Advanced Topics in Computer Architecture

Lecture 3

Instruction-Level Parallelism and

Its Exploitation

Marenglen Biba
Department of Computer Science
University of New York Tirana

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Reducing Branch Costs with Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

Towards instruction-level parallelism

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

Hardware and Software Approach

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

Pipeline CPI

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

  Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

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

What Is Instruction-Level Parallelism?

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

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

- Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.
- There are a number of techniques we will examine for converting such loop-level parallelism into instruction-level parallelism.

Basically, such techniques work by unrolling the loop either statically by the compiler or dynamically by the hardware.

Data Dependences and Hazards

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

  The key in both cases is to determine whether an instruction is dependent on another instruction.

Data Dependences

- There are three different types of dependences: *data dependences* (also called true data dependences), *name dependences*, and *control dependences*
- An instruction *j* is *data dependent* on instruction *i* if either of the following holds:
  - instruction *i* produces a result that may be used by instruction *j*, or
  - instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i*.
- The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions.
  - This dependence chain can be as long as the entire program.

Example

- For example, consider the following MIPS code sequence that increments a vector of values in memory (starting at 0(R1), and with the last element at 8(R2)), by a scalar in register F2.

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

```
Loop:   L.D    F0,0(R1)    ;F0=array element
        ADD.D  F4,F0,F2    ;add scalar in F2
        S.D    F4,0(R1)    ;store result
```

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

Data dependent instructions

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

Overcoming limits of data dependence

- Since a data dependence can limit the amount of instruction-level parallelism we can exploit, a major focus is overcoming these limitations.
- A dependence can be overcome in two different ways:
  - *maintaining the dependence but avoiding a hazard*
  - *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.

Detecting dependencies

- A data value may flow between instructions either through registers or through memory locations.
- When the data flow occurs in a register, detecting the dependence is straightforward since the register names are fixed in the instructions, although it gets more complicated when branches intervene and correctness concerns force a compiler or hardware to be conservative.
- Dependences that flow through memory locations are more difficult to detect, since two addresses may refer to the same location but look different: For example, 100(R4) and 20(R6) may be identical memory addresses.
- In addition, the effective address of a load or store may change from one execution of the instruction to another (so that 20(R4) and 20(R4) may be different), further complicating the detection of a dependence.

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:

Antidependence

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.

Output dependence

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*.

Register renaming

- Both antidependences and output dependences are name dependences, as opposed to true data dependences, since there is no value being transmitted between the instructions.
- Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.
- This renaming can be more easily done for register operands, where it is called *register renaming*.
- Register renaming can be done either statically by a compiler or dynamically by the hardware.

Data Hazards

- A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence.
- Because of the dependence, we must preserve what is called *program order*, that is, the order that the instructions would execute if executed sequentially one at a time as determined by the original source program.
- The goal of both 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

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

Data Hazards

- WAR *(write after read)* — *j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the *new* value.
- WAR hazards cannot occur in most static issue pipelines — even deeper pipelines or floating-point pipelines — because all reads are early (in ID) and all writes are late (in WB).
- 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 => our case here.

Control Dependences

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

Control Dependences: Example

- 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.

Control dependences

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

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

Programs critical properties

- When processors preserve *strict program order*, they ensure that control dependences are also preserved.
- We may be willing to execute instructions that should not have been executed, however, (thereby violating the control dependences), *if* we can do so without affecting the correctness of the program.
- Control dependence is not the critical property that must be preserved.
  - Instead, the two properties critical to program correctness — and normally preserved by maintaining both data and control dependence — are the *exception behavior* and the *data flow*.

Programs critical properties

- 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.
- The *data flow* is the actual flow of data values among instructions that produce results and those that consume them.
- Branches make the data flow dynamic, since they allow the source of data for a given instruction to come from many points.

Program order is ensured by maintaining the control dependences.

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Reducing Branch Costs with Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

Basic Pipeline Scheduling and Loop Unrolling

- To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline.

- To avoid a pipeline stall, *a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction*.

- A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline

Latencies of FP operations

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

Figure 2.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0, since the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0.

Loop unrolling

- A simple scheme for increasing performance is *loop unrolling*.

- Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

- Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together.

Decisions for Loop Unrolling and Scheduling

- To obtain the final unrolled code the following decisions and transformations could be made:
  - Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
  - Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations.
  - Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.

Decisions for Loop Unrolling and Scheduling

- 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.

Limits of Loop Unrolling

- A decrease in the amount of overhead amortized with each unroll
  - The more you unroll, the more the amortized overhead is reduced
- Code size limitations
  - A second limit to unrolling is the growth in code size that results. For larger loops, the code size growth may be a concern particularly if it causes an increase in the instruction cache miss rate.
- Compiler limitations
  - Effect caused: *Register Pressure*

Register Pressure

- Aggressive unrolling and scheduling can cause potential shortfall in registers.
- This secondary effect that results from instruction scheduling in large code segments is called *register pressure*.
- It arises because scheduling code to increase ILP causes the number of live values to increase.
- After aggressive instruction scheduling, it may not be possible to allocate all the live values to registers.
- The transformed code, while theoretically faster, may lose some or all of its advantage because it generates a shortage of registers.

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Reducing Branch Costs with Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

Branches

- Branches hurt pipeline performance.
- Loop unrolling is one way to reduce the number of branch hazards.
- We can also reduce the performance losses of branches by predicting how they will behave.
- The behavior of branches can be predicted both statically at compile time and dynamically by the hardware at execution time.
- Static branch predictors are sometimes used in processors where the expectation is that branch behavior is highly predictable at compile time;
- Static prediction can also be used to assist dynamic predictors.

Static Branch Prediction

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

Profile-based prediction

- A more accurate technique is to predict branches on the basis of profile information collected from earlier runs.

- The key observation that makes this worthwhile is that the behavior of branches is often *bimodally distributed*:
  - An individual branch is often highly biased toward taken or untaken.

Profile-based prediction

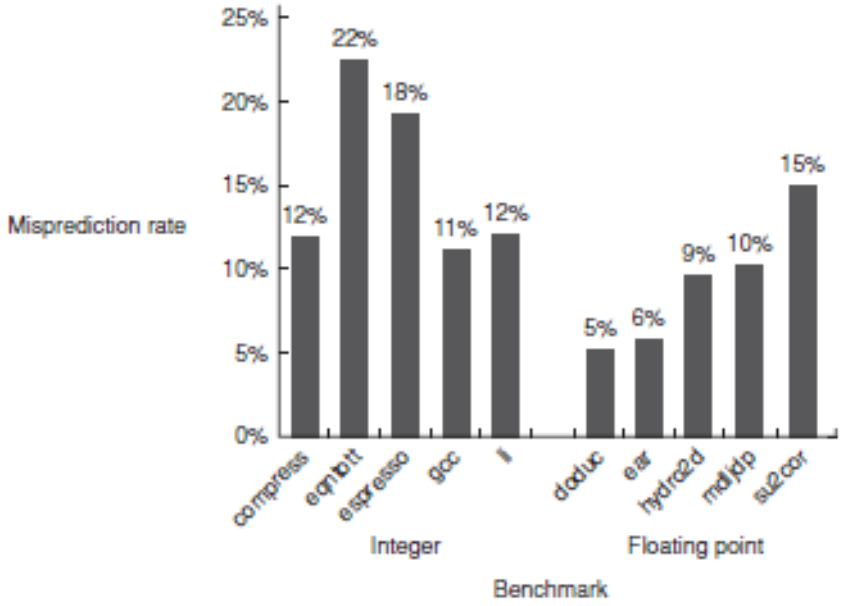


Figure 2.3 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard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

Dynamic Branch Prediction

- The simplest dynamic branch-prediction scheme is a *branch-prediction buffer* or *branch history table*.
- A branch-prediction buffer is a small memory that contains a bit that says whether the branch was recently taken or not.
- This scheme is the simplest sort of buffer.
- With such a buffer, we don't know, in fact, if the prediction is correct.
  - But this doesn't matter. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction.
  - If the hint turns out to be wrong, the prediction bit is inverted and stored back.

2-bit prediction

- The simple 1-bit prediction scheme has a performance shortcoming:

  Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, since the misprediction causes the prediction bit to be flipped.
- To remedy this weakness, 2-bit prediction schemes are often used.
- In a 2-bit scheme, a prediction must miss twice before it is changed.

  (*This scheme can be generalized to the n-bit scheme)*

A finite-state processor for a 2-bit prediction scheme.

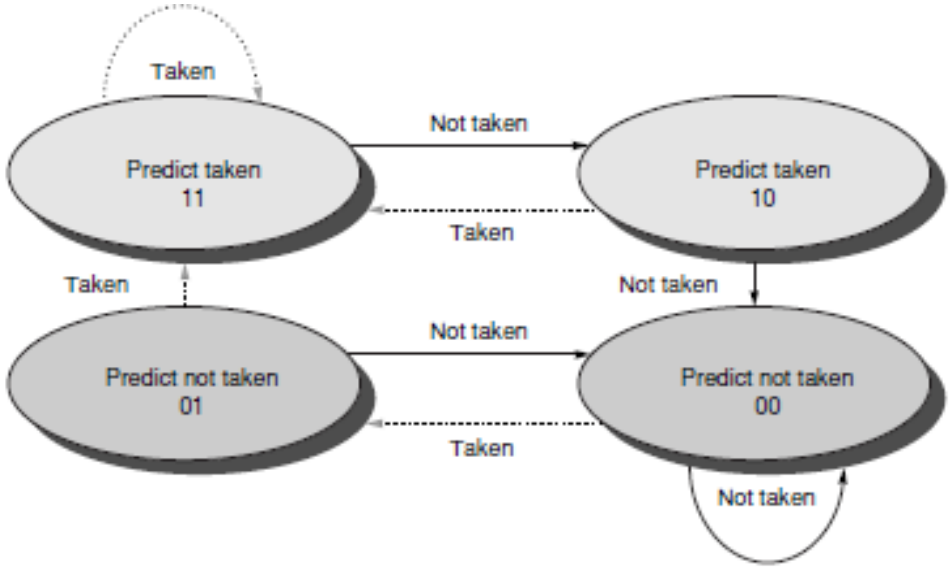


Figure 2.4 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually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

Performance of 2-bit prediction

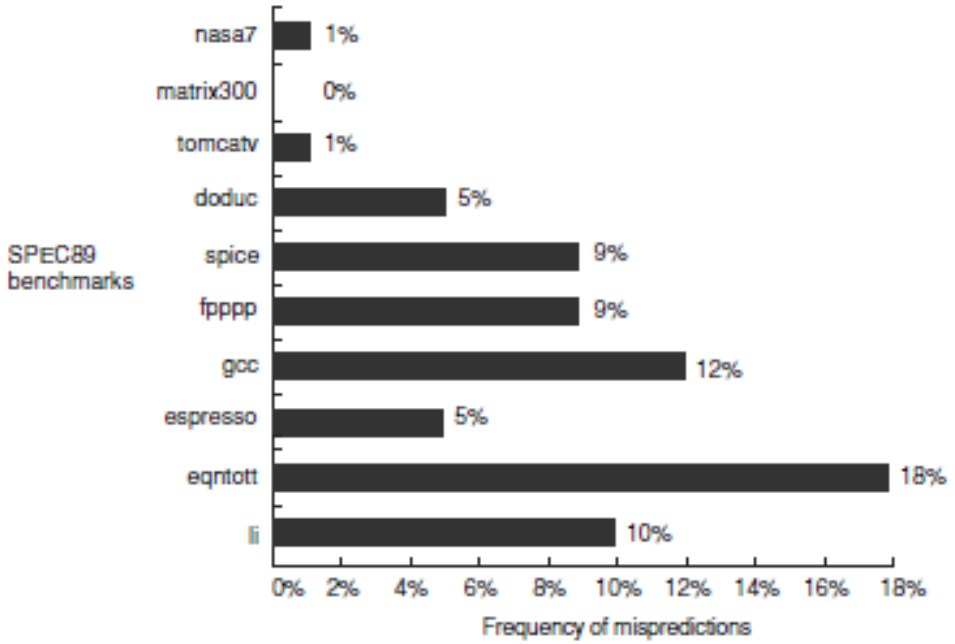


Figure 2.5 Prediction accuracy of a 4096-entry 2-bit prediction buffer for the SPEC89 benchmarks. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the FP programs (average of 4%). Omitting the FP kernels (nasa7,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

Correlating Branch Predictors

- The 2-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch.
- It may be possible to improve the prediction accuracy if we also look at the recent behavior of *other* branches rather than just the branch we are trying to predict.
- Example:

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

Correlating Branch Predictors

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

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

Correlating predictors or two-level predictors

- 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.

Tournament Predictors

- The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor using only local information failed on some important branches and that, by adding global information, the performance could be improved.
- *Tournament predictors* take this insight to the next level, by using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector.
- Existing tournament predictors use a 2-bit scheme per branch to choose among two different predictors based on which predictor (local, global, or even some mix) was most effective in recent predictions.

Comparison of predictors

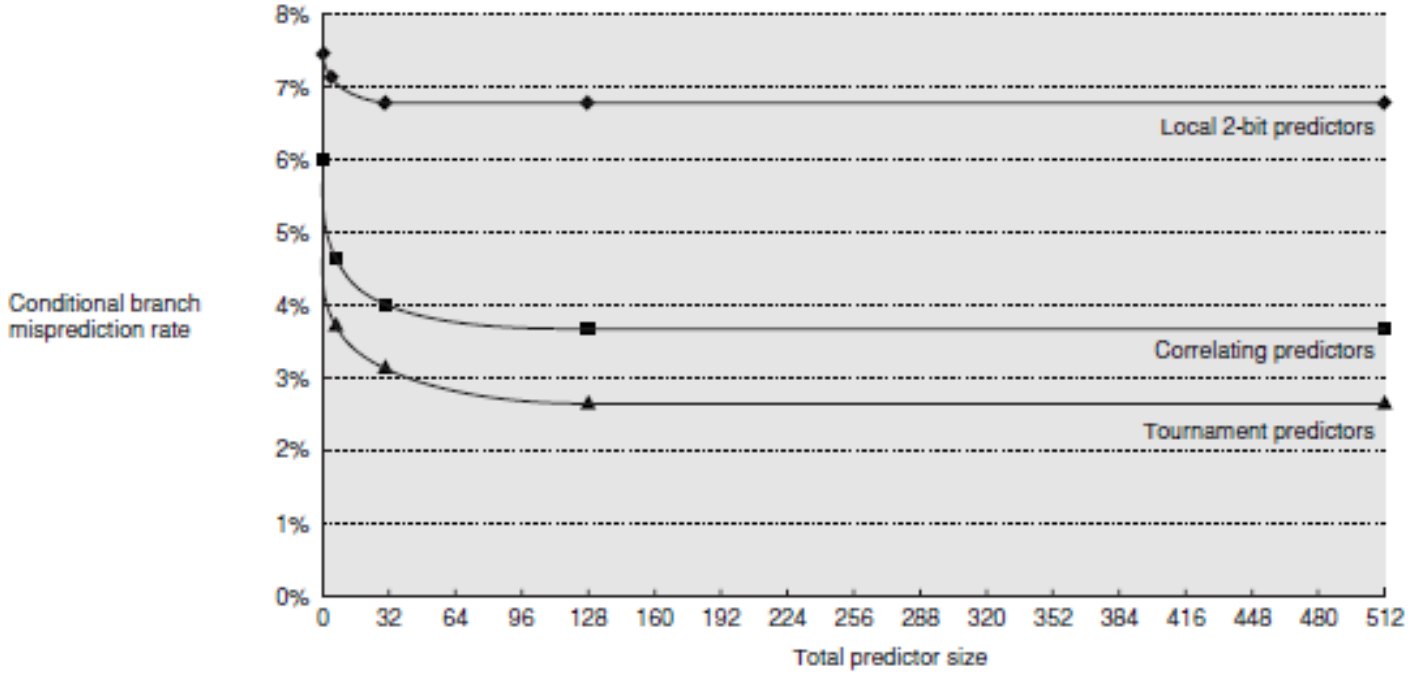


Figure 2.8 The misprediction rate for three different predictors on SPEC89 as the total number of bits is increased. The predictors are a local 2-bit predictor, a correlating predictor, which is optimally structured in its use of global and local information at each point in the graph, and a tournament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

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Reducing Branch Costs with Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

Advantages of Dynamic Scheduling

- It enables handling some cases when dependences are unknown at compile time (for example, because they may involve a memory reference), and it simplifies the compiler.

- It allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve.

- It allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline.

- However, advantages of dynamic scheduling are gained at a cost of a significant increase in hardware complexity.

Static Vs. Dynamic Scheduling

- Although a dynamically scheduled processor cannot change the data flow, it tries to avoid stalling when dependences are present.

- In contrast, static pipeline scheduling by the compiler tries to minimize stalls by separating dependent instructions so that they will not lead to hazards.

- Of course, compiler pipeline scheduling can also be used on code destined to run on a processor with a dynamically scheduled pipeline.

Dynamic Scheduling: The Idea

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

Example

- Consider this program:

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

- The SUB.D instruction cannot execute because the dependence of ADD.D on DIV.D causes the pipeline to stall; yet SUB.D is not data dependent on anything in the pipeline.

- This hazard creates a performance limitation that can be eliminated by not requiring instructions to execute in program order.

Out-of-order execution

- To allow us to begin executing the SUB.D in the example, we must separate the issue process into two parts:
  - checking for any structural hazards and
  - waiting for the absence of a data hazard.

- Thus, we still use in-order instruction issue (i.e., instructions issued in program order), but we want an instruction to begin execution as soon as its data operands are available.

- Such a pipeline does *out-of-order execution*, which implies *out-of-order completion*.

Problem!

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

- Example – next slide

Problem example

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

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

  (both these hazards are avoided by the use of register renaming, explained later)

Out-of-order execution and exceptions

- Out-of-order completion also creates major complications in handling exceptions.

- Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that *exactly* those exceptions that would arise if the program were executed in strict program order *actually* do arise.

- Dynamically scheduled processors preserve exception behavior by ensuring that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed.

Out-order execution and imprecise exceptions

- Although exception behavior must be preserved, dynamically scheduled processors may generate *imprecise* exceptions.
- Imprecise exceptions can occur because of two possibilities:
  1. The pipeline may have *already completed* instructions that are *later* in program order than the instruction causing the exception.
  2. The pipeline may have *not yet completed* some instructions that are *earlier* in program order than the instruction causing the exception.

Implementing out-of-order execution

- To allow out-of-order execution, we essentially split the ID pipe stage of a simple five-stage pipeline into two stages:
  1. *Issue* — Decode instructions, check for structural hazards.
  2. *Read operands* — Wait until no data hazards, then read operands.

- In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in-order issue);
- However, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order.

Dynamic Scheduling Using Tomasulo's Approach

- The IBM 360/91 floating-point unit used a sophisticated scheme to allow out-of order execution.

- This scheme, invented by Robert Tomasulo, tracks when operands for instructions are available, to minimize RAW hazards, and introduces register renaming, to minimize WAW and WAR hazards.

- IBM's goal was to achieve high floating-point performance from an instruction set and from compilers designed for the entire 360 computer family, rather than from specialized compilers for the high-end processors.

Avoiding Hazards

- As we will see, RAW hazards are avoided by executing an instruction only when its operands are available.

- WAR and WAW hazards, which arise from name dependences, are eliminated by register renaming.

- *Register renaming* eliminates these hazards by renaming all destination registers, including those with a pending read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of an operand.

Eliminating Hazards

- Consider the following example:

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

- There is an antidependence between the ADD.D and the SUB.D and an output dependence between the ADD.D and the MUL.D, leading to two possible hazards: a WAR hazard on the use of F8 by ADD.D and a WAW hazard on the use of F6 since the ADD.D may finish later than the MUL.D.

Eliminating Hazards

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

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

- In addition, any subsequent uses of F8 must be replaced by the register T. In this code segment, the renaming process can be done statically by the compiler.

Reservation Stations

- In Tomasulo's scheme, register renaming is provided by *reservation stations*, which buffer the operands of instructions waiting to issue.

- The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register.

- In addition, pending instructions designate the reservation station that will provide their input.

Properties of Reservation Stations

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

MIPS floating-point unit using Tomasulo's algorithm

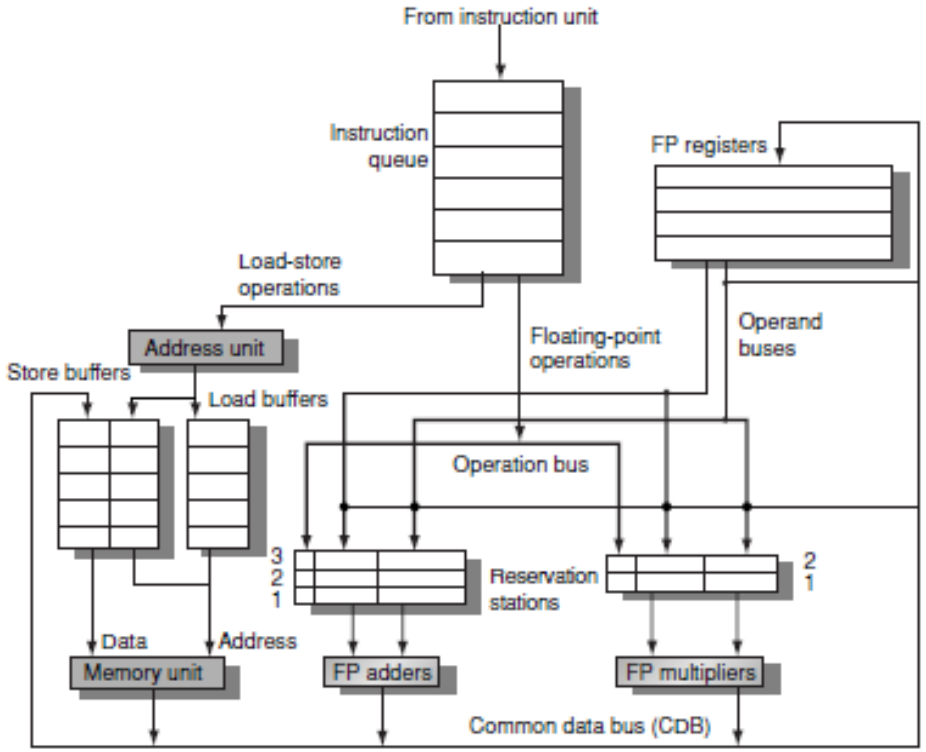


Figure 2.9 The basic structure of a MIPS floating-point unit using Tomasulo's algorithm. Instructions are sent from the instruction unit into the instruction queue from which they are issued in FIFO order. The reservation stations include the operation and the actual operands, as well as information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

Fields of Reservation Stations

- Each reservation station has seven fields:

- Op — The operation to perform on source operands S1 and S2.

- Qj, Qk — The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.

- Vj, Vk — The value of the source operands. Note that only one of the V field or the Q field is valid for each operand.

Fields of Reservation Stations

- A — Used to hold information for the memory address calculation for a load or store.

- Busy — Indicates that this reservation station and its accompanying functional unit are occupied.

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

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

Dynamic Scheduling: Example 1

- Consider the program:

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

- Let’s show what the information tables look like for the code sequence when only the first load has completed and written its result:

  (assume the following latencies: load is 1 clock cycle, add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles.)

Reservation stations in Tomasulo's approach

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

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

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

The code sequence issues both the DIV.D and the ADD.D, even though there is a WAR hazard involving F6. The hazard is eliminated in one of two ways => next slide

Using reservation stations

1. First, if the instruction providing the value for the DIV.D has completed, then Vk will store the result, allowing DIV.D to execute independent of the ADD.D (this is the case shown).
2. On the other hand, if the L.D had not completed, then Qk would point to the Load1 reservation station, and the DIV.D instruction would be independent of the ADD.D.
   - Thus, in either case, the ADD.D can issue and begin executing.
   - Any uses of the result of the DIV.D would point to the reservation station, allowing the ADD.D to complete and store its value into the registers without affecting the DIV.D.

Dynamic Scheduling: Example 2

- Using the same code segment as in the previous example, we show what the status tables look like when the MUL.D is ready to write its result => next slide

Reservation stations in Tomasulo's approach

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

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

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

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

- ADD.D has completed since the operands of DIV.D were copied, thereby overcoming the WAR hazard.
- Notice that even if the load of F6 was delayed, the add into F6 could be executed without triggering a WAW hazard.

Tomasulo's Algorithm

Instruction state	Wait until	Action or bookkeeping

Issue
FP operation
	Station r empty	`if (RegisterStat[rs].Qi≠0)`
		`  {RS[r].Qj ← RegisterStat[rs].Qi}`
		`else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};`
		`if (RegisterStat[rt].Qi≠0)`
		`  {RS[r].Qk ← RegisterStat[rt].Qi`
		`else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
		`RS[r].Busy ← yes; RegisterStat[rd].Q ← r;`

Load or store	Buffer r empty	`if (RegisterStat[rs].Qi≠0)`
		`  {RS[r].Qj ← RegisterStat[rs].Qi}`
		`else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};`
		`RS[r].A ← imm; RS[r].Busy ← yes;`

Load only		`RegisterStat[rt].Qi ← r;`

Store only		`if (RegisterStat[rt].Qi≠0)`
		`  {RS[r].Qk ← RegisterStat[rs].Qi}`
		`  else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`

Execute FP operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load-store step 1	RS[r].Qj = 0 & r is head of load-store queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 complete	Read from Mem[RS[r].A]

Write Result
FP operation or load
	Execution complete at r &
	CDB available
		`∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result;`
		`  RegisterStat[x].Qi ← 0});`
		`∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ←`
		`  0});`
		`∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ←`
		`  0});`
		`RS[r].Busy ← no;`

Store	Execution complete at r & RS[r].Qk = 0	`Mem[RS[r].A] ← RS[r].Vk;` `RS[r].Busy ← no;`

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

Speculation

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

Hardware-based speculation

Hardware-based speculation combines three key ideas:

1. dynamic branch prediction to choose which instructions to execute,
2. speculation to allow the execution of instructions before the control dependences are resolved (with the ability to undo the effects of an incorrectly speculated sequence),
3. dynamic scheduling to deal with the scheduling of different combinations of basic blocks.

Data flow execution

- Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions.
- This method of executing programs is essentially a *data flow execution:* Operations execute as soon as their operands are available.
- To extend Tomasulo's algorithm to support speculation, we must separate the bypassing of results among instructions, which is needed to execute an instruction speculatively, from the actual completion of an instruction.
- By making this separation, we can allow an instruction to execute and to bypass its results to other instructions, without allowing the instruction to perform any updates that cannot be undone, until we know that the instruction is no longer speculative.

Instruction commit

- Using the bypassed value is like performing a speculative register read, since we do not know whether the instruction providing the source register value is providing the correct result until the instruction is no longer speculative.

- When an instruction is no longer speculative, we allow it to update the register file or memory;
  - we call this additional step in the instruction execution sequence *instruction commit*.

Implementing Speculation

- The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit *in order* and to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits.
- Hence, when we add speculation, we need to separate the process of completing execution from instruction commit, since instructions may finish execution considerably before they are ready to commit.
- Adding this commit phase to the instruction execution sequence requires an additional set of hardware buffers that hold the results of instructions that have finished execution but have not committed.
- This hardware buffer, which we call the *reorder buffer*, is also used to pass results among instructions that may be speculated.

Reorder buffer (ROB)

- The reorder buffer (ROB) provides additional registers in the same way as the reservation stations in Tomasulo's algorithm extend the register set.
- The ROB holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits.
- Hence, the ROB is a source of operands for instructions, just as the reservation stations provide operands in Tomasulo's algorithm.
- The key difference is that in Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find the result in the register file.
- With speculation, the register file is not updated until the instruction commits.

ROB Fields

Each entry in the ROB contains four fields:

1. The instruction type indicates whether the instruction is a branch (and has no destination result), a store (which has a memory address destination), or a register operation (ALU operation or load, which has register destinations).
2. The destination field supplies the register number (for loads and ALU operations) or the memory address (for stores where the instruction result should be written.
3. The value field is used to hold the value of the instruction result until the instruction commits.
4. The ready field indicates that the instruction has completed execution, and the value is ready.

FP Unit with Speculation

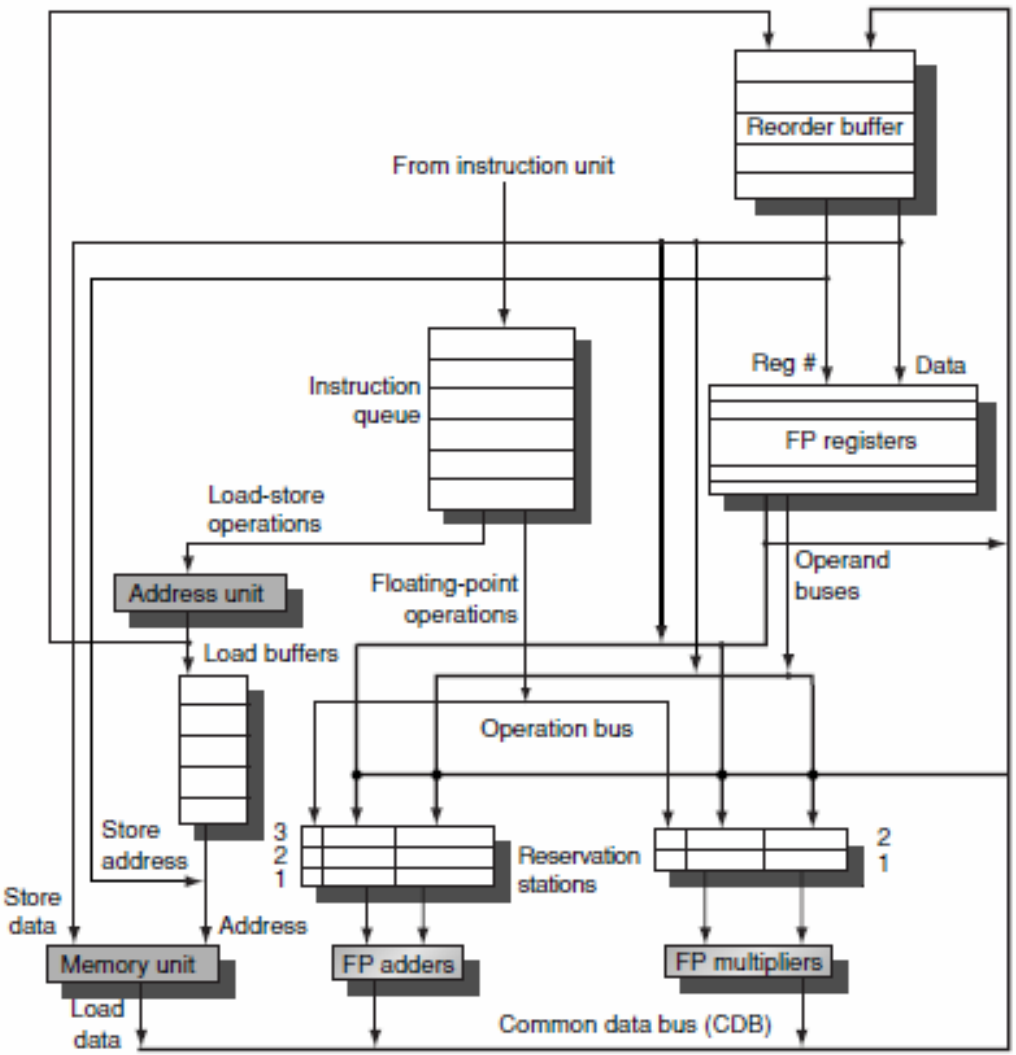


Figure 2.14 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 2.9 on page 94, which implemented Tomasulo's algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated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

Speculation: Example

- Assume latencies for the floating-point functional units are: add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles.
- Consider the program:

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

- Show what the status tables look like when the MUL.D is ready to go to commit.

ROBs

Reorder buffer

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

Reservation stations

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

FP register status

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

Figure 2.15 At the time the MUL.D is ready to commit, only the two L.D instructions have committed, although several others have completed execution. The MUL.D is at the head of the ROB, and the two L.D instructions are there only to ease understanding. The SUB.D and ADD.D instructions will not commit until the MUL.D instruction commits, although the results of the instructions are available and can be used as sources for other instructions.

ROB Vs. Tomasulo

- The key difference is that, with ROB, no instruction after the earliest uncompleted instruction (MUL.D above) is allowed to complete.
- In contrast, with Tomasulo, the SUB.D and ADD.D instructions have also completed.

  One implication of this difference is that the processor with the ROB can dynamically execute code while maintaining a precise interrupt model.
- For example, if the MUL.D instruction caused an interrupt, we could simply wait until it reached the head of the ROB and take the interrupt, flushing any other pending instructions from the ROB.
- Because instruction commit happens in order, this yields a precise exception.
- By contrast, in the example using Tomasulo's algorithm, the SUB.D and ADD.D instructions could both complete before the MUL.D raised the exception.
  - The result is that the registers F8 and F6 (destinations of the SUB.D and ADD.D instructions) could be overwritten, and the interrupt would be imprecise.

Speculation Algorithm

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

Figure 2.17 Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the destina-

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

Multiple Issue

- The techniques we have seen until now can be used to eliminate data and control stalls and achieve an ideal CPI of one.
- To improve performance further we would like to decrease the CPI to less than one.
- But the CPI cannot be reduced below one if we issue only one instruction every clock cycle.
- The goal of the *multiple-issue processors*, is to allow multiple instructions to issue in a clock cycle. Multiple-issue processors come in three major flavors:
  1. statically scheduled superscalar processors
  2. VLIW (very long instruction word) processors
  3. dynamically scheduled superscalar processors

Multiple Issue

- Superscalar processors issue varying numbers of instructions per clock and use in-order execution if they are statically scheduled or out-of-order execution if they are dynamically scheduled.
- VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction.
  - VLIW processors are inherently statically scheduled by the compiler.
- When Intel and HP created the IA-64 architecture, they also introduced the name EPIC — explicitly parallel instruction computer — for this architectural style.

Superscalar and VLIW processors

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

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

Static Vs. Dynamic Scheduling

- Although statically scheduled superscalars issue a varying rather than a fixed number of instructions per clock, they are actually closer in concept to VLIWs, since both approaches rely on the compiler to schedule code for the processor.

- Because of the diminishing advantages of a statically scheduled superscalar as the issue width grows, statically scheduled superscalars are used primarily for narrow issue widths, normally just two instructions.

- Beyond that width, most designers choose to implement either a VLIW or a dynamically scheduled superscalar.

The 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.

Local and Global Scheduling

- 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.
- To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. This parallelism is uncovered by unrolling loops and scheduling the code within the single larger loop body.
- If the unrolling generates straight-line code, then *local scheduling* techniques, which operate on a single basic block, can be used.
- If finding and exploiting the parallelism requires scheduling code across branches, a substantially more complex *global scheduling* algorithm must be used.
- Global scheduling algorithms are not only more complex in structure, but they also must deal with significantly more complicated trade-offs in optimization, since moving code across branches is expensive.

VLIW 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 for such a processor. Unroll as many times as necessary to eliminate any stalls.
- Ignore delayed branches.

for (i=1000; i>0; i=i–1)

x[i] = x[i] + s;

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

VLIW Execution

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

Figure 2.19 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming no branch delay; normally the branch delay would also need to be scheduled. The issue rate is 23 operations in 9 clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a larger number of registers than MIPS would normally use in this loop. The VLIW code sequence above requires at least eight FP registers, while the same code sequence for the base MIPS processor can use as few as two FP registers or as many as five when unrolled and scheduled.

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

- So far, we have seen how the individual mechanisms of dynamic scheduling, multiple issue, and speculation work.
- How about putting them all three together?
- This would yield a microarchitecture quite similar to those in modern microprocessors.
- For simplicity, we consider here only an issue rate of two instructions per clock, but the concepts are no different from modern processors that issue three or more instructions per clock.

ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

- Let's assume we want to extend Tomasulo's algorithm to support a two-issue superscalar pipeline with a separate integer and floating-point unit, each of which can initiate an operation on every clock.
- To gain the full advantage of dynamic scheduling we will allow the pipeline to issue any combination of two instructions in a clock, using the scheduling hardware to actually assign operations to the integer and floating-point unit.
- Because the interaction of the integer and floating-point instructions is crucial, we also extend Tomasulo's scheme to deal with both the integer and floating-point functional units and registers, as well as incorporating speculative execution.

Combining Multiple Instructions with Dynamic Scheduling

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

  They both pipeline and widen the issue logic.

Putting together speculative dynamic scheduling with multiple issue

- Putting together speculative dynamic scheduling with multiple issue requires overcoming one additional challenge at the back end of the pipeline: we must be able to complete and commit multiple instructions per clock.

- Like the challenge of issuing multiple instructions, the concepts are simple, although the implementation may be challenging in the same manner as the issue and register renaming process.

Example

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

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

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

Pipeline with dual-issue and without speculation

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

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

19 Cycles for three loops

Pipeline with dual-issue and with speculation

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

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

It is executed in clock cycle 19 without speculation.

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

Increasing Instruction Fetch Bandwidth

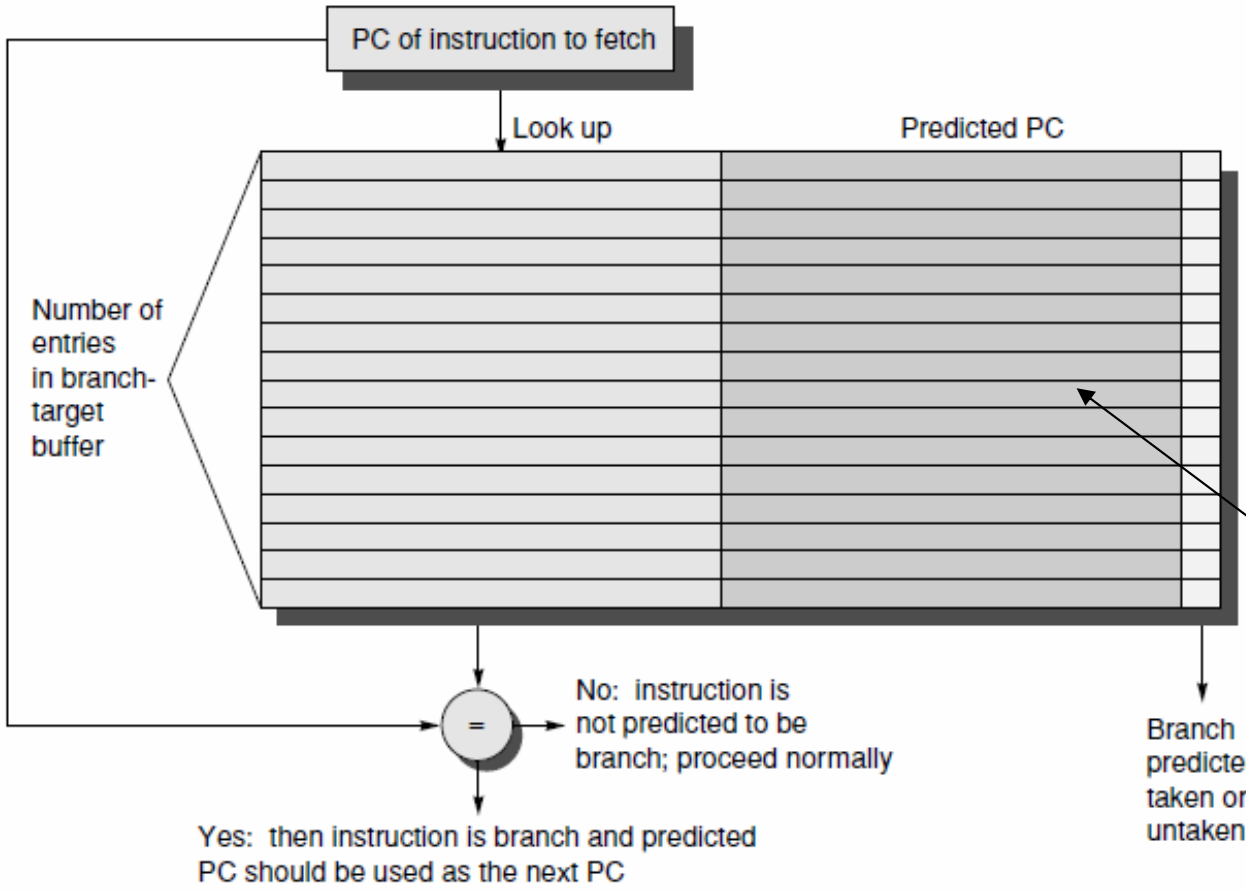
- A multiple issue processor will require that the average number of instructions fetched every clock cycle be at least as large as the average throughput.

- Fetching these instructions requires wide enough paths to the instruction cache, but the most difficult aspect is handling branches.

Branch-Target Buffers

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

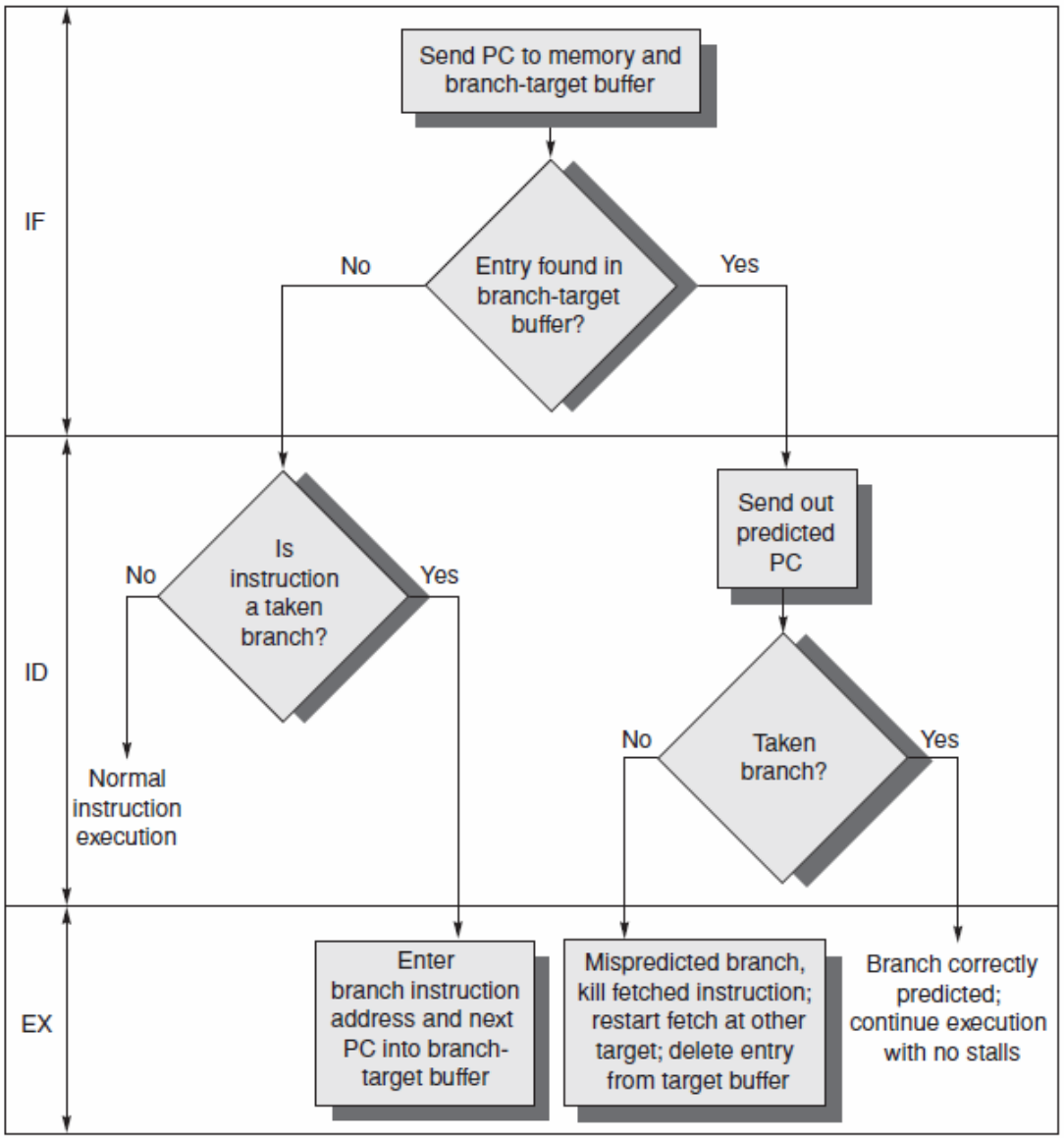
Structure of Branch-Target Buffers



Prediction for the next PC

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

Steps with a branch-target buffer



Pipeline stages

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

Penalties of branch-target buffer

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

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

Determining Penalty: Example

- Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions from previous slide.
- We 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).

Computing Penalty

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

Probability (branch in buffer, but actually not taken) = Percent buffer hit rate × Percent incorrect predictions = 90% × 10% = 0.09

Probability (branch not in buffer, but actually taken) = 10%

Branch penalty = (0.09 + 0.10) × 2

Branch penalty = 0.38 clock cycles

Integrated Instruction Fetch Units

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

Design of Integrated Instruction Fetch Unit

Recent designs have used an integrated instruction fetch unit that integrates several functions:

1. *Integrated branch prediction* — The branch predictor becomes part of the instruction fetch unit and is constantly predicting branches, so as to drive the fetch pipeline.
2. *Instruction prefetch* — To deliver multiple instructions per clock, the instruction fetch unit will likely need to fetch ahead.
   - The unit autonomously manages the prefetching of instructions, integrating it with branch prediction.

Design of Integrated Instruction Fetch Unit

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.

The fetch unit: the future bottleneck

- As designers try to increase the number of instructions executed per clock, instruction fetch will become an ever more significant bottleneck, and clever new ideas will be needed to deliver instructions at the necessary rate.
  - Research challenge for students of CS!

Value Prediction

- One technique for increasing the amount of ILP available in a program is *value prediction*.
- *Value prediction* attempts to predict the value that will be produced by an instruction. Obviously, since most instructions produce a different value every time they are executed (or at least a different value from a set of values), value prediction can have only limited success.
- There are, however, certain instructions for which it is easier to predict the resulting value — for example:
  - loads that load from a constant pool, or
  - loads that load a value that changes infrequently
- In addition, when an instruction produces a value chosen from a small set of potential values, it may be possible to predict the resulting value.

Value Prediction

- Much of the focus of research on value prediction has been on loads.
- We can estimate the maximum accuracy of a load value predictor by examining how often a load returns a value that matches a value returned in a recent execution of the load.
- The simplest case to examine is when the load returns a value that matches the value on the last execution of the load.

Predicting loads

- Because of the high costs of misprediction and the likely case that misprediction rates will be significant (20% to 50%), researchers have focused on assessing which loads are more predictable and only attempting to predict those.
- This leads to a lower misprediction rate, but also fewer candidates for accelerating through prediction.
- In the limit, if we attempt to predict only those loads that always return the same value, it is likely that only 10% to 15% of the loads can be predicted.
- Research on value prediction continues!!

Address Aliasing Prediction

- *Address aliasing prediction* is a simple technique that predicts whether two stores or a load and a store refer to the same memory address.
- If two such references do not refer to the same address, then they may be safely interchanged.
- Otherwise, we must wait until the memory addresses accessed by the instructions are known.
- This limited form of address value speculation has been used by a few processors.

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

Pentium 4

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

Trace Cache in Pentium 4

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

Execution in Pentium 4

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

Pentium 4 Microarchitecture

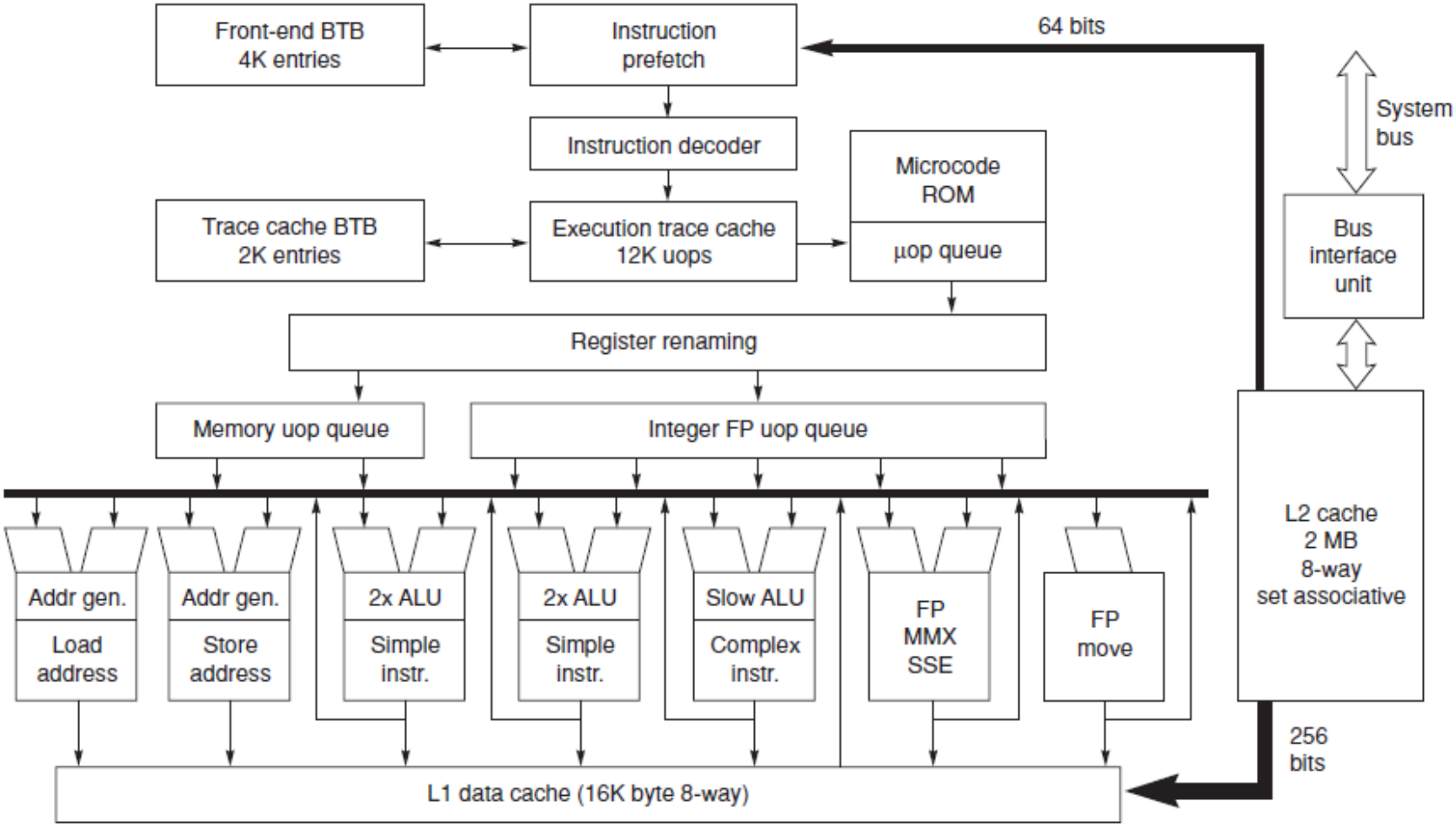


Figure 2.26 The Pentium 4 microarchitecture. The cache sizes represent the Pentium 4 640. Note that the instructions are usually coming from the trace cache; only when the trace cache misses is the front-end instruction prefetch unit consulted. This figure was adapted from Boggs et al. [2004].

L2 Cache in Pentium 4

- With deep pipelines and aggressive clock rates the cost of cache misses and branch mispredictions are both very high.
- A two-level cache is used to minimize the frequency of DRAM accesses.
- Branch prediction is done with a branch-target buffer using a two-level predictor with both local and global branch histories;
  - In the most recent Pentium 4, the size of the branch-target buffer was increased, and the static predictor, used when the branch-target buffer misses, was improved.

Evolution: Pentium 4 640

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

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

An Analysis of the Performance of the Pentium 4

- The deep pipeline of the Pentium 4 makes the use of speculation, and its dependence on branch prediction, critical to achieving high performance.
- Likewise, performance is very dependent on the memory system.
- Because of the importance of branch prediction and cache misses, we focus our attention on these two areas.
- We use five of the integer SPEC CPU2000 benchmarks and five of the FP benchmarks, and the data is captured using counters within the Pentium 4 designed for performance monitoring.
- The processor is a Pentium 4 640 running at 3.2 GHz with an 800 MHz system bus and 667 MHz DDR2 DRAMs for main memory.

Branch mispredictions per instruction

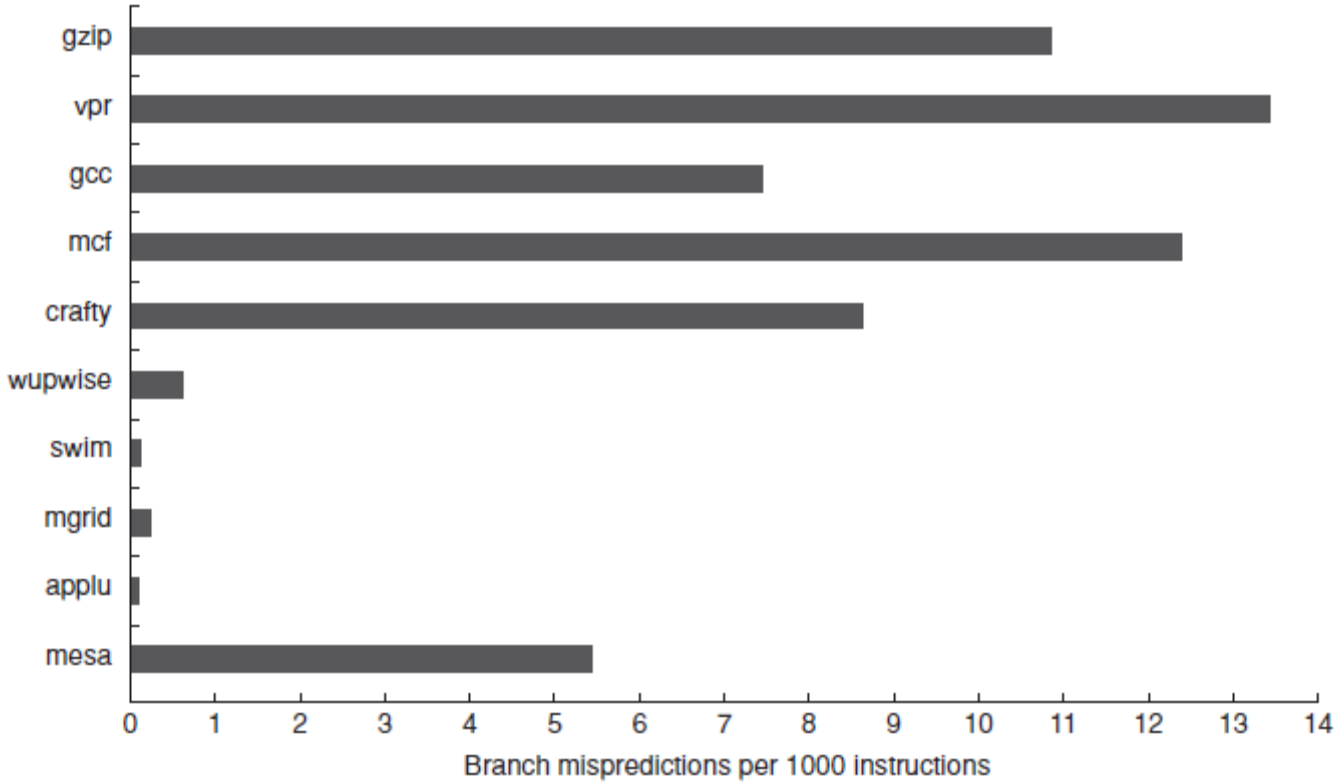


Figure 2.28 Branch misprediction rate per 1000 instructions for five integer and five floating-point benchmarks from the SPEC CPU2000 benchmark suite. This data and the rest of the data in this section were acquired by John Holm and Dileep Bhandarkar of Intel.

The *misprediction rate per instruction* for the integer benchmarks is more than 8 times higher than the rate for the FP benchmarks.

Misspeculation in Pentium 4

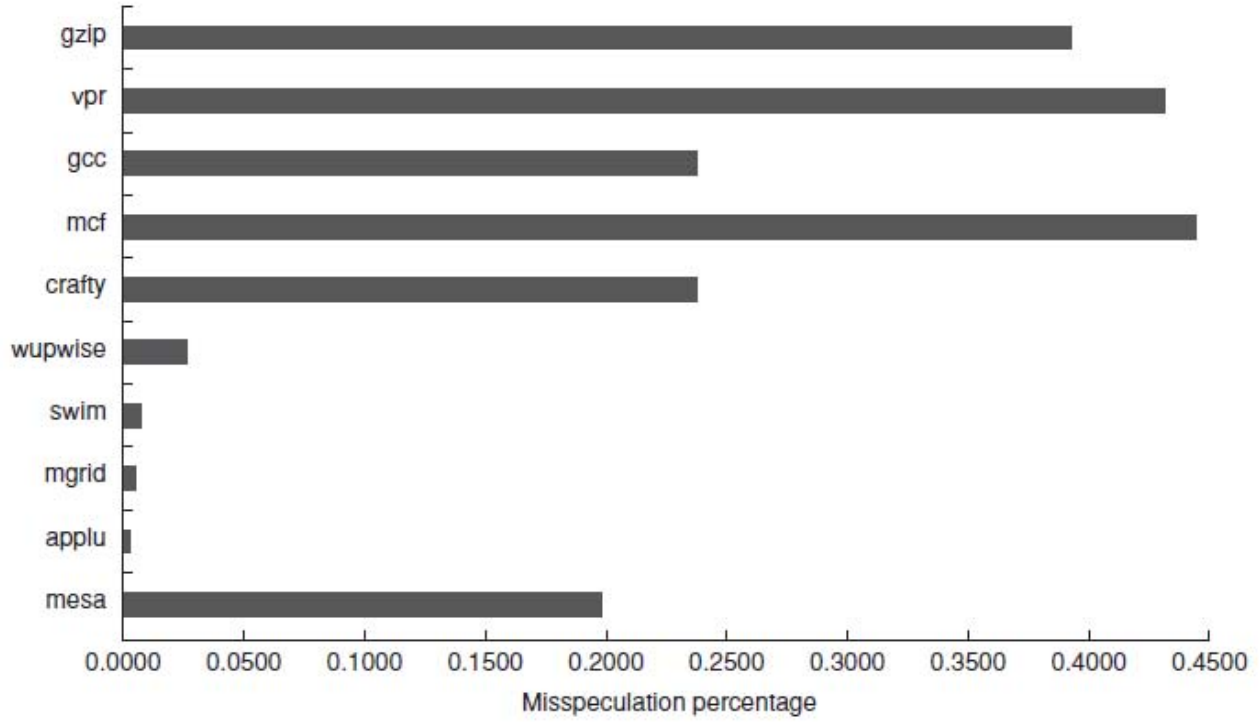


Figure 2.29 The percentage of uop instructions issued that are misspeculated.

Branch-prediction accuracy is crucial in speculative processors, since incorrect speculation requires recovery time and wastes energy pursuing the wrong path. As we would suspect, the misspeculation rate results look almost identical to the misprediction rates.

Cache misses in Pentium 4

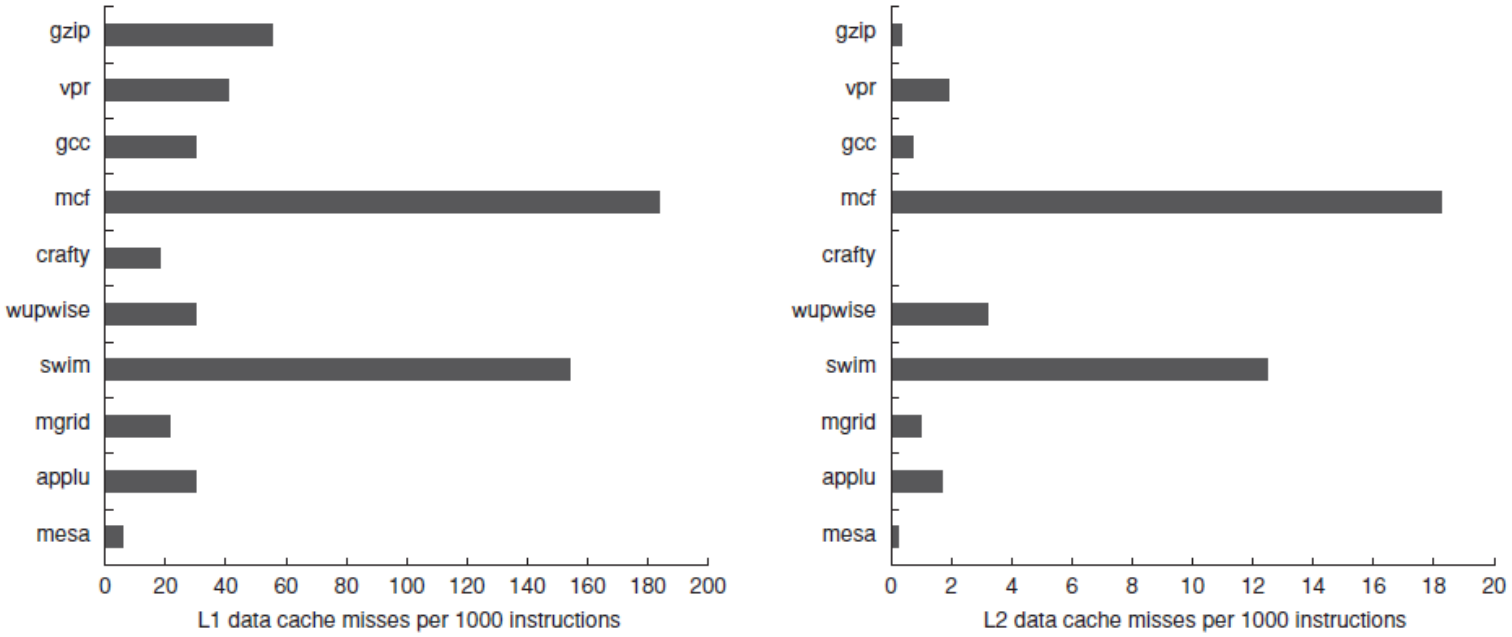


Figure 2.30 **L1 data cache and L2 cache misses per 1000 instructions for 10 SPEC CPU2000 benchmarks.** Note that the scale of the L1 misses is 10 times that of the L2 misses. Because the miss penalty for L2 is likely to be at least 10 times larger than for L1, the relative sizes of the bars are an indication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

Although the miss rate for L1 is about 14 times higher than the miss rate for L2, the miss penalty for L2 is comparably higher, and the inability of the microarchitecture to hide these very long misses means that L2 misses likely are responsible for an equal or greater performance loss than L1 misses.

How do the effects of misspeculation and cache misses translate to actual performance?

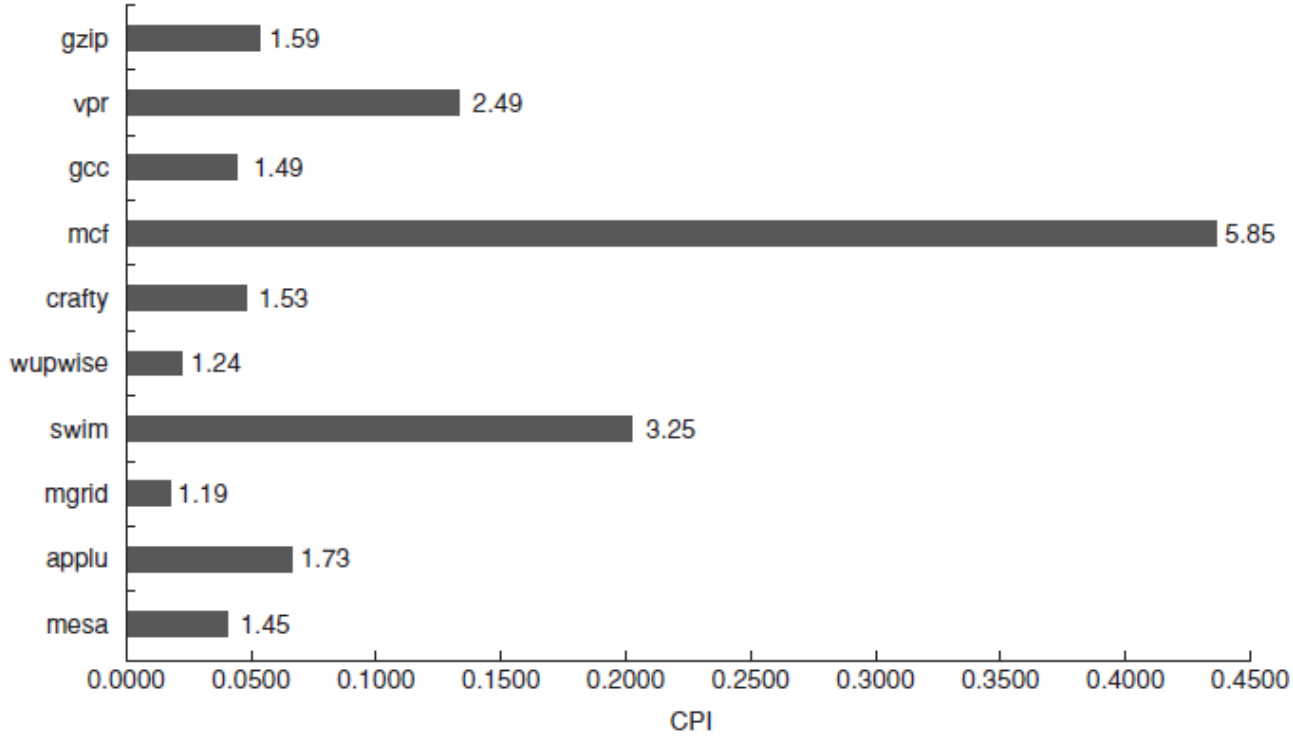


Figure 2.31 The CPI for the 10 SPEC CPU benchmarks. An increase in the CPI by a factor of 1.29 comes from the translation of IA-32 instructions into uops, which results in 1.29 uops per IA-32 instruction on average for these 10 benchmarks.

Let's analyse mcf, vpr and swim: next slide =>

Analyses of benchmarks

- mcf has a CPI that is more than four times higher than that of the four other integer benchmarks.
  - It has the worst misspeculation rate.
  - Equally importantly, mcf has the worst L1 and the worst L2 miss rate among any benchmark, integer or floating point, in the SPEC suite.
  - The high cache miss rates make it impossible for the processor to hide significant amounts of miss latency.
- vpr achieves a CPI that is 1.6 times higher than three of the five integer benchmarks (excluding mcf).
  - This appears to arise from a branch misprediction that is the worst among the integer benchmarks (although not much worse than the average) together with a high L2 miss rate, second only to mcf among the integer benchmarks.

Analyses of benchmarks

- swim is the lowest performing FP benchmark, with a CPI that is more than two times the average of the other four FP benchmarks.
  - swim's problems are high L1 and L2 cache miss rates, second only to mcf.
  - swim has excellent speculation results, but that success can probably not hide the high miss rates, especially in L2.
  - In contrast, several benchmarks with reasonable L1 miss rates and low L2 miss rates (such as mgrid and gzip) perform well.

Pentium 4 and AMD Opteron

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

Pentium 4 Vs AMD Opteron: CPI

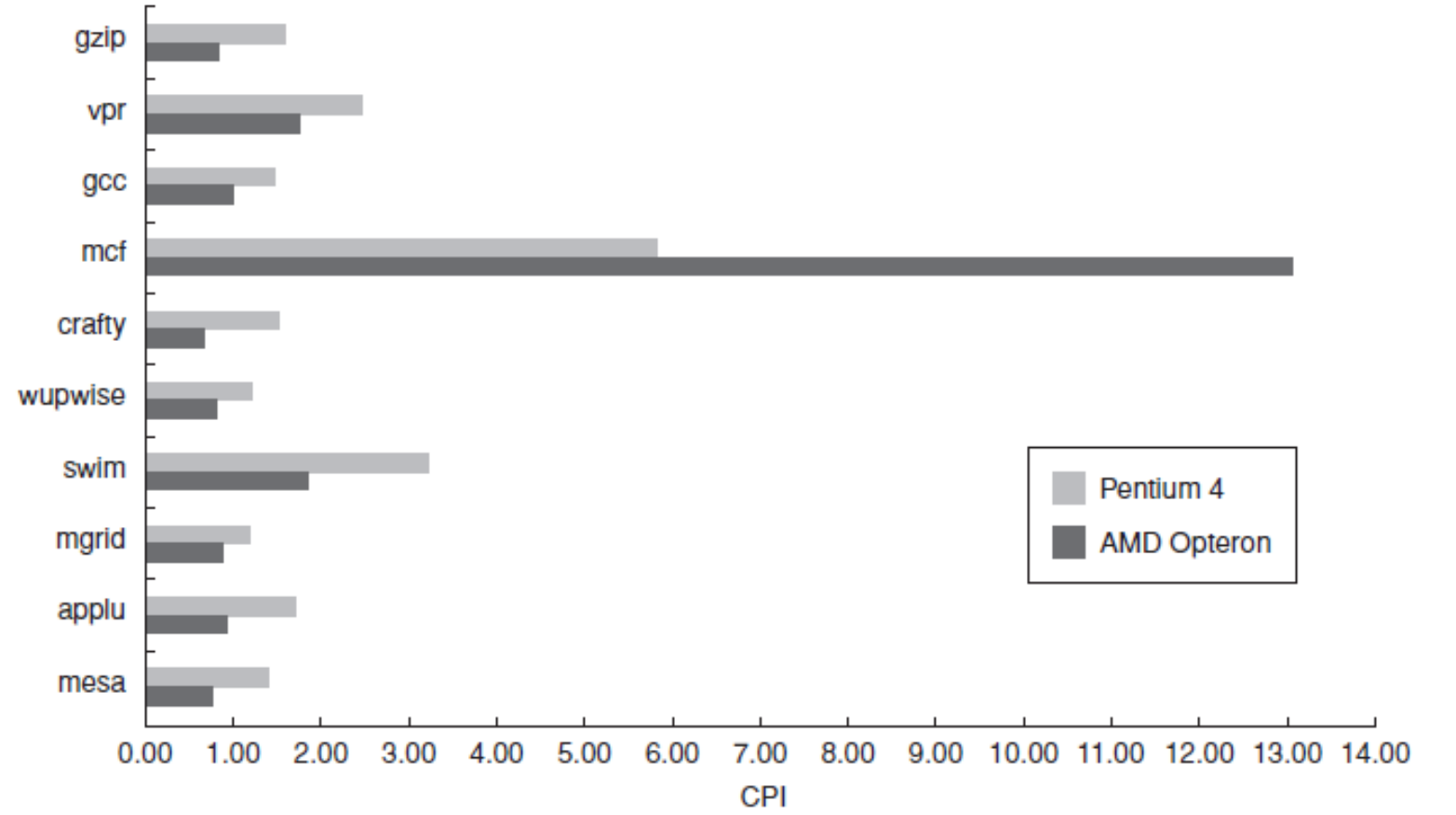


Figure 2.32 A 2.6 GHz AMD Opteron has a lower CPI by a factor of 1.27 versus a 3.2 GHz Pentium 4.

Pentium 4 Vs AMD Opteron: SPECRatio

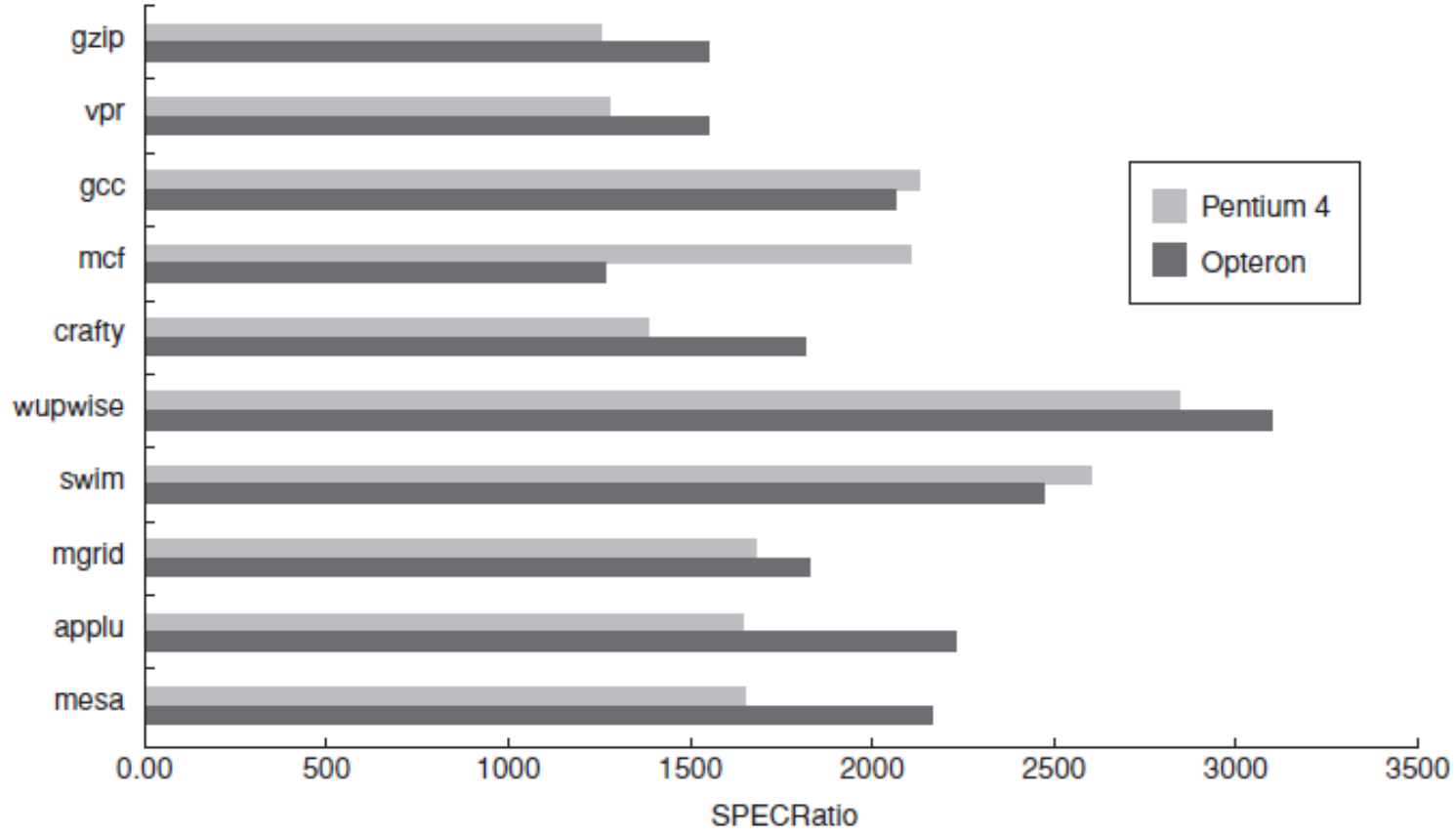


Figure 2.33 The performance of a 2.8 GHz AMD Opteron versus a 3.8 GHz Intel Pentium 4 shows a performance advantage for the Opteron of about 1.08.

Pentium 4 Vs AMD Opteron: Conclusions

- The Opteron is slightly faster, meaning that the higher clock rate of the Pentium 4 is insufficient to overcome the higher CPI arising from more pipeline stalls.
- Hence, while the Pentium 4 performs well, it is clear that the attempt to achieve both high clock rates via a deep pipeline and high instruction throughput via multiple issue is not as successful as the designers once believed it would be.

Outline

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and the Algorithm
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Putting It All Together: The Intel Pentium 4
- Fallacies and Pitfalls

Fallacies and Pitfalls

- **Fallacy** *Processors with lower CPIs will always be faster.*
- **Fallacy** *Processors with faster clock rates will always be faster.*
- Although a lower CPI is certainly better, sophisticated multiple-issue pipelines typically have slower clock rates than processors with simple pipelines.
- In applications with limited ILP or where the parallelism cannot be exploited by the hardware resources, the faster clock rate often wins.
- But, when significant ILP exists, a processor that exploits lots of ILP may be better.

IBM Power5

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

Thus, the Power5 will be faster if the product of the instruction count and CPI is less than one-half the same product for the Pentium 4.

No one wins

- The CPI × instruction count advantages of the Power5 are significant for the FP programs, sometimes by more than a factor of 2,
- While for the integer programs the CPI × instruction count advantage of the Power5 is usually not enough to overcome the clock rate advantage of the Pentium 4.
- By comparing the SPEC numbers, we find that the product of instruction count and CPI advantage for the Power5 is 3.1 times on the floating-point programs but only 1.5 times on the integer programs.
- Because the maximum clock rate of the Pentium 4 in 2005 is exactly twice that of the Power5, the Power5 is faster by 1.5 on SPECfp2000 and the Pentium 4 will be faster by 1.3 on SPECint2000.

Intel Pentium 4 Vs. IBM Power5

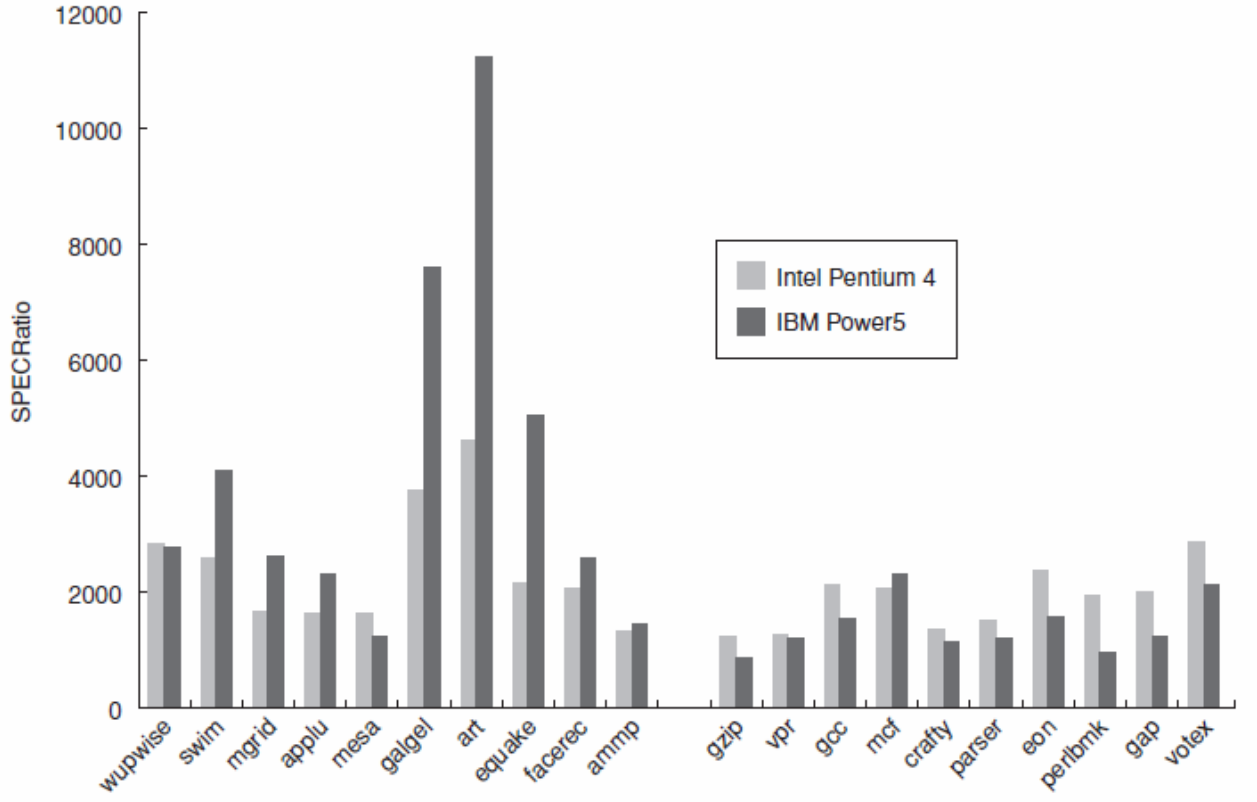


Figure 2.34 A comparison of the 1.9 GHZ IBM Power5 processor versus the 3.8 GHz Intel Pentium 4 for 20 SPEC benchmarks (10 integer on the left and 10 floating point on the right) shows that the higher clock Pentium 4 is generally faster for the integer workload, while the lower CPI Power5 is usually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

Pitfall: Sometimes bigger and dumber is better

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

Pitfall: Sometimes bigger and dumber is better

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

End of Lecture 3

- Readings
  - Book: Chapter 2