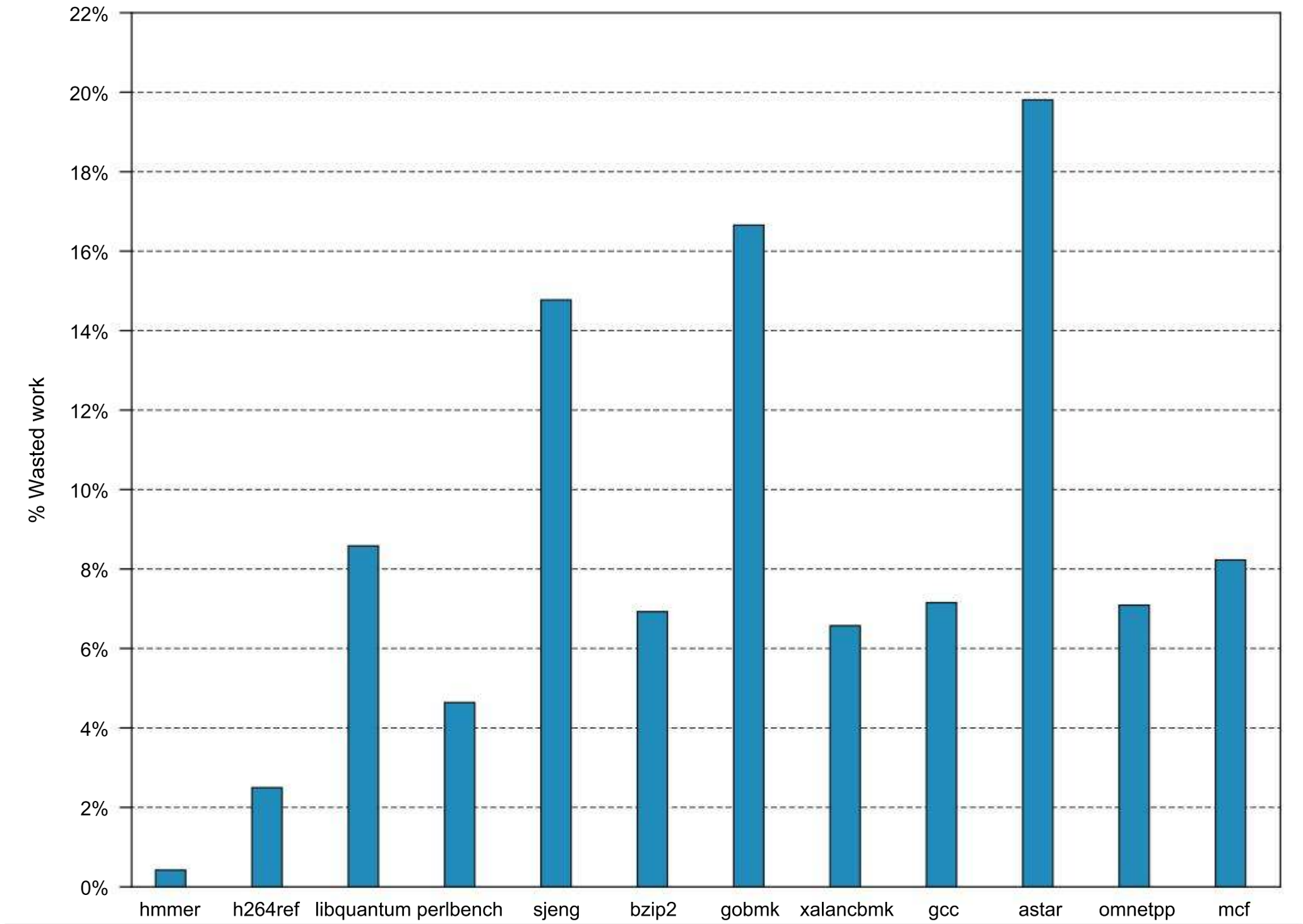


Figure 3.35 Wasted work due to branch misprediction on the A53. Because the A53 is an in-order machine, the amount of wasted work depends on a variety of factors, including data dependences and cache misses, both of which will cause a stall.

1. Functional hazards, which occur because two adjacent instructions selected for issue simultaneously use the same functional pipeline. Because the A53 is statically scheduled, the compiler should try to avoid such conflicts. When such instructions appear sequentially, they will be serialized at the beginning of the execution pipeline, when only the first instruction will begin execution.
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. Again, the compiler should try to prevent such stalls when possible.
3. Control hazards, which arise only when branches are mispredicted.

Both TLB misses and cache misses also cause stalls. On the instruction side, a TLB or cache miss causes a delay in filling the instruction queue, likely leading to a downstream stall of the pipeline. Of course, this depends on whether it is an L1 miss, which might be largely hidden if the instruction queue was full at the time of the miss, or an L2 miss, which takes considerably longer. On the data side, a cache or TLB miss will cause the pipeline to stall because the load or store that caused the miss cannot proceed down the pipeline. All other subsequent instructions will thus be stalled. Figure 3.36 shows the CPI and the estimated contributions from various sources.

The A53 uses a shallow pipeline and a reasonably aggressive branch predictor, leading to modest pipeline losses while allowing the processor to achieve high clock rates at modest power consumption. In comparison with the i7, the A53 consumes approximately 1/200 the power for a quad-core processor! Overall, the A53 is a very successful low-power, superscalar processor that is often combined with higher-performance Arm cores in big.LITTLE arrangements.

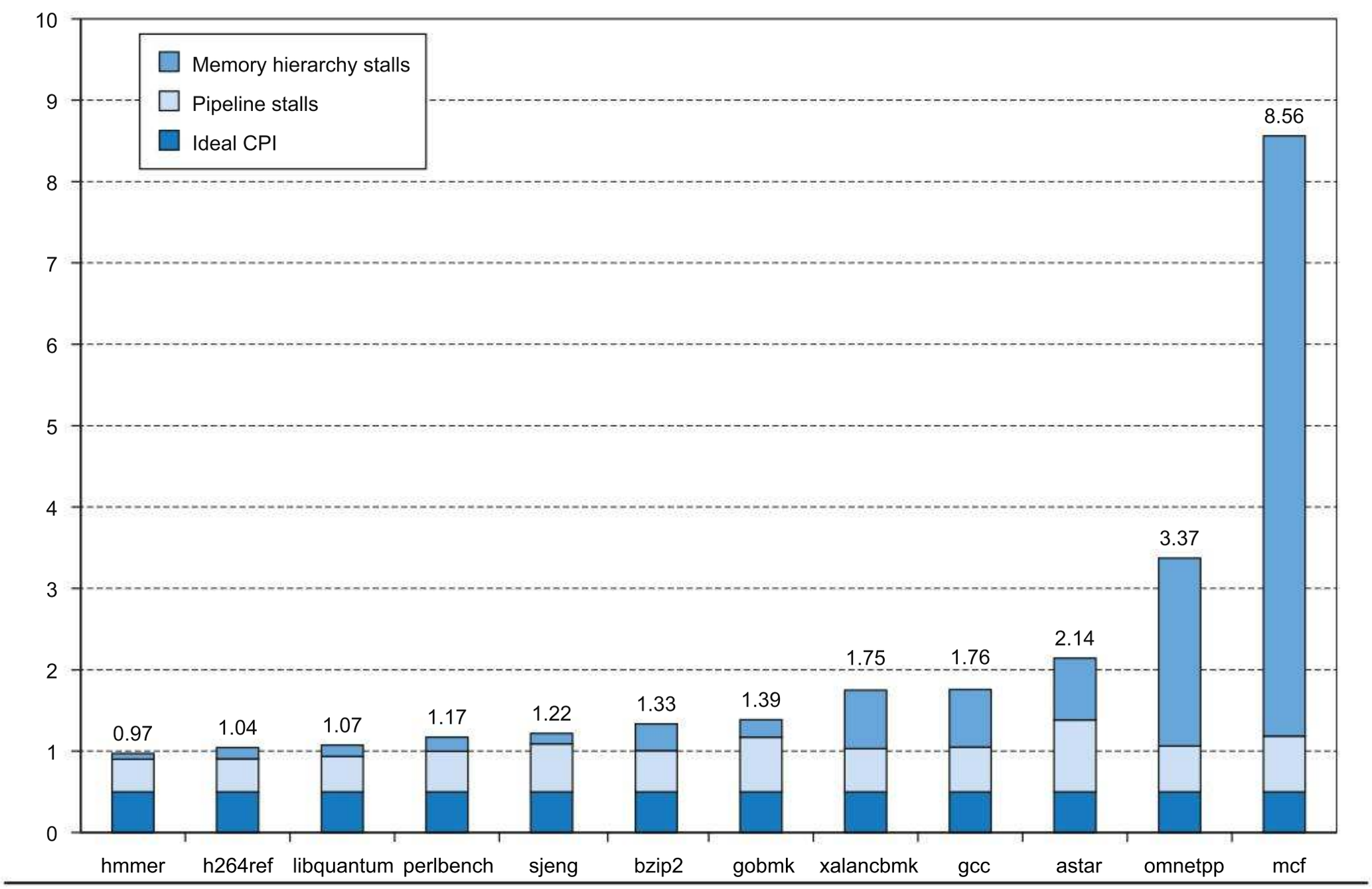


Figure 3.36 The estimated composition of the CPI on the Arm A53 shows that pipeline stalls are significant but are outweighed by cache misses in the poorest performing programs. This estimate is obtained by using the L1 and L2 miss rates and penalties to compute the L1 and L2 generated stalls per instruction. These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards.

The Intel Golden Cove

Golden Cove is an aggressive OOO, speculative microarchitecture that aims to increase the instruction level parallelism a core can extract for a single thread (Rotem et al., 2022). Golden Cove cores also support SMT execution of 2 threads, as has been the case for all high-end Intel microarchitectures since 2008. Golden Cove was released in 2021 and succeeded the Sunny Cove microarchitecture. Figure 3.37 shows the overall structure of the Golden Cove pipeline and the size of many of its key structures.

Instruction fetch: Instruction delivery is particularly complex for x86 processors as the length of instructions ranges from 1 to 15 bytes. The Golden Cove front-end fetches 32 bytes per cycle from the instruction cache and uses 6 decoders to translate x86 instructions into RISC-like *micro-ops* that are buffered in the 144-entry *micro-op queue*. 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. 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 front-end includes a *micro-op cache* that stores 4K decoded micro-ops and can deliver up to 8 micro-ops per cycle to the micro-op queue. The micro-op cache allows instruction delivery without using the instruction cache and decoders. Moreover, there is a *loop stream detector (LSD)* that tracks loops that fit in the micro-op queue. The LSD allows the micro-op queue to provide the back-end with instructions until a branch misprediction inevitably ends the loop stream. In single-thread mode, the LSD saves power by allowing the rest of the front-end to sleep. In SMT mode, when one thread utilizes the LSD, more front-end resources can be used to speed up instruction delivery for the other thread.

To better support workloads with a large code footprint, Golden Cove uses an instruction TLB that is twice as large as that of Sunny Cove, with 256 entries for 4 KB pages and 32 entries for 2 MB/4 MB pages. Code prefetching was also improved. While the organization of the branch predictors is secret, we know that several structures were enlarged. The second-level branch target buffer has 12K entries, compared to 5K entries in Ice Lake.

Out-of-order engine: Golden Cove uses a large ROB with 512 entries to support OOO execution. Up to 6 micro-ops are renamed each cycle and issued to schedulers that orchestrate execution on the various functional units. The hardware detects certain x86 instruction idioms that can be optimized. For example, instructions that effectively zero registers or move values between registers do not need to be scheduled to a functional unit.

The scheduler for arithmetic micro-ops has 97 entries used for both integer and floating-point operations. Using its 5 ports, the scheduler can issue up to 5 ready micro-ops per cycle to integer functional units and up to 3 micro-ops per cycle to floating-point and vector functional units. Golden Cove implements

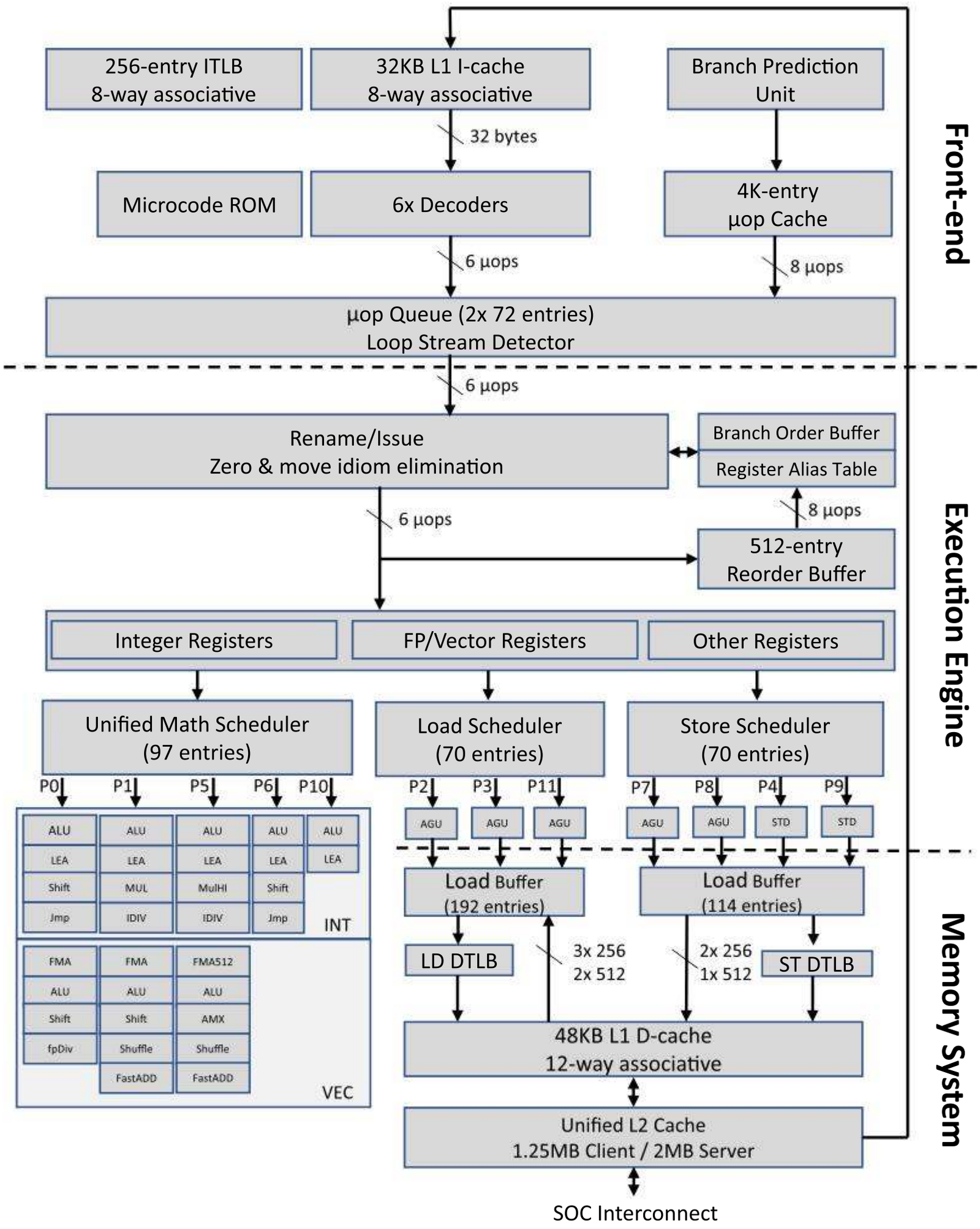


Figure 3.37 The block diagram of the Intel Gold Cove microarchitecture. The out-of-order core can issue up to 6 micro-ops per cycle, can dispatch up to 12 micro-ops per cycle, and can retire up to 8 micro-ops per cycle.

several x86 ISA extensions for data center and HPC workloads, including 512 SIMD instructions (AVX-512) with support for datatypes used in machine-learning workloads such as the Brain Floating-point (BF16) format and 8-bit integers (INT8). We discuss the significance of narrow datatypes for machine learning in Chapter 7. Golden Cove is the first Intel microarchitecture to implement *advanced matrix extensions (AMX)* for machine-learning workloads.

AMX defines a new 2D register file with 8 registers, which are arrays of 16 64-byte rows, and a set of matrix multiplication instructions that operate on these registers using different 2D geometries at a maximum rate of 2K Int8 operations per cycle. The 70-entry scheduler for loads can feed 3 address-generation units per cycle. The 38-entry scheduler for stores can feed 2 address generation units and 2 store data units each cycle.

Golden Cove recovers from branch mispredictions using a 128-entry *branch order buffer* (Boggs et al., 2004). This buffer takes snapshots of the register renaming state on conditional and indirect branch instructions that may be mispredicted. As soon as a branch misprediction is detected, all younger instructions are discarded and the last snapshot prior to the branch is restored before instruction fetching resumes from the correct target. Older instructions continue executing during the misprediction recovery. There is no need to wait for the mispredicted instruction to reach the head of the ROB and to flush the whole pipeline.

Memory subsystem: Golden Cove can handle up to three 32-byte loads or up to two 64-byte loads per cycle, as well as up to two 64-byte stores per cycle. The pipeline includes a load buffer with 192 entries and a store buffer with 114 entries and uses speculative memory disambiguation to expose parallelism between load and store operations. Golden Cove supports store forwarding, including for cases where the first load bytes are forwarded from a store, and the rest are read from the data cache.

The 48 KB L1 data cache is 12-way associative and its load-to-use latency for hits is 5 cycles. A *fill buffer* can track up to 16 outstanding L1 cache misses. The L2 cache is sized at 1.25 MB (client chips) or 2 MB (server chips) and delivers 64 bytes of data per cycle at a latency of 15 cycles. Golden Cove includes improved prefetchers both at the L1 and the L2 cache levels. The L2 cache includes a pattern-based multipath prefetcher with feedback-based adaptive prefetch throttling. The L2 cache can have 48 outstanding misses or prefetch requests.

Address translation for load operations uses a TLB with 96 entries for 4 KB pages (6-way associative), 32 entries for 2 MB/4 MB pages (4-way associative), and 8 entries for 1 GB pages. For store operations, there is a 16-entry TLB that serves all page sizes. All first-level TLBs for instructions and data are backed by a second-level TLB (STLB) with 2K entries. STLB misses are sent to the page miss handler (PMH) that can perform up to 4 page-table walks in parallel.

Comparison to Previous Intel Microarchitectures: Figure 3.38 compares the key characteristics of Golden Cove to those of the Skylake (2015) and the Sunny Cove (2019) microarchitectures. Golden Cove represents significant improvement in ILP capabilities as it can process more instructions per cycle in the decode, issue, dispatch, and retire stages and all key buffer structures are larger compared to its predecessors. At the same time, Golden Cove core can reach high frequencies with client and server parts announced at up to 5.5Ghz and 4.2Ghz, respectively.

	Skylake (2015)	Sunny Cove (2019)	Golden Cove (201)
Micro-op queue (per SMT thread)	64	70	72
Micro-op cache size	1,500	2,304	4,096
Reservation stations (Int + FP)	97	97	80
Integer registers	180	280	280
FP registers	168	224	332
Outstanding load buffer	72	128	192
Outstanding store buffer	56	72	114
Reorder buffer	256	352	512
Instruction bytes fetched per cycle	16	16	32
Number of instructrion decoders	1 complex + 5 simple	4	6
Instructions issued per cycle	4	5	6
Scheduler ports for functional units	8	10	12
Instructions retired per cycle	4	4	8

Figure 3.38 The characteristics of three generations of out-of-order microarchitectures by Intel. The width of key pipeline stages (decode, issue, dispatch, retire) and the capacity of key buffers, queues, and register files have been steadily increasing over the years in order to extract higher amounts of ILP. The choices of the size of various buffers, while appearing sometimes arbitrary, are likely based on extensive simulation.

Performance of the Golden Cove Core

Figure 3.39 shows the performance characteristics of the Golden Cove core running the 10 SPECPUint2017 benchmarks in single-threaded mode. The CPI varies between 0.32 and 0.89. In other words, the core executes on average 1.11 to 3.14 instructions per cycle. The CPI correlates well with the misprediction rates, cache miss rates, and TLB miss rates. For example, the `mcf` benchmark is an implementation of the network simplex method for minimum cost flow problems such as vehicle scheduling in public transportation. This method replaces the linear algebra of the general simplex algorithm with network operations such as finding cycles or modifying spanning trees that can be performed very quickly. Hence the benchmark includes data-dependent branches and unpredictable pointer accesses, which leads to high branch misprediction and cache miss rates across the memory hierarchy. A high branch misprediction rate alone can starve the core from useful instructions to execute, as is the case for `leela`, an engine for the game Go that uses Monte Carlo–based position estimation. For benchmarks such as `perlbench`, a Perl script for spam filtering; `x264`, H.264/MPEG-4 AVC video encoding; and `exchange2`, a recursive program for Sudoku puzzle development, the Golden Cove predictor and caches perform well and that allows the core back-end to exploit a significant amount of ILP.

Figure 3.40 shows the performance benefit from running the 10 SPECCPUint2017 benchmarks in dual-threaded mode. Compared to running a single thread, a Golden Cove core running two SMT threads achieves 5% to 46% higher throughput for these benchmarks. The SMT benefit is typically higher for benchmarks that achieve low single-thread performance, such as `mcf` (46%) and `leela` (45%). These benchmarks underutilize the core resources and allow a second SMT

	perlbench_r	gcc_r	mcf_r	omnetpp_r	xalancbmk_r	x264_r	deepsjeng_r	leela_r	exchange2_r	xz_r
SpecRate	6.68	7.84	7.12	6.31	7.9	16.43	5.5	4.97	17.53	3.92
CPI	0.328	0.577	0.899	0.717	0.505	0.321	0.512	0.666	0.318	0.730
L1D miss rate (%)	1.33%	7.81%	14.85%	8.90%	14.20%	2.05%	1.29%	1.21%	0.03%	5.02%
L2 miss rate (%)	14.29%	39.58%	32.13%	46.19%	24.50%	11.44%	15.45%	20.65%	40.63%	23.51%
L3 miss rate (%)	8.89%	23.67%	59.06%	23.01%	2.05%	31.62%	66.67%	1.43%	0.00%	24.03%
L1D MPKI	3.88	22.08	38.25	29.33	38.2	4.33	3.04	3.19	0.1	12.51
L2 MPKI	1.35	11.83	12.36	13.95	9.77	1.36	1.38	0.7	0.13	3.08
L3 MPKI	0.12	2.8	7.3	3.21	0.2	0.43	0.92	0.01	0	0.74
Data TLB MPKI	0.1	0.45	1.56	3.36	0.11	0.04	0.23	0.01	0	1.43
Branch misprediction rate (%)	0.49%	1.30%	8.77%	2.06%	0.28%	1.46%	4.04%	9.44%	1.81%	3.95%

Figure 3.39 The key performance characteristics of a Golden Cove core running the SPECCPUint2017 benchmarks in single-threaded mode. The measurements were made on a 2-socket Sapphire Rapids system with 56 cores per socket and 8 DDR5-4800 channels per socket. The benchmark binaries were compiled with Intel ICC 18 and used AVX2 instructions. MPKI refers to misses per 1K instructions.

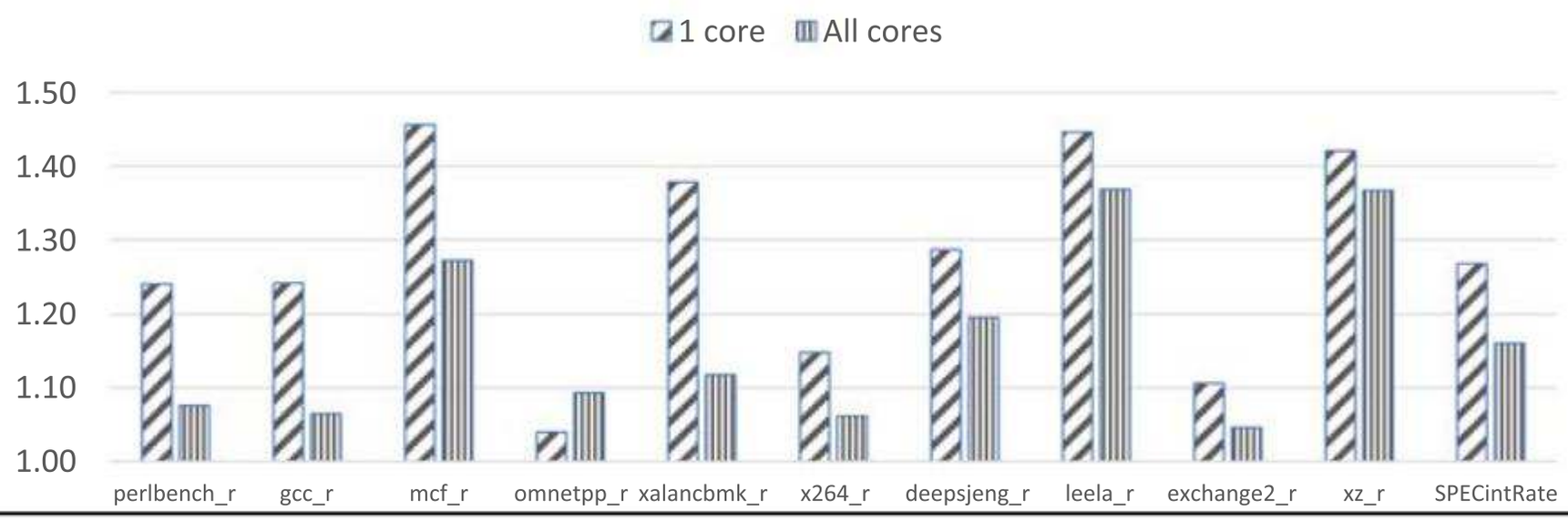


Figure 3.40 The performance benefit from running 2 SMT threads per core on a Sapphire Rapids system running the SPECCPUint2017 benchmarks. The benchmark setup is the same as the one described in Figure 3.39. We show the SMT benefit when a single core is used (up to 2 SMT threads) and when all 112 cores in the 2-socket system are used (up to 224 SMT threads).

thread to execute at a reasonable throughput. When all 112 cores in a 2-socket system run in dual-threaded mode, the SMT benefit ranges from 5% to 37%. For example, the SMT benefit for the memory-intensive `mcf` benchmark drops from 46% with 1 core active to 27% with all cores active, as the 112 threads within each socket compete for resources in the shared L3 cache and the main memory system. Overall, the SMT benefit for the geometric mean of the SPECPUint2017 rates is 27% for a single core and 16% for a 2-socket system. The significant throughput improvements from SMT justify the wider OOO pipeline even though it does not benefit equally all workloads equally in single-threaded mode.

Figure 3.41 shows the per-core performance characteristics when all 112 cores in a 2-socket system run in SMT mode. The differences in misprediction rates and L1 and L2 miss rates between Figures 3.38 and 3.40 are due to competition for

	perlbench_r	gcc_r	mcf_r	omnetpp_r	xalancbmk_r	x264_r	deepsjeng_r	leela_r	exchange2_r	xz_r
CPI (per thread)	0.590	1.260	1.420	1.950	0.958	0.556	0.794	0.934	0.552	1.150
L1D miss rate (%)	1.85%	9.10%	18.98%	10.43%	17.03%	2.96%	2.68%	1.92%	0.00%	7.37%
L2 miss rate (%)	11.32%	42.70%	25.63%	43.99%	19.01%	6.63%	12.17%	10.36%	0.00%	23.12%
L3 miss rate (%)	82.96%	65.62%	78.92%	80.76%	63.95%	77.78%	81.21%	35.19%	0.00%	71.43%
L1D MPKI	5.53	26.74	48.47	35.1	47.14	6.32	6.39	5.16	0.01	17.82
L2 MPKI	1.35	15.27	12.43	15.85	9.32	0.9	1.49	0.54	0	4.13
L3 MPKI	1.12	10.02	9.81	12.8	5.96	0.7	1.21	0.19	0	2.95
Data TLB MPKI	0.15	0.6	1.32	4.53	0.11	0.01	0.24	0	0	1.78
Branch misprediction rate (%)	0.63%	1.11%	9.04%	1.97%	0.29%	1.38%	4.22%	9.86%	1.91%	4.03%

Figure 3.41 The key performance characteristics of a single Golden Cove core, when a 2-socket system with a total of 112 cores runs the SPECCPUint2017 benchmarks in dual-threaded mode. The benchmark setup is the same as the one described in Figure 3.39. We quote the per SMT thread CPI. MPKI refers to misses per 1K instructions.

core resources. The differences in L3 miss rates are due to competition for resources across all threads in the two sockets. For example, the L3 cache misses per 1K instructions (MPKI) for *gcc* grows from 2.8 with 1 thread (Figure 3.38) to 20.2 with 224 threads (Figure 3.40). This increase highlights the importance of cache and memory system design for multicore chips with hundreds of concurrently executing threads. Also note that the per-thread CPI increases when SMT is turned on. For example, the CPI for `perlbench` increases from 0.328 to 0.590 from Figures 3.38–3.40. However, there are now two threads executing instructions at this rate, so the effective throughput gain at the core level is (2/0.590)/(1/0.328) = 1.111 or 11.1%.

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

In 2008, Intel offered a processor for the low-end Netbook and personal mobile device (PMD) space called the Atom 230, which implements both the 64-bit and 32-bit versions of the x86 architecture. The Atom is a statically scheduled, 2-issue superscalar, quite similar in its microarchitecture to the Arm A8, a single-core predecessor of the A53. In 2008 Intel also offered the Core i7 920, based on the OOO Nehalem microarchitecture. Interestingly, both the Atom 230 and the Core i7 920 have been fabricated using the same 45 nm Intel technology. Figure 3.42 summarizes the Intel Core i7 920, the Arm Cortex-A8, and the Intel

Area	Specific characteristic	Intel i7 920 Four cores, each with FP	ARM A8 One core, no FP	Intel Atom 230 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
	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 KiB/32 KiB 256 KiB 2–8 MiB	Two-level 16/16 or 32/32 KiB 128 KiB–1 MiB	Two-level 32/24 KiB 512 KiB
Memory system	Peak memory BW	17 GB/s	12 GB/sec	8 GB/s
	Peak issue rate	4 ops/clock with fusion	2 ops/clock	2 ops/clock
	Pipe line scheduling	Speculating out of order	In-order dynamic issue	In-order dynamic issue
Pipeline structure	Branch prediction	Two-level	Two-level 512-entry BTB 4 K global history 8-entry return stack	Two-level

Figure 3.42 An overview of the four-core Intel i7 920, an example of a typical Arm A8 processor chip (with 256 MiB L2, 32 KiB L1s, and no floating point), and the Intel Atom 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 higher in performance than the one-core A8 or Atom. All these processors are implemented in a comparable 45 nm technology.

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 micro-ops, 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 capability was added to later OOO microarchitectures by the use of macrofusion. This optimization means that on average for a typical instruction mix, only 4% of the instructions require more than one microoperation. The micro-ops 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 KiB instruction cache and a 24 KiB data cache, both backed by a shared 512 KiB L2 on the same die. (The Atom 230 also supports multithreading with two threads, but we will consider only single-threaded comparisons.)

We might expect that the Core i7 and Atom 230, implemented in the same technology and with the same instruction set, would exhibit predictable behavior, in 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 set 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 3.43, the i7 significantly outperforms the Atom. All benchmarks are at least 4 times faster on the i7, two SPECFP benchmarks are over 10 times faster, and one SPECINT benchmark runs over 8 times faster! Because 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 920: 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 920 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 an 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 processor 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 that are all designed for high-end applications. Figure 3.44 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.

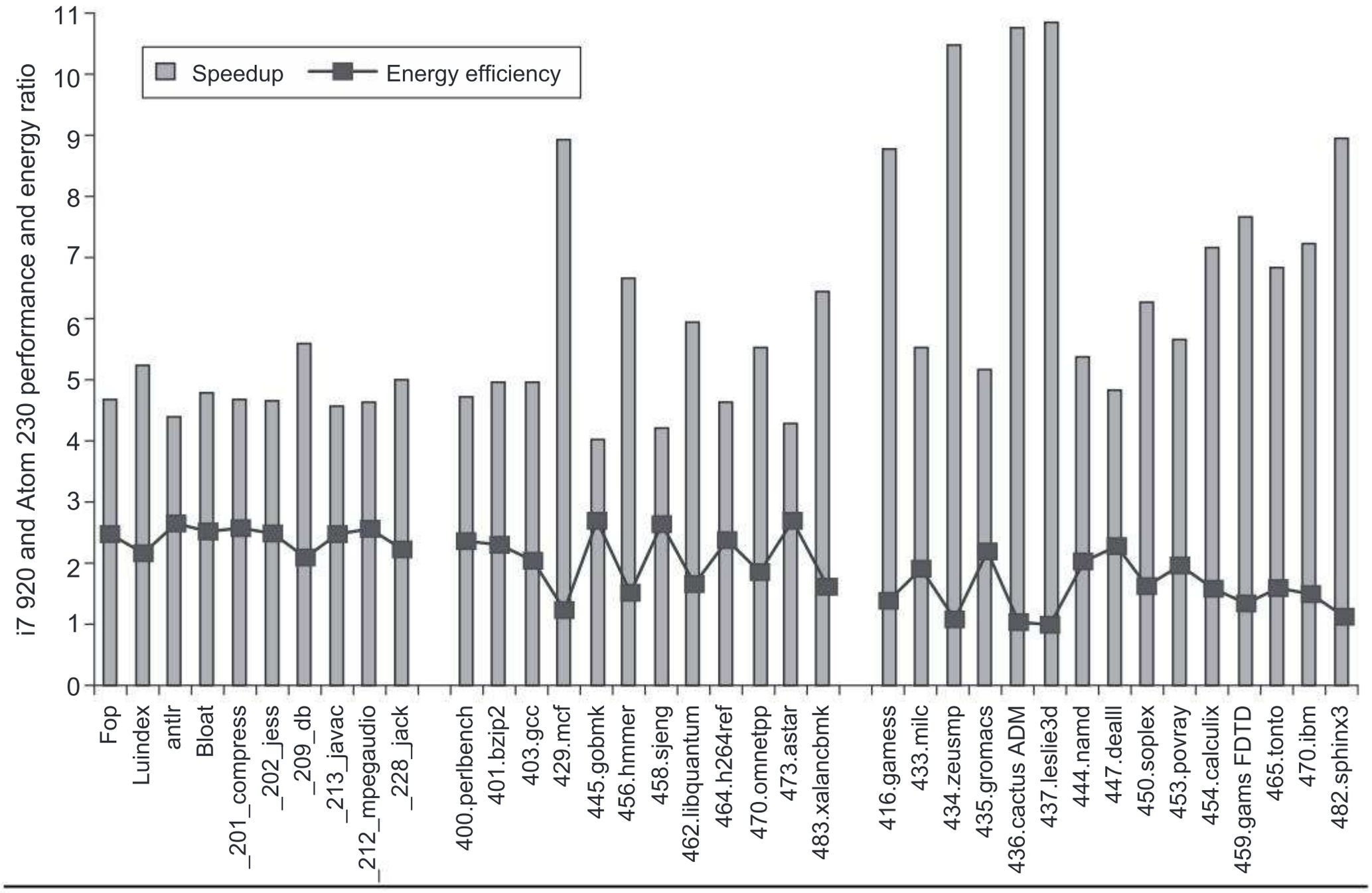


Figure 3.43 The relative performance and energy efficiency for a set of single-threaded benchmarks shows that the i7 920 is 4 to over 10 times faster than the Atom 230 but that it is about 2 times *less* power efficient on average! Performance is shown in the columns as i7 relative to Atom, which is 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 Esmaeilzadeh et al. (2011). The SPEC benchmarks were compiled with optimization 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.

Processor	Implementation technology	Clock rate	Power	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	90 nm	3.8 GHz	115 W	11.5	12.2
Intel Itanium 2	90 nm	1.66 GHz	104 W approx. 70 W one core	14.5	17.3
Intel i7 920	45 nm	3.3 GHz	130 W total approx. 80 W one core	35.5	38.4

Figure 3.44 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; the Power column is the thermal design power with estimates for only one core active in the multicore cases.

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 MiB 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.

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 OOO 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 last-level cache miss that, when going to main memory, were likely to be 50−100 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. There was another change that exemplified 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 and 8 MiB compared to the 2 MiB 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 based on the data in Figure 3.44, doing so seems to be more effective.

Pitfall *And sometimes smarter is better than bigger and dumber.*

One of the more surprising results of the past decade has been in branch prediction. The emergence of hybrid tagged predictors has shown that a more sophisticated predictor can outperform the simple gshare predictor with the same number

of bits (see Figure 3.10). One reason this result is so surprising is that the tagged predictor actually stores fewer predictions, because it also consumes bits to store tags, whereas gshare has only a large array of predictions. Nonetheless, it appears that the advantage gained by not misusing a prediction for one branch on another branch more than justifies the allocation of bits to tags versus predictions.

Pitfall *Believing that there are large amounts of ILP available, if only we had the right techniques.*

The attempts to exploit large amounts of ILP failed for several reasons, but one of the most important ones, which some designers did not initially accept, is that it is hard to find large amounts of ILP in conventionally structured programs, even with speculation. A famous study by David Wall in 1993 (see Wall, 1993) analyzed the amount of ILP available under a variety of idealistic conditions. We summarize his results for a processor configuration with roughly four to eight times the capability of the most advanced processors in 2023. Wall's study extensively documented a variety of different approaches, and the reader interested in the challenge of exploiting ILP should read the complete study.
The aggressive processor we consider has the following characteristics:

1. Up to 64 instruction issues and dispatches per clock with *no* issue restrictions, or 8 times the total issue width of the widest processor in 2023 (the IBM Power10) and with up to 32 times as many loads and stores allowed per clock! As we have discussed, there are serious complexity and power problems with large issue rates.
2. A tournament predictor with 1K entries and a 16-entry function return predictor. This predictor is comparable to the best predictors in 2016; the predictor is not a primary bottleneck. Mispredictions are handled in one cycle, but they limit the ability to speculate.
3. Perfect disambiguation of memory references done dynamically—this is ambitious but perhaps attainable for small window sizes.
4. Register renaming with 64 additional integer and 64 additional FP registers, which is somewhat less than the most aggressive processor in 2011. Because the study assumes a latency of only one cycle for all instructions (versus 1–4 for common instructions and up to 15 for complex instructions on processors like Golden Cove), the effective number of rename registers is about five times larger than either of those processors.

Figure 3.45 shows the result for this configuration as we vary the window size. This configuration is more complex and expensive than existing implementations, especially in terms of the number of instruction issues. Nonetheless, it gives a useful upper limit 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; for example, they may all be memory references. No one would even contemplate this capability in a processor in the near

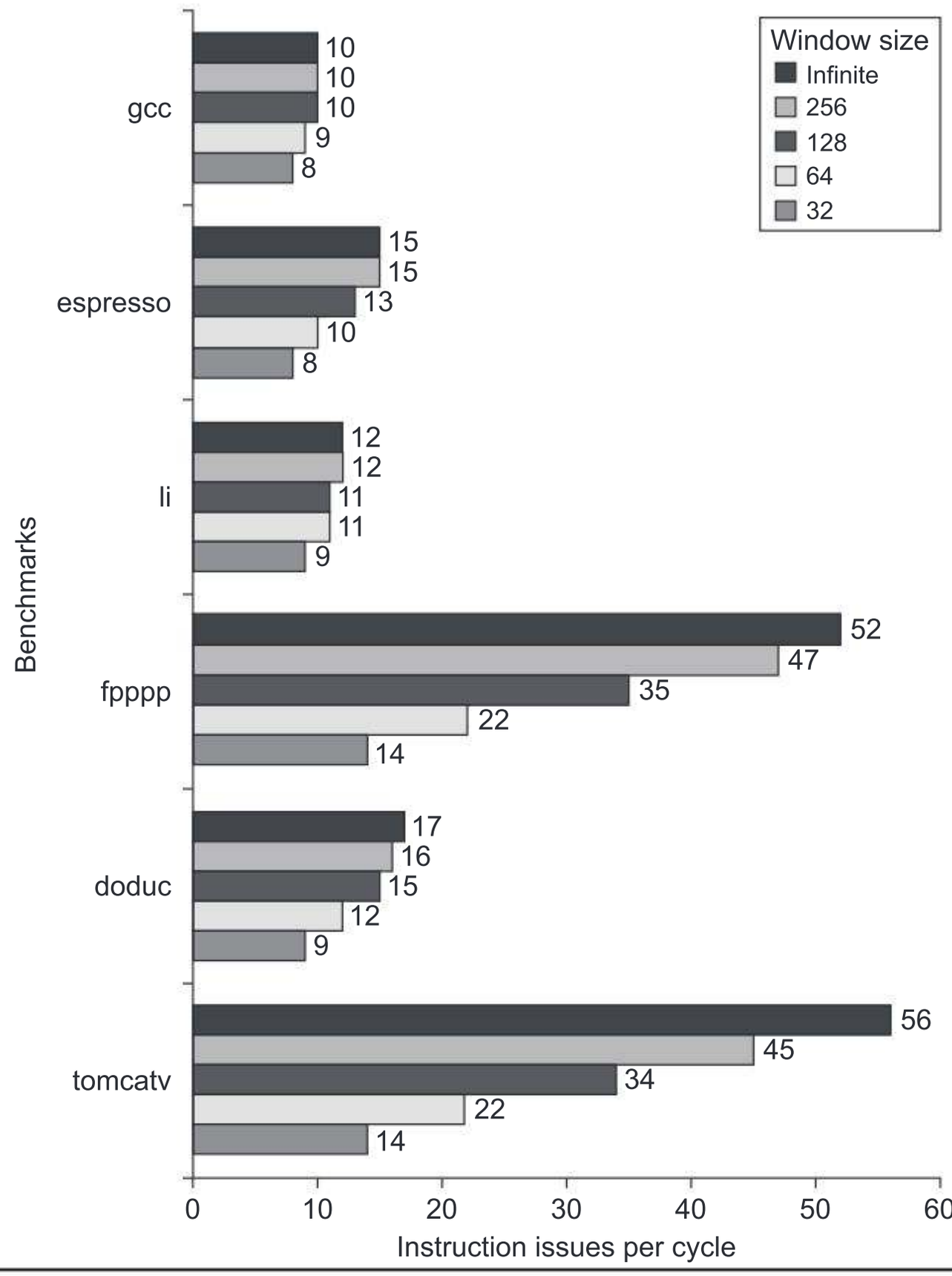


Figure 3.45 The amount of parallelism available versus the window size for a variety of integer and floating-point programs with up to 64 arbitrary instruction issues per clock. Although there are fewer renaming registers than the window size, the fact that all operations have 1-cycle latency and that the number of renaming registers equals the issue width allows the processor to exploit parallelism within the entire window.

future. In addition, remember that in interpreting these results, cache misses and nonunit latencies were not taken into account, and both these effects have significant impacts.

The most startling observation in Figure 3.45 is that with the preceding realistic processor constraints, 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 most of the market growth in the past decade—transaction processing, web servers, and the like—depended on integer performance rather than floating point.

Wall's study was not believed by some, but 10 years later, the reality had sunk in, and the combination of modest performance increases with significant hardware resources and major energy issues coming from incorrect speculation forced a change in direction. We will return to this discussion in our concluding remarks.

3.15 Concluding Remarks

As 2000 began, the focus on exploiting ILP was at its peak. In the first 5 years of the new century, it became clear that the ILP approach had likely peaked and that new approaches would be needed. By 2005, Intel and all the other major processor manufacturers had revamped their approach to focus on multicore. Higher performance would be achieved through TLP rather than ILP, 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 ILP some 25+ years earlier. At this point, it is difficult to find a single-core chip in any but the most deeply embedded application domains.

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, GPUs pursued aggressive use of SIMD, achieving significant performance advantages for applications with extensive DLP. For scientific applications, such approaches represent a viable alternative to the more general, but less efficient, TLP exploited in multicores. Chapter 4 explores these developments in the use of DLP. In another dramatic technology turn around 2015, designers began exploring domain-specific architectures, trading off programmability for large gains in performance and power efficiency for key applications such as machine-learning computations. Chapter 7 focuses on building domain-specific hardware.

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 slow but steady ILP improvements in the cores used as the primary building blocks for multicore designs even for application domains with ample TLP (e.g., data centers). The addition of SMT and its effectiveness, both for performance and energy efficiency, further cemented the position of the OOO speculative approaches. While the rate of improvement in superscalar designs is definitely slower, there is still a steady stream of

improvements introduced in a timely manner by vendors such as Intel, AMD, Arm, Apple, and IBM. Indeed, even in the embedded market, the newest processors have introduced dynamic scheduling, speculation, and wider issue rates. One of the best-performing OOO cores in 2023 is the Apple Avalanche core that is used in the A15 chip for smartphones and the M2 chip for notebooks.

It is unlikely that the rate of ILP improvements will accelerate significantly. It is simply too inefficient from the viewpoint of silicon utilization and power efficiency. However, it is likely that the slow rate of ILP improvements will continue. The importance of single-thread performance for several workloads and the wide applicability of ILP techniques are still important considerations. Moreover, the ability to balance efficiency concerns by combining within a single chip high-performance and high-efficiency cores or programmable cores and custom accelerators enables the deployment of ever-improving OOO cores. Consider the data in Figure 3.46 that shows the seven processors in the IBM Power series. The Power processors target high performance for enterprise applications such as banking systems and customer relationship management and tend to be leading ILP designs. Over two decades, there has been a steady improvement in the ILP support in the Power processors, but the dominant portion of the increase in transistor count (a factor of more than 100 from the Power4 to the Power10) went to increasing the capacity of on-chip caches and the number of cores per die. The maximum number of instructions issued per cycle for a single thread went from 5 in Power4 to 8 in Power10. The Power10 SMT8 core is organized in two execution domains, with each domain servicing 8 threads with enough resources to issue 8 instructions per cycle. At the same time, the SMT support went from nonexistent to 8 threads/processor. A similar trend can be observed across multiple generations of high-performance Intel microarchitectures, where a big percentage of the additional silicon has gone to supporting more cores, wider SIMD units, and larger caches. The next two chapters focus on approaches that exploit data-level and thread-level parallelism.

	Power4	Power5	Power6	Power7	Power8	Power9	Power10
Year introduced	2001	2004	2007	2010	2014	2017	2021
Peak clock rate (GHz)	1.3	1.9	5.0	4.14	4.35	4.0	4.15
Transistor count (M)	174	276	790	1,200	4,200	8,000	18,000
Issues per clock	5	5	7	6	8	18	16
Functional units per core	8	8	9	12	16	38	42
SMT threads per core	0	2	2	4	8	8	8
Cores/chip	2	2	2	8	12	12	16
Total on chip cache (MiB)	1.5	2	8	35	103.0	128	163

Figure 3.46 Characteristics of seven generations of IBM Power processors. All except the Power6, which is static and in-order, are dynamically scheduled superscalar processors. Power7 and Power8 use embedded DRAM for the L3 cache. Power9 has been described briefly; it further expands the caches and supports off-chip HBM. The Power10 SMT8 core includes two execution resource domains. Each domain provides resources to service up to four hardware threads. Hence the cumulative issue width is 16 instructions per cycle, while a single thread can only issue up to 8 instructions per cycle.

3.16 Historical Perspective and References

Section M.5 (available online) features a discussion on the development of pipelining and ILP. We provide numerous references for further reading and exploration of these topics. Section M.5 covers both Chapter 3 and Appendix H.

Case Studies and Exercises by Jason D. Bakos

Case Study: Dynamic Scheduling

This case study will examine runtime behavior of the following simple loop:

```
loop: fld f2,0(x2)          # load guess
fmul.d f4,f2,f2             # temp= guess^2
fsub.d f4,f4,f0             # temp = temp - s
fmul.d f6,f2,f8             # temp2 = guess * 2
fdiv.d f4,f4,f6             # temp = temp/temp2
fsub.d f2,f2,f4             # temp = guess - temp
fsd f2,0(x2)                # store guess
addi x1,x1,-1               # decrement counter
addi x2,x2,8                # increment address
bne x1,zero,loop            # loop
```

This loop performs one Newton-Raphson iteration toward finding the root of equation $x^2 - s$, assuming:

- s is stored in register f0,
- the constant 2.0 is stored in register f8,
- the iteration count is stored in x1, and
- the base address of the input array is stored in register x2.

3.1 [15] <3.3, 3.6> How many cycles are required to execute this loop body on a single-issue, in-order CPU, assuming:

- loads have a latency of 4 cycles (requires at least 3 intervening cycles between a producer and consumer),
- floating-point instructions have a latency of 5 cycles,
- integer instructions have a latency of 2 cycles,
- branches resolve in 4 cycles,
- stores have an effective latency of 0 cycles,
- dependent instructions can issue in the same cycle as their last producing instruction completes, with infinite number of forwarding busses, and

- the loop is considered completed in the cycle the latest instruction commits.

id	instruction	depends on	consumer instruction	ready in cycle	dispatches in cycle	completes in cycle	commits/ resolves in cycle
0	fld f2,0(x2)						
1	fmul.d f4,f2,f2						
9	bne x1,zero,loop						

To solve this problem, fill in a table like the one below:

3.2 [5] <3.3, 3.6> What percentage of cycles did the CPU issue an instruction in your solution to 3.1?

3.3 [15] <3.3, 3.6> Assume that the CPU can issue and commit instructions out of order, but only one instruction can dispatch per cycle and only one instruction can commit each cycle.

If more than one instruction is ready to dispatch or commit in the same cycle, then the instruction fetched latest must stall. For now, assume that the registers are automatically renamed to avoid inconsistencies caused by WAR and WAW name dependencies. We will examine if this is necessary in a later question.

How many cycles are required for the loop body? Use a table like the one you used for 3.1.

3.4 [5] <3.3, 3.6> What percentage of cycles did the CPU issue instructions in your solution 3.3?

3.5 [15] <3.4, 3.6> Now assume that branches are speculated, the branch predictor is 100% accurate. Instructions fetched after the branch can be dispatched before the branch resolves, but they cannot be committed until after the branch resolves.

On average, how many cycles are required for each iteration over the course of two iterations? To solve this, use a table to schedule the instructions like the one you used for 3.1.

3.6 [5] <3.4, 3.6> What percentage of cycles did the CPU dispatch instructions in your solution to 3.5?

3.7 [15] <3.5> Assume each instruction's destination register is considered "live" starting from the cycle the instruction commits until the last of its dependent instructions are dispatched.

For the schedule you developed in 3.5, find the ranges of live cycles for registers x1, x2, f2, f4, and f6.

3.8 [10] <3.5> Do the register lifetimes you found in 3.7 indicate that there are any name (WAR, WAW) dependencies in the code that could cause problems given the dynamic ordering of the instructions? Why or why not?

3.9 [15] <3.3, 3.6> Now assume the target architecture supports dual issue and dual commit. Assume the architecture has an infinitely large instruction window from which to select instructions to schedule. How many cycles are required for each iteration, on average, for two iterations?

3.10 [5] <3.6> What percentage of the issue slots were utilized in your response to 3.9?

3.11 [5] <3.6> Are the performance results from 3.9 what you expected? Why or why not?

Exercises

3.12 [25] <3.11> In this exercise you will explore performance trade-offs between three processors that each employ different types of multithreading (MT). 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 dependencies.

Our application is a list searcher, which scans a region of memory for a specific value stored in `x9` between the address range specified in `x16` and `x17`. It is parallelized by evenly dividing the search space into four equal-sized contiguous blocks and assigning one search thread to each block (yielding four threads). Most of each thread's runtime is spent in the following unrolled loop body:

```
loop: lw x1,0(x16)
      lw x2,8(x16)
      lw x3,16(x16)
      lw x4,24(x16)
      lw x5,32(x16)
      lw x6,40(x16)
      lw x7,48(x16)
      lw x8,56(x16)
      beq x9,x1,match0
      beq x9,x2,match1
      beq x9,x3,match2
      beq x9,x4,match3
      beq x9,x5,match4
      beq x9,x6,match5
      beq x9,x7,match6
      beq x9,x8,match7
      daddiu x16,x16,#64
      blt x16,x17,loop
```

Assume the following:

- A barrier is used to ensure that all threads begin simultaneously.
- The first L1 cache miss occurs after two iterations of the loop.
- None of the BEQAL branches is taken.
- The BLT is always taken.
- All three processors schedule threads in a round-robin fashion.
- Processor A is a superscalar simultaneous MT architecture, capable of issuing up to two instructions per cycle from two threads.
- Processor B is a fine-grained MT architecture, capable of issuing up to four instructions per cycle from a single thread and switches threads on any pipeline stall.
- Processor C is a coarse-grained MT architecture, capable of issuing up to eight instructions per cycle from a single thread and switches threads on an L1 cache miss.

Determine how many cycles are required for each processor to complete the first two iterations of the loop.

3.13 [25/25/25] <3.1, 3.2, 3.9> In this exercise we look at how software techniques can extract instruction-level parallelism (ILP) in a common vector loop. The following loop is the so-called DAXPY loop (double-precision aX plus Y) and is the central operation in Gaussian elimination. The following code implements the DAXPY operation, $Y = aX + Y$, for a vector length 100. Initially, R1 is set to the base address of array X and R2 is set to the base address of Y:

```
addi  x4,x1,#800 ; x1 = upper bound for X
foo: fld         F2,0(x1)    ; (F2) = X(i)
     fmul.d      F4,F2,F0    ; (F4) = a*X(i)
     fld         F6,0(x2)    ; (F6) = Y(i)
     fadd.d      F6,F4,F6    ; (F6) = a*X(i) + Y(i)
     fsd         F6,0(x2)    ; Y(i) = a*X(i) + Y(i)
     addi  x1,x1,#8    ; increment X index
     addi  x2,x2,#8    ; increment Y index
     sltu  x3,x1,x4    ; test: continue loop?
     bnez  x3,foo      ; loop if needed
```

Assume the functional unit latencies as shown in the following table. Assume a one-cycle delayed branch that resolves in the ID stage. Assume that results are fully bypassed.

Instruction producing result	Instruction using result	Latency in clock cycles
FP multiply	FP ALU op	6
FP add	FP ALU op	4
FP multiply	FP store	5
FP add	FP store	4
Integer operations and all loads	Any	2

a. [25] <3.1> Assume a single-issue pipeline. Show how the loop would look both unscheduled by the compiler and after compiler scheduling for both floating-point operation and branch delays, including any stalls or idle clock cycles. What is the execution time (in cycles) per element of the result vector, Y, unscheduled and scheduled? How much faster must the clock be for processor hardware alone to match the performance improvement achieved by the scheduling compiler? (Neglect any possible effects of increased clock speed on memory system performance.)

b. [25] <3.2> Assume a single-issue pipeline. Unroll the loop as many times as necessary to schedule it without any stalls, collapsing the loop overhead instructions. How many times must the loop be unrolled? Show the instruction schedule. What is the execution time per element of the result?

c. [25] <3.9> Assume a VLIW processor with instructions that contain five operations, as shown in Figure 3.28. We will compare two degrees of loop unrolling. First, unroll the loop 6 times to extract ILP and schedule it without any stalls (i.e., completely empty issue cycles), collapsing the loop overhead instructions, and then repeat the process but unroll the loop 10 times. Ignore the branch delay slot. Show the two schedules. What is the execution time per element of the result vector for each schedule? What percent of the operation slots are used in each schedule? How much does the size of the code differ between the two schedules? What is the total register demand for the two schedules?

3.14 [20/20] <3.3,3.6> In this exercise we will look at how a processor with dynamic scheduling performs when running the loop from Exercise 3.13. The functional units are described in the following table.

Functional unit type	Cycles in EX	Number of functional units	Number of reservation stations
Integer	1	1	5
FP adder	10	1	3
FP multiplier	15	1	2

Assume the following:

- Functional units are not pipelined.
- There is no forwarding between functional units; results are communicated when instructions complete.
- The execution stage (EX) does both the effective address calculation and the memory access for loads and stores. Thus the pipeline is IF/ID/DIS/EX/WB.
- Loads require one clock cycle.
- The dispatch (DIS) and write-back (WB) result stages each require one clock cycle.
- There are five load buffer slots and five store buffer slots.
- Assume that the Branch on Not Equal to Zero (BNEZ) instruction requires one clock cycle.

a. [20] <3.3,3.6> For this problem use a single-issue pipeline with the pipeline latencies from the preceding table. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution in (i.e., enters its first EX cycle) for three iterations of the loop. How many cycles does each loop iteration take? Report your answer in the form of a table with the following column headers:

   - Iteration (loop iteration number)
   - Instruction
   - Dispatches (cycle when instruction dispatches)
   - Executes (cycle when instruction executes)
   - Memory access (cycle when memory is accessed)
   - Completes (cycle when result becomes available to other instructions)
   - Comment (description of any event on which the instruction is waiting)

   Show three iterations of the loop in your table. You may ignore the first instruction.

b. [20] <3.3, 3.6> Repeat part (a) but this time assume a two-issue processor and a fully pipelined floating-point unit (FPU).

3.15 [10] <3.6> Assume a dynamically scheduled processor that can only broadcast (forward) one functional unit result to waiting instructions per cycle. Use the hardware configuration and latencies from the previous question (3.14) and find a code sequence of no more than 10 instructions where this processor must stall due to the structural hazard caused by this limitation. Indicate where this occurs in your sequence.

3.16 [20] <3.4> An (m,n) correlating branch predictor uses the behavior of the most recent m executed branches to choose from 2^m predictors, each of which is an n-bit predictor. A two-level local predictor works in a similar fashion but only keeps track of the past behavior of each individual branch to predict future behavior.

There is a design trade-off involved with such predictors: correlating predictors require little memory for history, which allows them to maintain 2-bit predictors for a large number of individual branches (reducing the probability of branch instructions reusing the same predictor), while local predictors require substantially more memory to keep history and are thus limited to tracking a relatively small number of branch instructions.

For this exercise, consider a (1,2) correlating predictor that can track four branches (requiring 16 bits) versus a (1,2) local predictor that can track two branches using the same amount of memory. For the following branch outcomes, provide each prediction, the table entry used to make the prediction, any updates to the table as a result of the prediction, and the final misprediction rate of each predictor. Assume that all branches up to this point have been taken. Initialize each predictor to the following:

Correlating predictor

Entry	Branch	Last outcome	Prediction
0	0	T	T with one misprediction
1	0	NT	NT
2	1	T	NT
3	1	NT	T
4	2	T	T
5	2	NT	T
6	3	T	NT with one misprediction
7	3	NT	NT

Local predictor

Entry	Branch	Last 2 outcomes (right is most recent)	Prediction
0	0	T,T	T with one misprediction
1	0	T,NT	NT
2	0	NT,T	NT
3	0	NT	T
4	1	T,T	T
5	1	T,NT	T with one misprediction
6	1	NT,T	NT
7	1	NT,NT	NT

Branch PC (word address)	Outcome
454	T
543	NT
777	NT
543	NT
777	NT
454	T
777	NT
454	T
543	T

3.17 [10] <3.4> Suppose we have a deeply pipelined processor, for which we implement a branch-target buffer for the conditional branches only. Assume that the misprediction penalty is always four cycles and the buffer miss penalty is always three cycles. Assume a 90% hit rate, 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed two-cycle branch penalty? Assume a base clock cycle per instruction (CPI) without branch stalls of one.

3.18 [10/5] <3.4> Consider a branch-target buffer that has penalties of zero, two, and two clock cycles for correct conditional branch prediction, incorrect prediction, and a buffer miss, respectively. Consider a branch-target buffer design that distinguishes conditional and unconditional branches, storing the target address for a conditional branch and the target instruction for an unconditional branch.

a. [10] <3.9> What is the penalty in clock cycles when an unconditional branch is found in the buffer?
b. [10] <3.9> Determine the improvement from branch folding for unconditional branches. Assume a 90% hit rate, an unconditional branch frequency of 5%, and a two-cycle penalty for a buffer miss. How much improvement is gained by this enhancement? How high must the hit rate be for this enhancement to provide a performance gain?

3.19 [10] <3.2, 3.7> Consider the following code, which multiplies a sparse matrix A by a dense vector x. The sparse matrix is stored in "Compressed Sparse Row (CSR)" format, where all the non-zero entries of the matrix are stored in a contiguous block of memory in row-major order and the *i*th nonzero matrix entry is stored in `val[i]`, its corresponding column in `col[i]`, and its corresponding row is `n`, where `ptr[n]` $<= i <$ `ptr[n+1]` (i.e. the `ptr` array stores the index of the first element of each row).

```
loop:  fld f0,0(x1) # load val[i]
       lw x3,0(x2) # load col[i]
       slli x3,x3,3 # temp=col[i] * 8
       add x4,x10,x3 # compute eff. address (&x[0]+temp)
       fld f2,0(x4) # load x[col[i]]
       fmac.d f4,f0,f2 # multiply-accumulate
       addi x1,x1,8 # increment &val[i]
       addi x2,x2,8 # increment &col[i]
       addi x5,x5,1 # increment i
       bne x5,x11,skip # check for end of row (i!=ptr[i])
       fsd f4,0(x6) # store dot product
       addi x7,x7,8 # increment &ptr[row]
       beq x7,x12,done # check if last row was processed
       addi x6,x6,8 # increment &x[row]
       lw x11,(x7) # load ptr[row]
       fli f4,0 # clear accumulator
skip:
       b loop
```

a. [5] <3.7> Which load instructions might require the use of speculative memory disambiguation? Is there a possibility of a RAW dependency through memory?
b. [5] <3.2> How well would you expect a branch predictor to perform for the branch shown on the 10th instruction?

3.20 [10] <3.2, 3.5> Consider the following code, which performs Horner's method to compute a polynomial for two input values stored in registers f6 and f8.

```
loop: fld f0,0(x1) # load coefficient
      fmul.d f2,f6,f10 # temp = x1 * accum
      fadd.d f10,f2,f0 # accum=temp+coeff (f10 is loop-carried)
      fmul.d f2,f8,f12 # temp = x2 * accum
      fadd.d f12,f2,f0 # accum=temp2+coeff(f12 is loop-carried)
      addi x1,x1,8
      bne x1,x2,loop
```

Unroll the loop by a factor of 2 and rename any floating-point registers with alternative names to avoid performance loss caused by name dependencies. Afterward, show the final register map.

3.21 [5] <3.7> For the following code snippet, find the number of cycles required to execute this code for a CPU without load bypassing and speculative memory disambiguation and for a CPU with load bypassing and speculative memory disambiguation. Assume there is no dependency between the store and load instructions.

Assume a single-issue OOO pipeline with one integer unit with a 3-cycle latency and one load/store unit with a 4-cycle latency. Assume instructions can issue in the same cycle as the instruction with its last dependency completes. State any other assumptions you make.

```
add x8,x10,x11
add x6,x8,x9
add x3,x6,x7
sw x2,0(x3)
lw x4,0(x5)
```

Additional References

Boggs, D., Weiss, S., Kyker, A., 2004. Branch Reorder Buffer. US Patent 6,799,268.

Hu, G., He, Z., Lee, R.B., 2021. SoK: hardware defenses against speculative execution attacks. In: Proceedings of the 2021 International Symposium on Secure and Private Execution Environment Design (SEED), pp. 108–120. https://doi.org/10.1109/SEED51797.2021.00023.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., et al., 2019. Spectre attacks: exploiting speculative execution. In: The Proceedings of the 40th IEEE Symposium on Security and Privacy.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., et al., 2018. Meltdown: reading kernel memory from user space. In: Proceedings of the 27th USENIX Security Symposium.

Moshovos, A., Breach, S.E., Vijaykumar, T.N., Sohi, G.S., 1997. Dynamic speculation and synchronization of data dependences. In: Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97). Association for Computing Machinery, New York, NY, USA, pp. 181–193. https://doi.org/10.1145/264107.264189.

Rotem, E., et al., 2022. Intel Alder Lake CPU Architectures. in IEEE Micro 42 (3), 13–19. https://doi.org/10.1109/MM.2022.3164338.

This page intentionally left blank

4.1	Introduction	288
4.2	Vector Architecture	289
4.3	SIMD Instruction Set Extensions for Multimedia	309
4.4	Graphics Processing Units	318
4.5	Detecting and Enhancing Loop-Level Parallelism	347
4.6	Cross-Cutting Issues	356
4.7	Putting It All Together: Embedded Versus Server GPUs and Tesla Versus Core i7	357
4.8	Fallacies and Pitfalls	366
4.9	Concluding Remarks	368
4.10	Historical Perspective and References	369
	Case Study and Exercises by Jason D. Bakos	369
	References	376