

Chapter 3

Instruction-Level Parallelism and Its Exploitation

ILP: Concepts and Challenges

- All processors since about 1985 use pipelining to improve performance
- The potential overlap among instructions is called ILP
- Two approaches to exploit ILP
 - Dynamic approach depend on HW
 - Static approach depend on SW
- Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

Major Techniques to Reduce CPI

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	A.2
Delayed branches and simple branch scheduling	Control hazard stalls	A.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	A.8
Dynamic scheduling with renaming	Data hazard stalls and stalls from antidependences and output dependences	3.2
Dynamic branch prediction	Control stalls	3.4
Issuing multiple instructions per cycle	Ideal CPI	3.6
Speculation	Data hazard and control hazard stalls	3.5
Dynamic memory disambiguation	Data hazard stalls with memory	3.2, 3.7
Loop unrolling	Control hazard stalls	4.1
Basic compiler pipeline scheduling	Data hazard stalls	A.2, 4.1
Compiler dependence analysis	Ideal CPI, data hazard stalls	4.4
Software pipelining, trace scheduling	Ideal CPI, data hazard stalls	4.3
Compiler speculation	Ideal CPI, data, control stalls	4.4

What Is ILP?

- ILP in a basic block is quite small
- It's better to exploit ILP across basic blocks
- Loop-level parallelism
 - For (i=1;I<=1000;i=i+1)
 $x[i] = x[i] + y[i]$
- Vector instructions can exploit loop-level parallelism

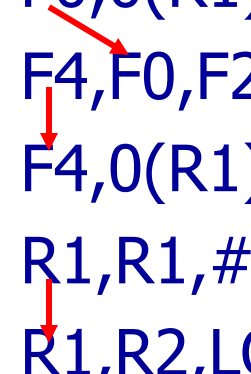
Three Types of Dependences

- Data dependences
- Name dependences
- Control dependences

Data Dependences

- An instruction j is *data dependent* on instruction i if either of the following holds
 - i produces a result used by j
 - j is data dependent on k , k is data dependent on i
- Example

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



Data Dependences (Cont'd)

- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependences
 - indicate the possibility of a hazard
 - determine the order of calculating the results
 - set a upper bound of parallelism
- Data Dependences can be overcome in two ways
 - Maintain the dependence but avoid a hazard
 - Eliminate them by transforming the code
- Data Dependences may flow through registers or memory

Name Dependences

- 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
- Two types of name dependences between instruction i preceding instruction j in program order
 - Antidependence
 - Output dependence

Antidependence Example

add r1, r2, r3

sub r2, r4, r5



- sub is antidependent on the add

Output Dependence Example

add r1, r2, r3

sub r1, r4, r5

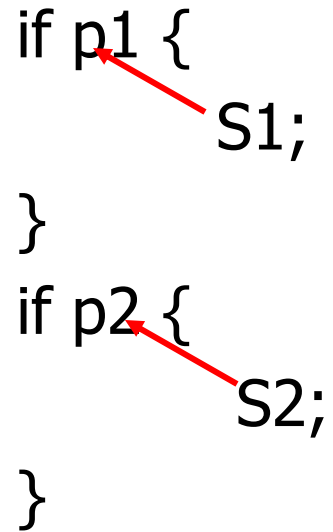
- The sub is output dependent on add

Data Hazards

- A hazard occurs whenever dependences exist between instructions
- Three types of data hazards
 - RAW (read after write): akin to data dependency
 - WAW (write after write): akin to output dependency
 - WAR (write after read): akin to anti dependency

Control Dependences

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

A diagram illustrating control dependence. In the code snippet, a red arrow points from the condition 'p1' to the statement 'S1;'. Another red arrow points from the condition 'p2' to the statement 'S2;'. This indicates that the execution of S1 is controlled by p1, and the execution of S2 is controlled by p2.

- Control dependence is preserved by
 - Program order
 - Branch detection

Control Dependences - Example

DADDU R1,R2,R3

BEQZ R4,L

DSUBU R1,R5,R6

L: ...

OR R7,R1,R8

Control Dependences - Speculation

DADDU R1,R2,R3

BEQZ R12,skipnext

DSUBU R4,R5,R6

DADDU R5,R4,R9

skipnext: OR R7,R8,R9

- If R4 in DSUBU is unused, we can move it before BEQZ

Basic Compiler Techniques for Exposing ILP

- To keep a pipeline full, ILP must be exploited by finding independent 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 on
 - ILP exploiting
 - the latencies in the pipeline

Latencies of FP operations

Assume:

Branch delay: 1 cycle

FP load to store : no delay (forwarding)

Int load delay: 1 cycle;

Int ALU: no delay

No structural hazards

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

Example

for (i=1000; i>0; i=i-1)	Loop:	L.D F0,0(R1)	ADD.D
x[i] = x[i] + s;		F4,F0,F2	
		S.D F4,0(R1)	
		DADDUI R1,R1,#-8	
		BNE R1,R2,Loop	

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

Answer Without Any Scheduling

			<u>Clock cycle issued</u>
Loop:	L.D	F0,0(R1)	1
	<i>stall</i>		2
	ADD.D	F4,F0,F2	3
	<i>stall</i>		4
	<i>stall</i>		5
	S.D	F4,0(R1)	6
	DADDUI	R1,R1,#-8	7
	<i>stall</i>		8
	ENE	R1,R2,Loop	9
	<i>stall</i>		10

Answer With Scheduling

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

Loop Unrolling

- Loop unrolling is a simple scheme for increasing ILP
- Unrolling simply replicates the loop body multiple times, adjusting the loop termination code
- Loop unrolling can eliminate the branch and allow instructions from different iterations to be scheduled together

Loop Unrolling (Cont'd)

- Suppose the upper bound of the loop is n and we would like to unroll the loop to make k copies of the body
 - The first executes $(n \bmod k)$ times and has a body that is the original loop
 - The second is the unrolled body surrounded by an outer loop that iterates (n/k) times
- For large values of n , most of the execution time will be spent in the unrolled loop body

Example

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

Answer

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

Example

- Show the unrolled loop in the previous example after it has been scheduled for the pipeline with the latencies shown below.

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

Answer

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

Summary of the Loop Unrolling and Scheduling Example

- The key to most of these techniques is to know when and how the ordering among instructions may be changed
- Some assumptions are made to obtain the final unrolled code
 - Determine that it was legal to move the S.D after the DADDUI and BNE, and find the amount to adjust the S.D offset
 - Determine loop unrolling loop would be useful
 - Use different registers to avoid unnecessary constraints
 - 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
 - Schedule the code, preserving any dependences needed to yield the same result as the original code

Example

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

Answer

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


The diagram illustrates the control flow of the assembly code. It shows four instances of the instruction 'DADDUI R1, R1, #-8 ;drop BNE'. From the 'drop BNE' text of each of these four instructions, a black arrow points to the 'BNE R1, R2, LOOP' instruction at the bottom of the code block, indicating that the loop is repeated four times.

Example

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

Answer

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



Answer (Cont'd)

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

Three Limits Against Loop Unrolling

- A decrease in the amount of overhead amortized with each unroll
- Code size limitations
- Compiler limitations

Loop Overhead

- 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 fourteen clock cycles, only two cycles were loop overhead:
 - the DSUBI, which maintains the index value
 - the BNE, which terminates the loop
- If the loop is unrolled eight times, the overhead is reduced from $1/2$ cycle per original iteration to $1/4$

Code Size Limitations

- For larger loops, the code size growth may be a concern
 - either in the embedded space where memory may be at a premium
 - or if the larger code size causes a decrease in the instruction cache miss rate

Compiler Limitations

- Code size is the potential shortfall in registers created by aggressive unrolling and scheduling
- Register pressure
 - It arises because scheduling code to increase ILP causes the number of live values to increase
 - After aggressive instruction scheduling, it not be possible to allocate all the live values to registers
- Without unrolling, aggressive scheduling is sufficiently limited by branches so that register pressure is rarely a problem

Static Branch Prediction

- 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
- Delayed branches expose a pipeline hazard so that the compiler can reduce the penalty associated with the hazard
- Being able to accurately predict a branch at compile time is also helpful for scheduling data hazards

Example

LD R1,0(R2)

DSUBU R1,R1,R3

BEQZ R1,L

OR R4,R5,R6

DADDU R10,R4,R3

L: DADDU R7,R8,R9

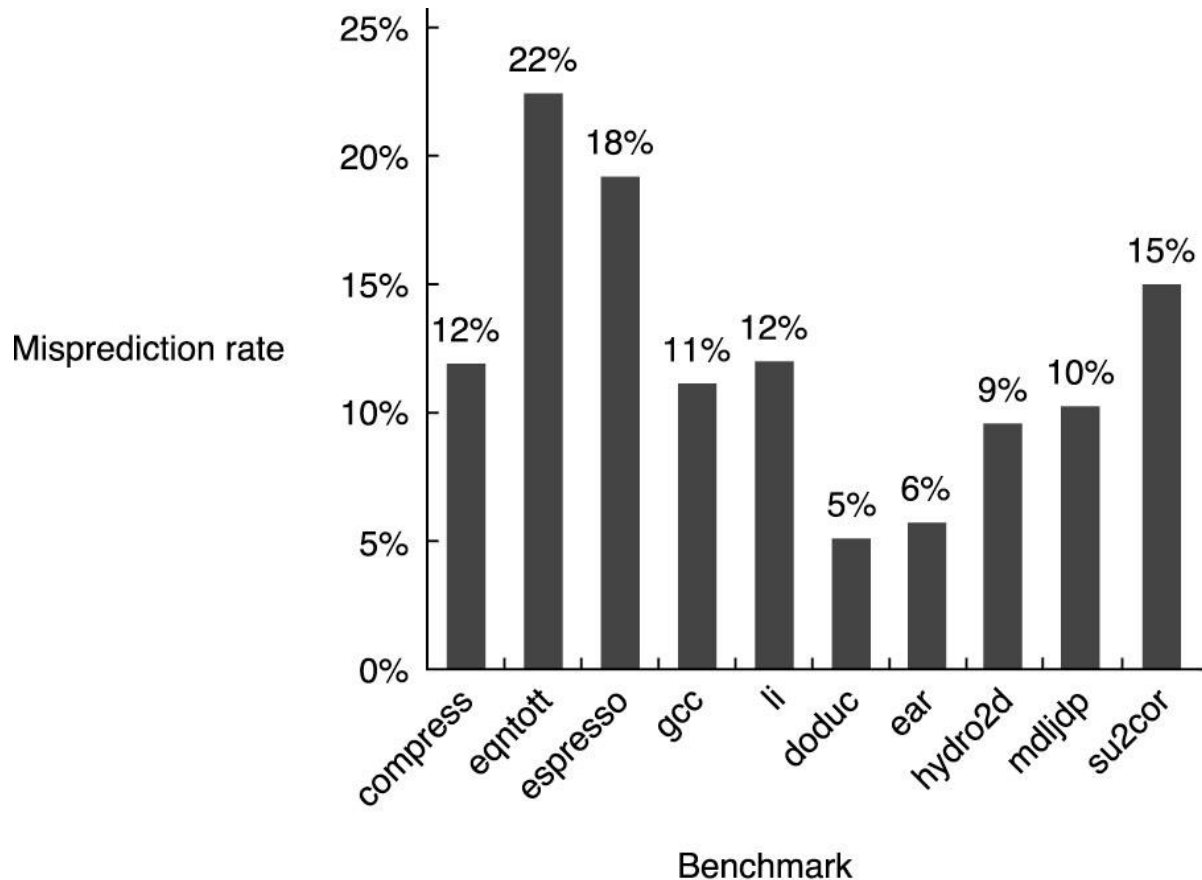
Static Prediction Method 1

- The simplest scheme is to predict a branch as taken
- This scheme has an average misprediction rate that is equal to the untaken branch frequency, which for the SPEC programs is 34%
- Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%).

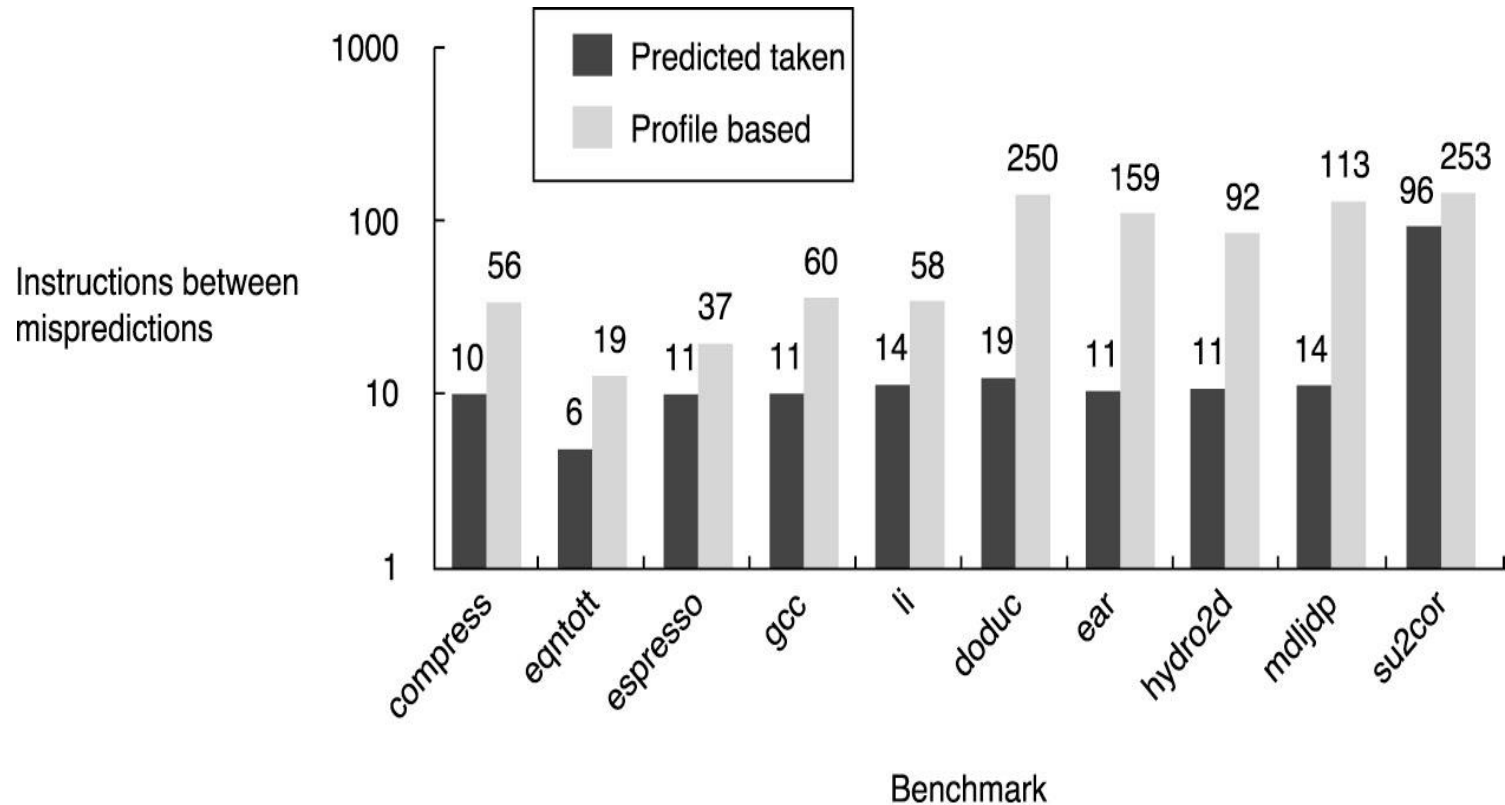
Static Prediction Method 2

- A still more accurate technique is to predict branches on the basis of profile information collected from earlier runs
- The behavior of branches is often bimodally distributed

Static Prediction Method 2 (Cont'd)



Accuracy of a Predict-Taken Strategy and a Profile-Based Predictor for SPEC92



Reducing Branch Costs with Dynamic Hardware Prediction

- As the amount of ILP we attempt to exploit grows, control dependences rapidly become the limiting factor
- Branches probably be predicted on an n-issue processor
- Amdahl's Law reminds us that relative impact of the control stalls will be larger with the lower potential CPI in such machines

Basic Branch Prediction

- Static schemes : predict not taken and delayed branch
- Dynamic schemes: the prediction will depend on the behavior of the branch at runtime
- The goal of all these mechanisms is to allow the processor to resolve the outcome of a branch early
- It prevents control dependences from causing stalls

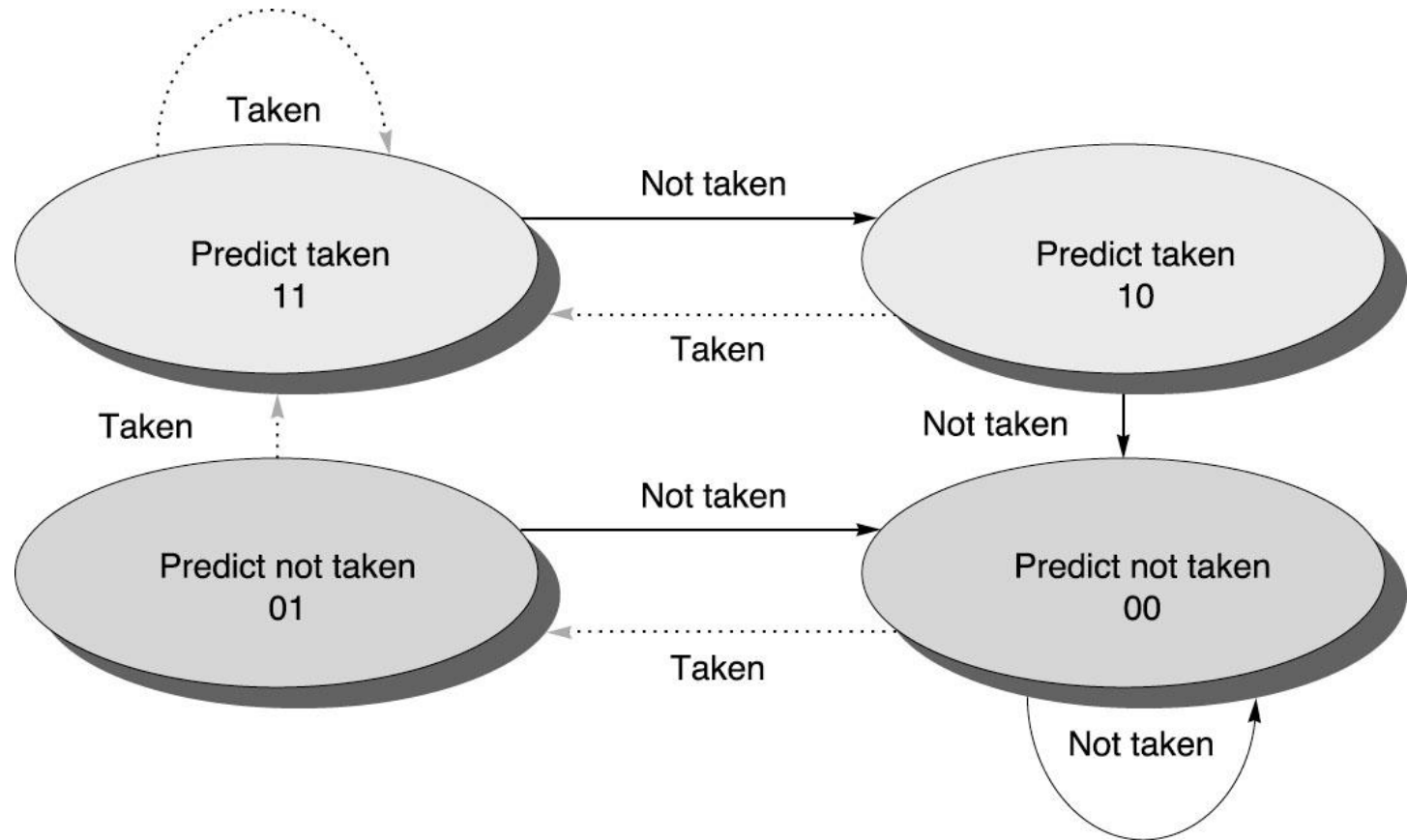
Branch-Prediction Buffers

- A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction
- The memory contains a bit that says whether the branch was recently taken or not
- It has no tags and is useful only to reduce the branch delay
- The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction
- If the hint is wrong, the prediction bit is inverted and stored back

1-Bit Prediction Scheme

- **Example:** Consider a loop branch whose behavior is taken nine times in a row, then not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?
- **Answer:** the prediction accuracy for this branch that is taken 90% of the time is only 80% (two incorrect predictions and eight correct ones)
- 1-bit predictor only updates the prediction bit on a mispredict

States in 2-Bit Prediction Scheme



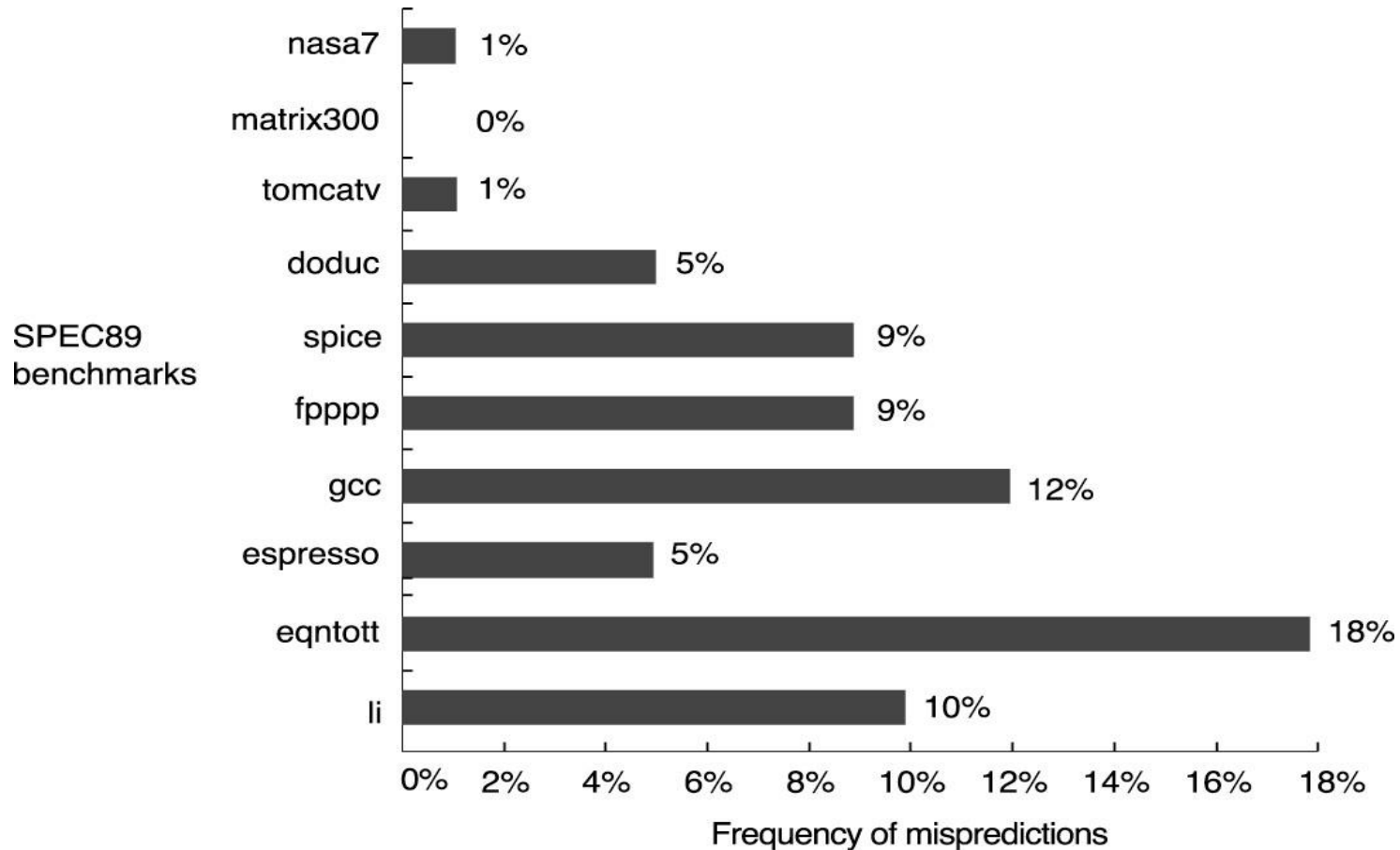
2-Bit Prediction Scheme

- With an n-bit counter, when the counter is greater than or equal to one half of its maximum value, the branch is predicted as taken; otherwise, it is predicted untaken
- In the 2-bit scheme, the counter is incremented on a taken branch and decremented on an untaken branch
- One complication of the 2-bit scheme is that it updates the prediction bits more often than a 1-bit
- Since we typically read the prediction bits on every cycle, a two-bit predictor will typically need both a read and a write access port

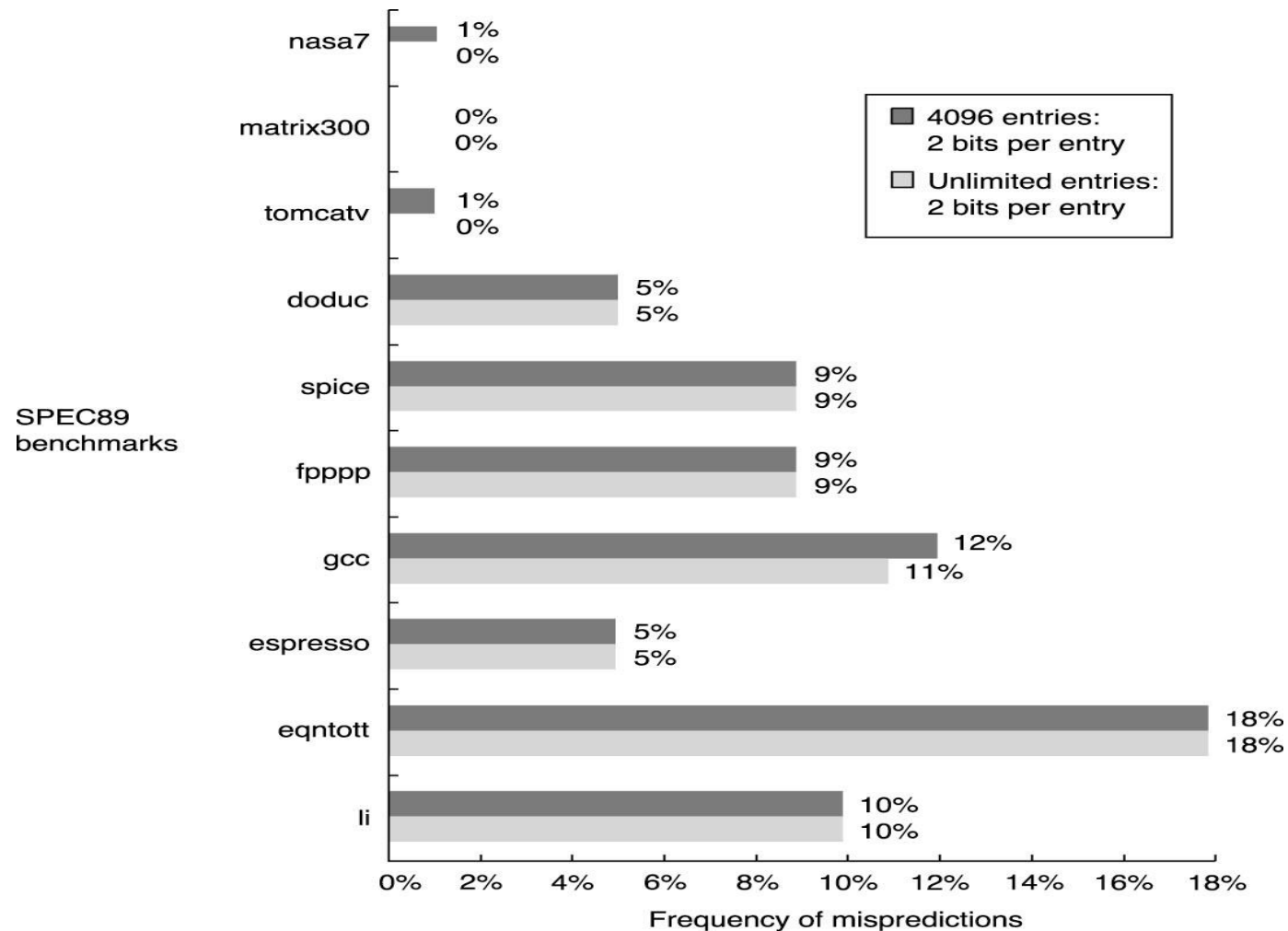
2-Bit Prediction Scheme (Cont'd)

- The branch frequency is also taken into account, since the importance of accurate prediction is larger in programs with higher branch frequency
- It is clear that the hit rate of the buffer is not the limiting factor
- Simply increasing the number of bits per predictor without changing the predictor structure also has little impact

Prediction Accuracy of 4k-Entry 2-Bit Prediction Buffer for SPEC89



Prediction Accuracy of 4k-Entry 2-Bit Prediction Buffer VS Infinite Buffer



Example

if (aa==2)		DSUBUI R3,R1,#2
aa=0;		BNEZ R3,L1 ; branch b1 (aa!=2)
if (bb==2)		DADD R1,R0,R0 ; aa=0
bb=0;	L1:	DSUBUI R3,R2,#2
if (aa!=bb) {		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)

Correlating Branch Predictors

- The behavior of branch b3 is correlated with the behavior of branches b1 and b2
- A predictor using only a single branch to predict the outcome 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*

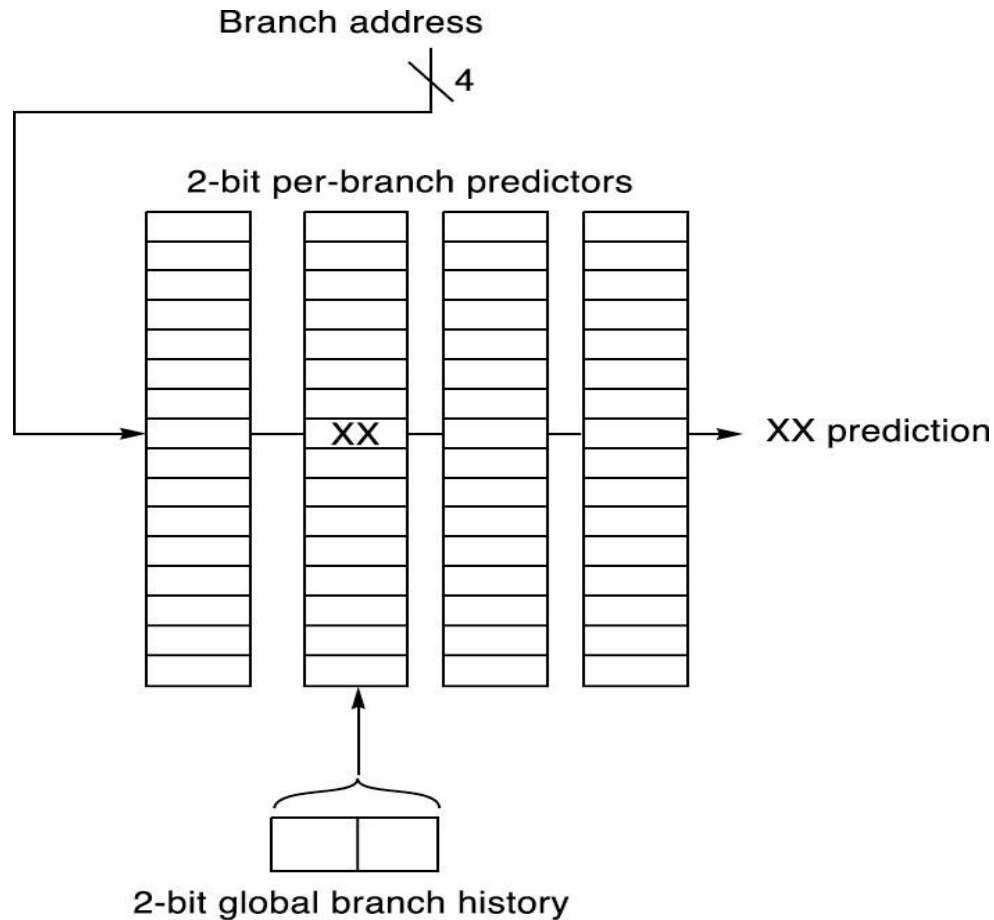
Correlating Branch Predictors - Example

if (d==0)		BNEZ R1,L1; branch b1(d!=0)
d=1;		DADDIU R1,R0,#1; d==0, so d=1
if (d==1)	L1:	DADDIU R3,R1,#-1
		BNEZ R3,L2; branch b2(d!=1)
		...
	L2:	

Correlating Branch Predictors (Cont'd)

- Correlating branch predictor can yield higher prediction rates than the two-bit scheme
- It requires only a trivial amount of additional hardware
- The simplicity of the hardware comes from a simple observation: The global history of the most recent m branches can be recorded in an m -bit shift register, where each bit records whether the branch was taken or not taken
- The branch-prediction buffer can be indexed using a concatenation of the low-order bits from the branch address with the m -bit global history₅₄

A (2,2) Branch-Prediction Buffer



The Number of Bits in An (m,n) Predictor

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

Example 4

- **Example:** How many bits are in the (0,2) branch predictor we examined earlier? How many bits are in the (2,2) branch predictor?
- **Answer:**

The earlier predictor had 4K entries selected by the branch address. Thus the total number of bits is

$$2^0 \times 2 \times 4K = 8K$$

The (2,2) predictor has

$$2^2 \times 2 \times 16 = 128 \text{ bits.}$$

Example 5

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

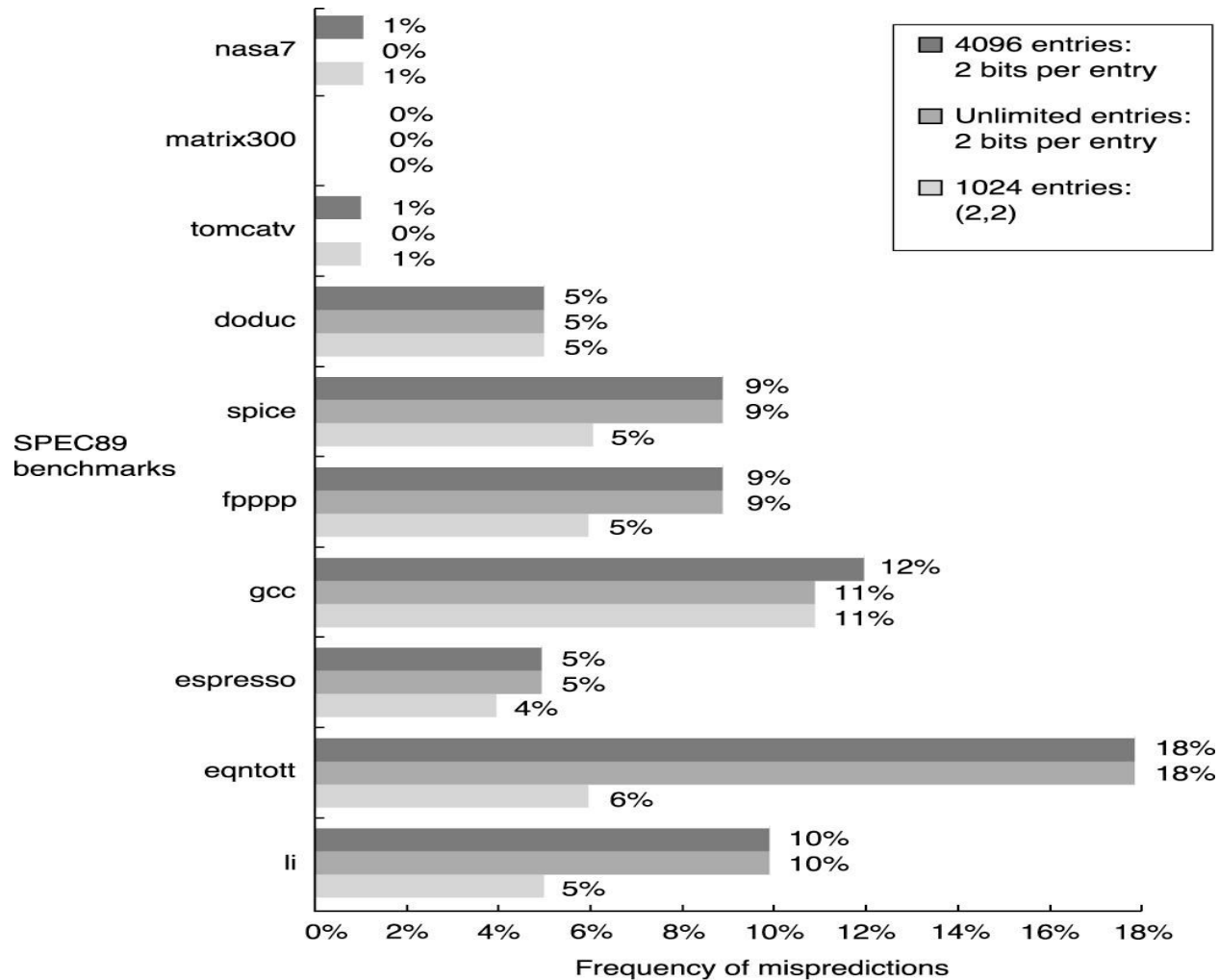
We know that

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

Hence

Number of prediction entries selected by the branch
= 1K

Comparison of 2-Bit Predictors



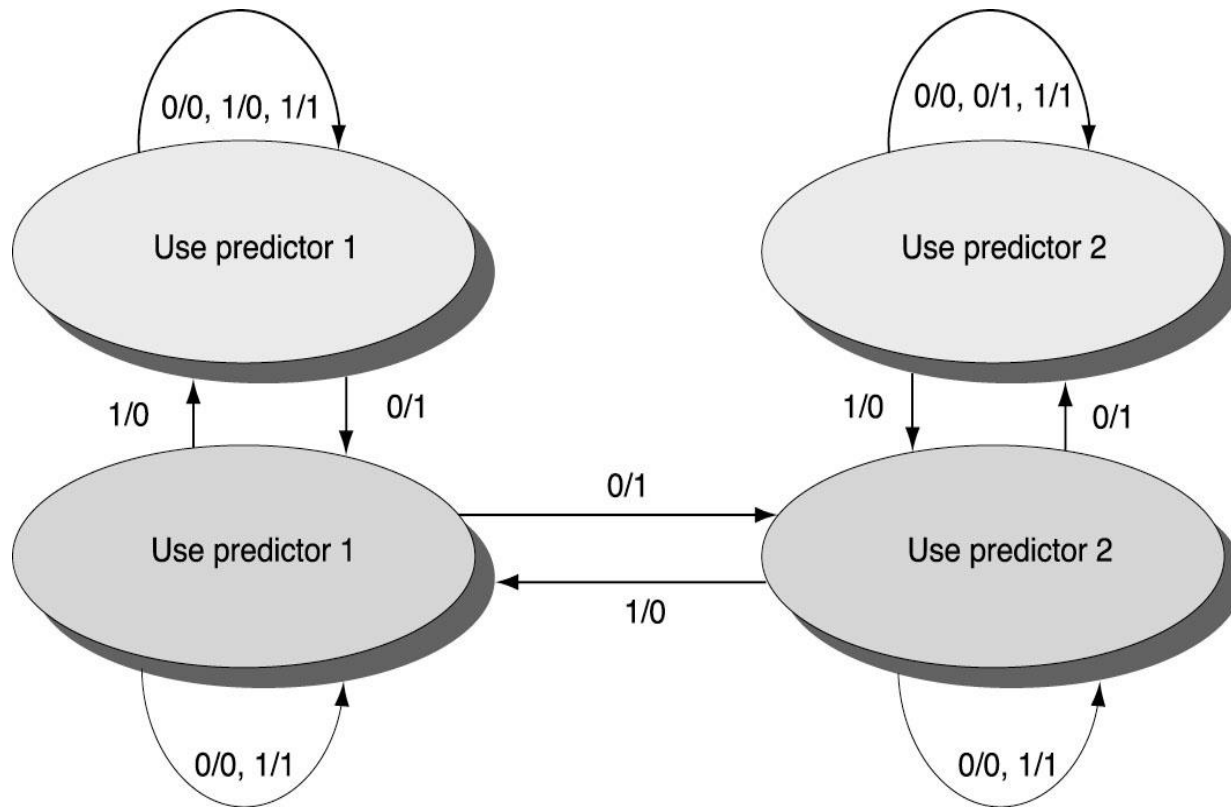
Tournament Predictors

- Correlating branch predictors uses only local information failed on some important branches
- Tournament predictors add global information to improve the performance
- Tournament uses multiple predictors, usually one based on global information and one based on local information, and combining them with a selector
- Tournament predictors can achieve both better accuracy at medium sizes (8Kb-32Kb) and also make use of very large numbers of prediction bits effectively

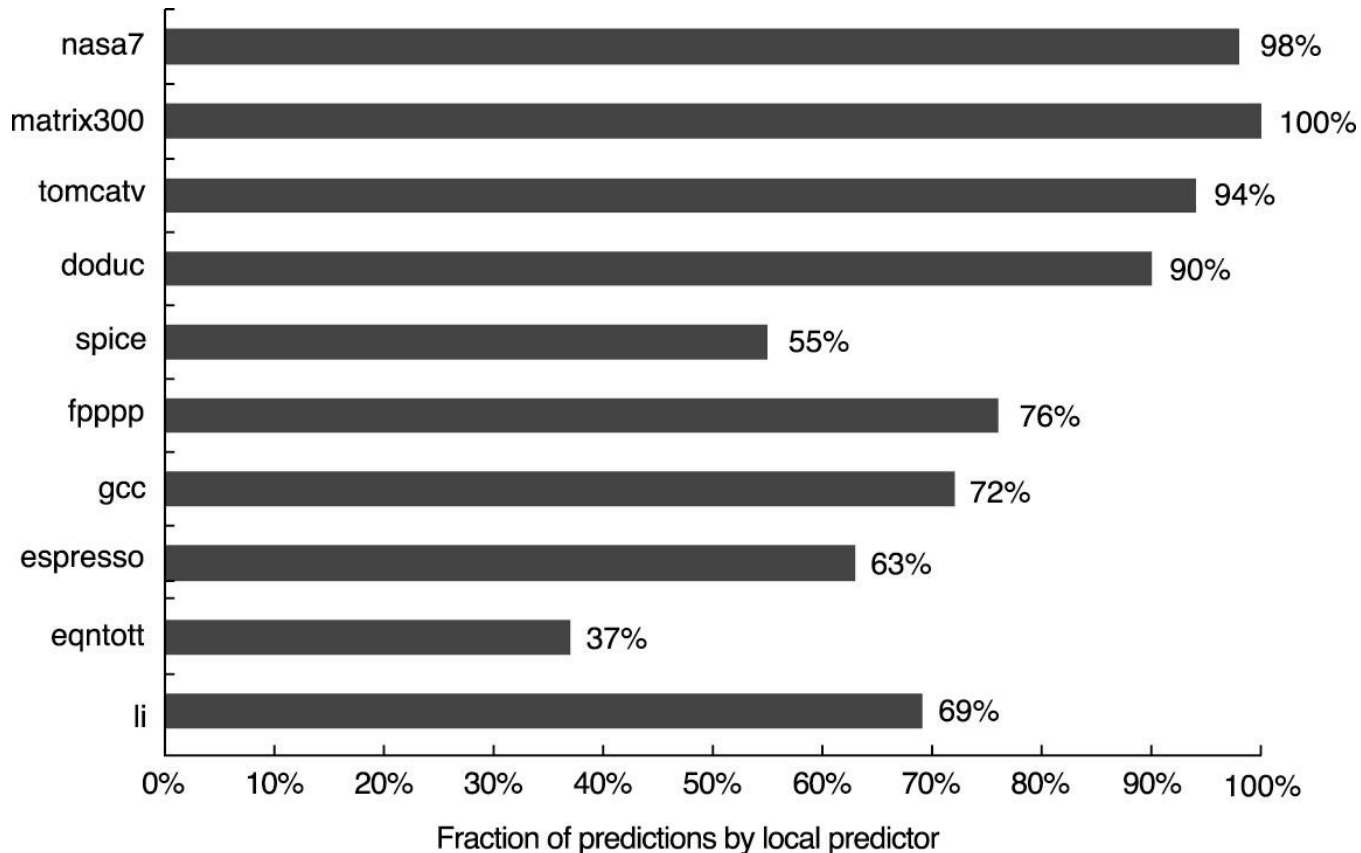
Tournament Predictors (Cont'd)

- A multilevel branch predictor use several levels of branch prediction tables together with an algorithm for choosing among the multiple predictors
- Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors
- The four states of the counter dictate whether to use predictor 1 or predictor 2
- The advantage of a tournament predictor is its ability to select the right predictor for the right branch

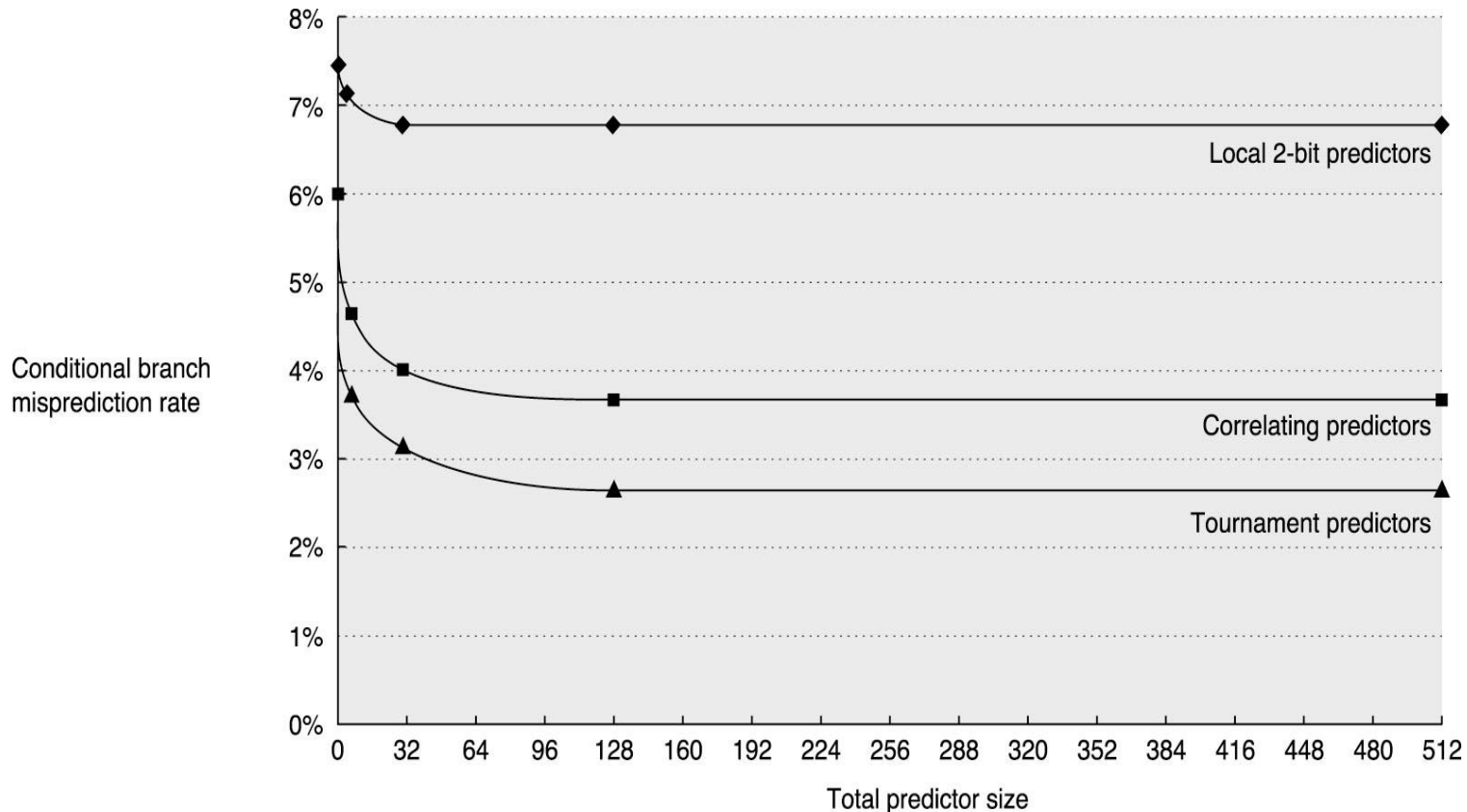
An Example: the Alpha 21264 Branch Predictor



Predictions from the Local Predictor of Tournament Predictor Using the SPEC89



The Misprediction Rate for Three Different Predictors



Dynamic Scheduling

- The hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior
- Advantages:
 - Handle some cases at run time
 - Simplify the compiler
- It's gained at a cost of a significant increase in hardware complexity
- Static pipeline scheduling by the compiler tries to minimize stalls by separating dependent instructions so that they will not lead to hazards

Dynamic Scheduling: The Idea

- Instructions are issued in program order and if an instruction is stalled in the pipeline, no later instructions can proceed
- 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

Dynamic Scheduling – Example 1

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
- SUB.D is not data dependent on anything in the pipeline
- This hazard creates a performance limitation

Out-of-order execution

- Out-of-order execution implies out-of-order completion
- Example:

DIV.D F0,F2,F4

ADD.D F6,F0,F8

SUB.D F8,F10,F14

MULT.D F6,F10,F8

- There is an antidependence between the ADD.D and the SUB.D, yielding a WAR hazard
- There exists an output dependencies, the write of F6 by MULT.D, yielding WAW hazards

Out-of-order execution (Cont'd)

- Dynamic scheduling with out-of-order completion must preserve exception behavior
- To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages:
 - Issue—Decode instructions, check for structural hazards.
 - Read operands—Wait until no data hazards, then read operands

Imprecise Exceptions

- An exception is imprecise if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order
- Imprecise exceptions make it difficult to restart execution after an exception
- Two possibilities to cause imprecise exceptions:
 - The pipeline may have already completed instructions that are later in program order than the instruction causing the exception
 - The pipeline may have not yet completed some instructions that are earlier in program order than the instruction causing the exception

Dynamic Scheduling Using Tomasulo's Approach

- The objective is to minimize RAW hazards, and use register renaming to avoid WAW and WAR hazards
- RAW hazards are avoided by executing an instruction only when its operands are available
- Register renaming allows that the out-of-order write does not affect any instructions that depend on an earlier value of an operand

Register Renaming

- Before the renaming

DIV.D F0,F2,F4

ADD.D F6,F0,F8

S.D F6,0(R1)

SUB.D F8,F10,F14

MULT.D F6,F10,F8

- After the renaming

DIV.D F0,F2,F4

ADD.D S,F0,F8

S.D S,0(R1)

SUB.D T,F10,F14

MULT.D F6,F10,T

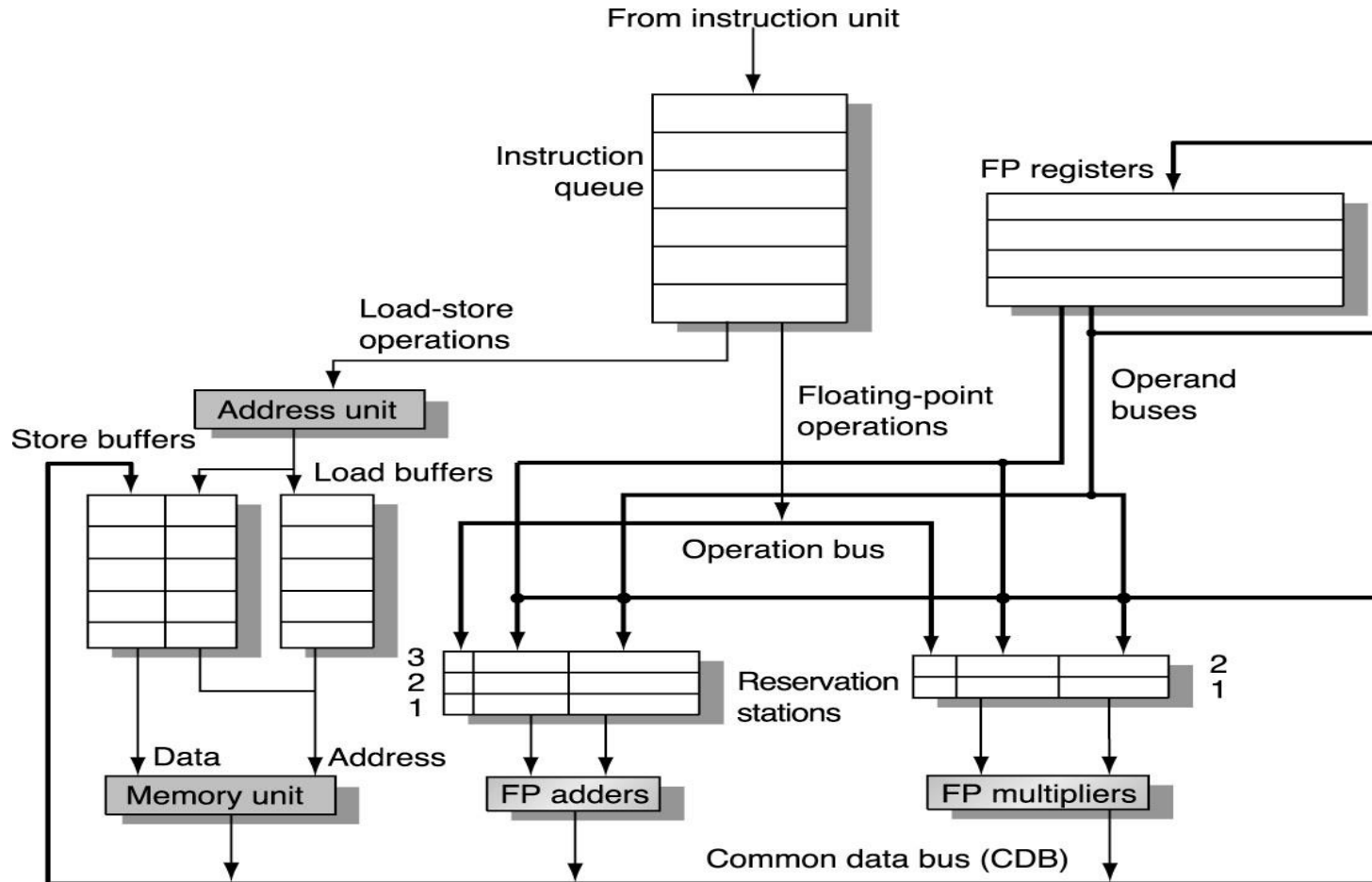
Dynamic Scheduling Using Tomasulo's Approach (Cont'd)

- In Tomasulo's scheme, register renaming is provided by the reservation stations
- 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
- Pending instructions designate the reservation station that will provide their inputs
- Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register

Dynamic Scheduling Using Tomasulo's Approach (Cont'd)

- As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station
- The number of reservation stations is greater than real registers, leading to two other important properties:
 - Hazard detection and execution control are distributed
 - Results are passed directly to functional units from the reservation stations where they are buffered

The Basic Structure of a MIPS Floating Point Unit Using Tomasulo's Algorithm



Three Pipeline Stages

- Issue
- Execute
- Write result

Issue Stage

- Get the next instruction from the head of the instruction queue
- If there is a matching reservation station that is empty, issue the instruction to the station with the operand values, if they are currently in the registers
- If there is not an empty reservation station, then there is a structural hazard and the instruction stalls until a station or buffer is freed
- If the operands are not in the registers, keep track of the functional units that will produce the operands
- This step renames registers, eliminating WAR and WAW hazards

Execute Stage

- If one or more of the operands is not yet available, monitor the CDB while waiting for it to be computed
- When an operand becomes available, it is placed into the corresponding reservation station
- When all the operands are available, the operation can be executed at the corresponding functional unit
- By delaying instruction execution until the operands are available, RAW hazards are avoided

Execute Stage (Cont'd)

- If more than one instruction is ready for a single functional unit, the unit will have to choose among them
- Loads and stores require a two-step execution process
- The first step computes the effective address and then places it in the load or store buffer. Loads in the load buffer execute as soon as the memory unit is available
- Stores wait from the value to be stored before being sent to the memory unit
- Loads and stores are maintained in program order through the *effective address calculation*, which will help to prevent hazards through memory

Execute Stage (Cont'd)

- To preserve exception behavior, no instruction is allowed to initiate execution until all branches that precede the instruction in program order have completed
- This restriction guarantees that an instruction that causes an exception during execution really would have been executed
- In a processor using branch prediction, this means that the processor must know that the branch prediction was correct before allowing an instruction after the branch to begin execution

Write Result

- When the result is available, write it on the CDB and from there into the registers and into any reservation stations waiting for this result
- When both the address and data value are available, they are sent to the memory unit and the store completes

Fields in Reservation Stations

- **Op**: The operation to perform on source operands
- **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,
- **Vj, Vk**: The value of the source operands. For loads, the Vk field is used to the offset from the instruction
- **A**: It's 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:
 - **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

Dynamic Scheduling – Example 2

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

1. L.D F6,34(R2)

2. L.D F2,45(R3)

3. MUL.D F0,F2,F4

4. SUB.D F8,F2,F6

5. DIV.D F10,F0,F6

6. ADD.D F6,F8,F2

Answer of Example 2

Instruction		Instruction status		
		Issue	Execute	Write result
L.D	F6, 34(R2)	✓	✓	✓
L.D	F2, 45(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			

Advantages of Tomasulo's Scheme

- Tomasulo's scheme offers two major advantages over earlier and simpler schemes:
 - The distribution of the hazard detection logic
 - The elimination of stalls for WAW and WAR hazards

The Distribution of the Hazard Detection Logic

- It's arising from the distributed reservation stations and the use of the CDB
- If multiple instructions are waiting on a single result, and each instruction already has its other operand, then the instructions can be released simultaneously by the broadcast on the CDB

The elimination of stalls for WAW and WAR hazards

- The elimination of WAW and WAR hazards, is accomplished by
 - renaming registers using the reservation stations
 - the process of storing operands into the reservation station as soon as they are available
- For example, in WAR hazard between DIV.D and the ADD.D. The hazard is eliminated in one of two ways
 - 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
 - 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

Example 3

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

Answer of Example 3

Instruction	Instruction status		
	Issue	Execute	Write result
L.D F6, 34(R2)	✓	✓	✓
L.D F2, 45(R3)	✓	✓	✓
MUL.D F0, F2, F4	✓	✓	
SUB.D F8, F2, F6	✓	✓	✓
DIV.D F10, F0, F6	✓		
ADD.D F6, F8, F2	✓	✓	✓

Name	Reservation stations						
	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		

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

Tomasulo's Algorithm: A Loop-Based Example

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

A Loop-Based Example (Cont'd)

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

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

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

Issue of Load and Store Access

- If a load and a store access the same address, then either:
 - The load is before the store in program order and interchanging them results in a WAR hazard, or
 - The store is before the load in program order and interchanging them results in a RAW hazard
- Interchanging two stores to the same address results in WAW hazard
- A simple, but not necessarily optimal, way to guarantee that the processor has all such addresses is to perform the effective address calculations in program order

Two Techniques in Tomasulo's Algorithm

- The renaming of the architectural registers to a larger set of registers
- The buffering of source operands from the register file

Key Components in Tomasulo's Algorithm

- Dynamic scheduling
- Register renaming
- Dynamic memory disambiguation

Hardware-Based Speculation

- Control dependences becomes an increasing burden in exploiting ILP
- Overcoming control dependence is done by speculating on the outcome of branches and executing the program as if our guesses were correct
- Hardware speculation extends the ideas of dynamic scheduling

Key Ideas of Hardware-Based Speculation

- Dynamic branch prediction chooses which instructions to execute
- Speculation allows the execution of instructions before the control dependences are resolved
- Dynamic scheduling deals with the scheduling of different combinations of basic blocks
- Branch prediction reduces the direct stalls attributable to branches

Key Ideas of Hardware-Based Speculation (Cont'd)

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

Extension of Dynamic Scheduling : Instruction Commit

- The hardware that implements Tomasulo's algorithm can be extended to support speculation
- The bypassing of results among instructions and the actual completion of an instruction must be separated
- Thus, we can allow instruction executions and result bypassings without performing instruction updates, until the instruction is no longer speculative

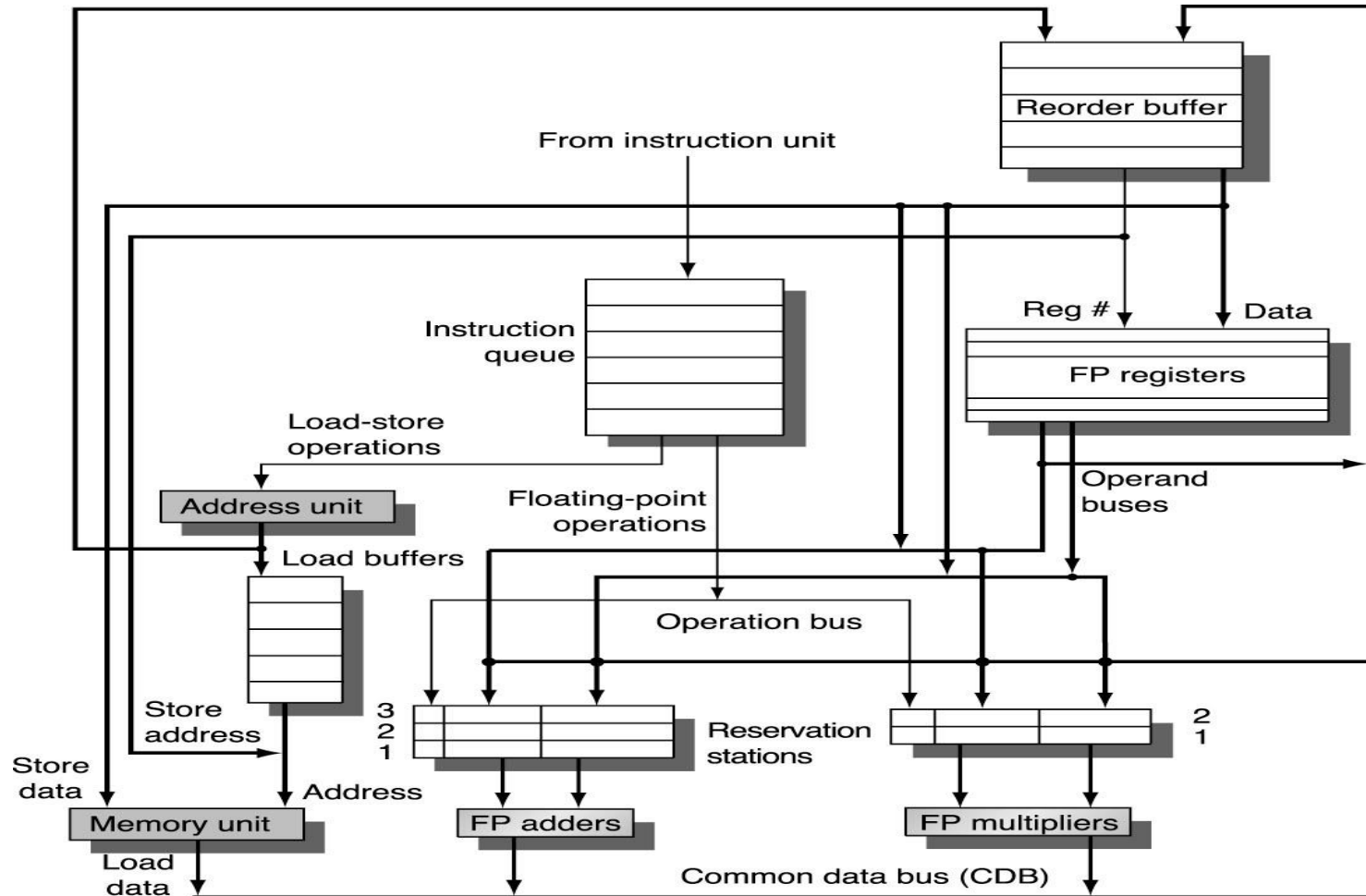
Reorder Buffer (ROB)

- Adding this commit phase to the instruction execution sequence requires
 - some changes to the sequence
 - reorder buffer (ROB)
- ROB passes results among instructions that may be speculated
- ROB holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits
- ROB is a source of operands for instructions, just as the reservation stations provide operands in Tomasulo's algorithm

ROB (Cont'd)

- 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
- Each entry in the ROB contains four fields:
 - Instruction type
 - Destination field
 - Value field
 - Ready field

Hardware Structure Including the ROB



Four Pipeline Stages

- Issue
- Execute
- Write result
- Commit

Issue Stage

- Get an instruction from the instruction queue
- Issue the instruction if there is an empty reservation station and an empty slot in the ROB
- Send the operands to the reservation station if they are available in either the registers or the ROB
- Update the control entries to indicate the buffers are in use
- The number of the ROB allocated for the result is also sent to the reservation station, so that the number can be used to tag the result when it is placed on the CDB

Execute Stage

- This step checks for RAW hazards
- If one or more of the operands is not yet available, monitor the CDB while waiting for the register to be computed
- When both operands are available at a reservation station, execute the operation
- Instructions may take multiple clock cycles in this stage
- Loads still require two steps in this stage
- Stores need only have the base register available at this step, since execution for a store at this point is only effective address calculation¹⁰⁴

Execute Stage (Cont'd)

- If more than one instruction is ready for a single functional unit, the unit will have to choose among them
- Loads and stores require a two-step execution process
- The first step computes the effective address and then places it in the load or store buffer. Loads in the load buffer execute as soon as the memory unit is available

Execute Stage (Cont'd)

- Stores wait from the value to be stored before being sent to the memory unit
- Loads and stores are maintained in program order through the effective address calculation, which will help to prevent hazards through memory

Write Result

- When the result is available
 - write it on the CDB (with the ROB tag sent when the instruction issued)
 - from the CDB into the ROB, as well as
 - to any reservation stations waiting for this result
- Mark the reservation station as available

Write Result (Cont'd)

- Special actions are required for store instructions
 - If the value to be stored is available, it is written into the Value field of the ROB entry for the store
 - If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated

Commit

- There are three different sequences of actions at commit depending on whether the committing instruction is
 - The normal commit case occurs when an instruction reaches the head of the ROB and its result is sent in the buffer
 - Committing a store is similar except that memory is updated rather than a result register
 - When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong

Example

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

L.D F6,34(R2)

L.D F2,45(R3)

MUL.D F0,F2,F4

SUB.D F8,F6,F2

DIV.D F10,F0,F6

ADD.D F6,F8,F2

Answer

Name	Reservation stations							
	Bu sy	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	

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

Field	FP register status									
	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

Example

- Consider the code example for Tomasulo's algorithm

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

Assume that we have issued all the instructions in the loop twice. Let's also assume that **the L.D and MUL.D from the first iteration have committed and all other instructions have completed execution**. Normally, the store would wait in the ROB for both the effective address operand (R1 in this example) and the value (F4 in this example). Since we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

Answer

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

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

Difference in Store Handling

- In Tomasulo's algorithm
 - A store can update memory when it reaches **Write Results**
 - It ensures that the effective address has been calculated and the data value to store is available
- In a speculative processor
 - A store updates memory only when it reaches the head of the ROB
 - This difference ensures that memory is not updated until an instruction is no longer speculative

Misprediction Handling

- Using ROB, the processor can easily undo its speculation action when a misprediction occurred
 - This recovery can be done by clearing the ROB for all entries that appear after the mispredicted branch
 - Allow those that are before the branch in the ROB to continue
 - Restart the fetch at the correct branch successor

Exception Handling

- Recognize exception until it is ready to commit
- If a speculated instruction raises an exception, the exception is recorded in ROB
- If a branch misprediction arises and the instruction should not have been executed, the exception is flushed along with the instruction when the ROB is cleared
- If the instruction reaches the head of the ROB, the exception will be taken

Eliminating Hazards Through Memory

- WAW and WAR hazards through memory are eliminated with speculation, because
 - The actual updating of memory occurs in order, when a store is at the head of the ROB
 - Hence, no earlier loads or stores can still be pending

Eliminating Hazards Through Memory (Cont'd)

- RAW hazards through memory are maintained by two restrictions
 - Not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has an Destination field that matches the value of the A field of the load
 - Maintaining the program order for the computation of an effective address of a load with respect to all earlier stores

Five Primary Approaches in Use for Multiple-Issue Processors

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	dynamic	hardware	static	in-order execution	Sun UltraSPARC II/III
Superscalar (dynamic)	dynamic	hardware	dynamic	some out-of-order execution	HP PA 8500, IBM RS64 III
Superscalar (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Pentium III/4, MIPS R10K, Alpha 21264
VLIW/LIW	static	software	static	no hazards between issue packets	Trimedia, i860
EPIC	mostly static	mostly software	mostly static	explicit dependences marked by compiler	Itanium

Statically-Scheduled Superscalar Processors

- In a typical superscalar processor, the hardware might issue from zero to eight instructions in a clock cycle
- In a statically-scheduled superscalar, instructions issue in order and all pipeline hazards are checked for at issue time
- A superscalar processor has dynamic issue capability
- A VLIW processor has static issue capability

Static Multiple Issue: the VLIW Approach

- The compiler may be required to ensure
 - Dependences within the issue packet cannot be present or
 - When a dependence may be present
- Such an approach offers the potential advantage of simpler hardware

The Basic VLIW Approach

- VLIWs use multiple, independent functional units
- The burden for choosing the instructions to be issued simultaneously falls on the compiler
- A VLIW processor might have instructions that contain five operations, including: one integer operation (which could also be a branch), two floating-point operations, and two memory references
- The instruction would have a set of fields for each functional unit—perhaps 16 to 24 bits per unit, yielding an instruction length of between 112 and 168 bits

The Basic VLIW Approach (Cont'd)

- To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots
- Local scheduling techniques operate on a single basic block
- Global scheduling can find and exploit the parallelism requires scheduling code across branches

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 the branch-delay slot.

Answer

- The loop has been unrolled to make seven copies of the body, which eliminates all stalls, and runs in 9 cycles. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result, nearly twice as fast as the two-issue superscalar

Answer (Cont'd)

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 -8(R1),F8	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D -24(R1),F16			
S.D F20,-32(R1)	S.D -40(R1),F24			DADDUI R1,R1,#-56
S.D F28,8(R1)				BNE R1,R2,Loop

Technical Problems

- The technical problems are the increase in code size and the limitations of lock-step operation
- Two different elements combine to increase code size substantially for a VLIW
 - First, generating enough operations in a BB requires ambitiously unrolling loops thereby increasing code size
 - Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding
- In most VLIWs, an instruction may need to be left completely empty if no operations can be scheduled

Technical Problems (Cont'd)

- A stall in any functional unit pipeline must cause the entire processor to stall, since all the functional units must be kept synchronized
- Caches needed to be blocking and to cause all the functional units to stall
- As the issue rate and number of memory references becomes large, this synchronization restriction becomes unacceptable
- The compiler is used to avoid hazards at issue time
- Hardware checks allow for unsynchronized execution once instructions are issued

Logistical Problems

- Binary code compatibility has also been a major logistical problem for VLIWs
- In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies
- Thus, different numbers of functional units and unit latencies require different versions of the code

Two Solutions of Logistical Problems

- One possible solution to this migration problem, and the problem of binary code compatibility in general, is object-code translation or emulation
- Another approach is to temper the strictness of the approach so that binary compatibility is still feasible

The Major Challenge for All Multiple-Issue Processors

- The major challenge for all multiple-issue processors is to try to exploit large amounts of ILP
- It is not clear that a multiple-issue processor is preferred over a vector processor for FP programs
- The potential advantages of a multiple-issue processor versus a vector processor are twofold
 - First, a multiple-issue processor has the potential to extract some amount of parallelism from less regularly structured code
 - Second, it has the ability to use a more conventional, and typically less expensive, cache-based memory system

Multiple Issue with Speculation

- A speculative processor can be extended to multiple issue
- We process multiple instructions per clock assigning reservation stations and reorder buffers to the instructions
- Two challenges of multiple issue with Tomasulo's algorithm
 - Instruction issue
 - Monitoring the CDBs for instruction completion
- To maintain throughput of greater than one instruction per cycle, a speculative processor must be able to handle multiple instruction commits per clock cycle

Example

- Consider the execution of the following loop, which searches an array, on a two issue processor, once without speculation and once with speculation. Assume that there are separate integer functional units for effective address, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations for both processors. Assume that up to two instructions of any type can commit per clock.

Loop:	LW	R2,0(R1);	R2=array element
	DADDIU	R2,R2,#1;	increment R2
	SW	0(R1),R2;	store result
	DADDIU	R1,R1,#4;	increment pointer
	BNE	R2,R3,LOOP;	branch if last element!=0

Answer

Iter. #	Instructions	Issues at clock cycle #	Executes at clock cycle #	Memory access at clock cycle #	Write CDB at clock cycle #	Comment
1	LW R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SW 0(R1),R2	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LW R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SW 0(R1),R2	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LW R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SW 0(R1),R2	8	19	20		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

Answer (Cont'd)

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

Advanced Techniques for Instruction Delivery and Speculation

- In a high performance pipeline, especially one with multiple issue, predicting branches well is not enough
- We actually have to be able to deliver a high bandwidth instruction stream
- In recent multiple issue processors, this has meant delivering 4-8 instructions every clock cycle

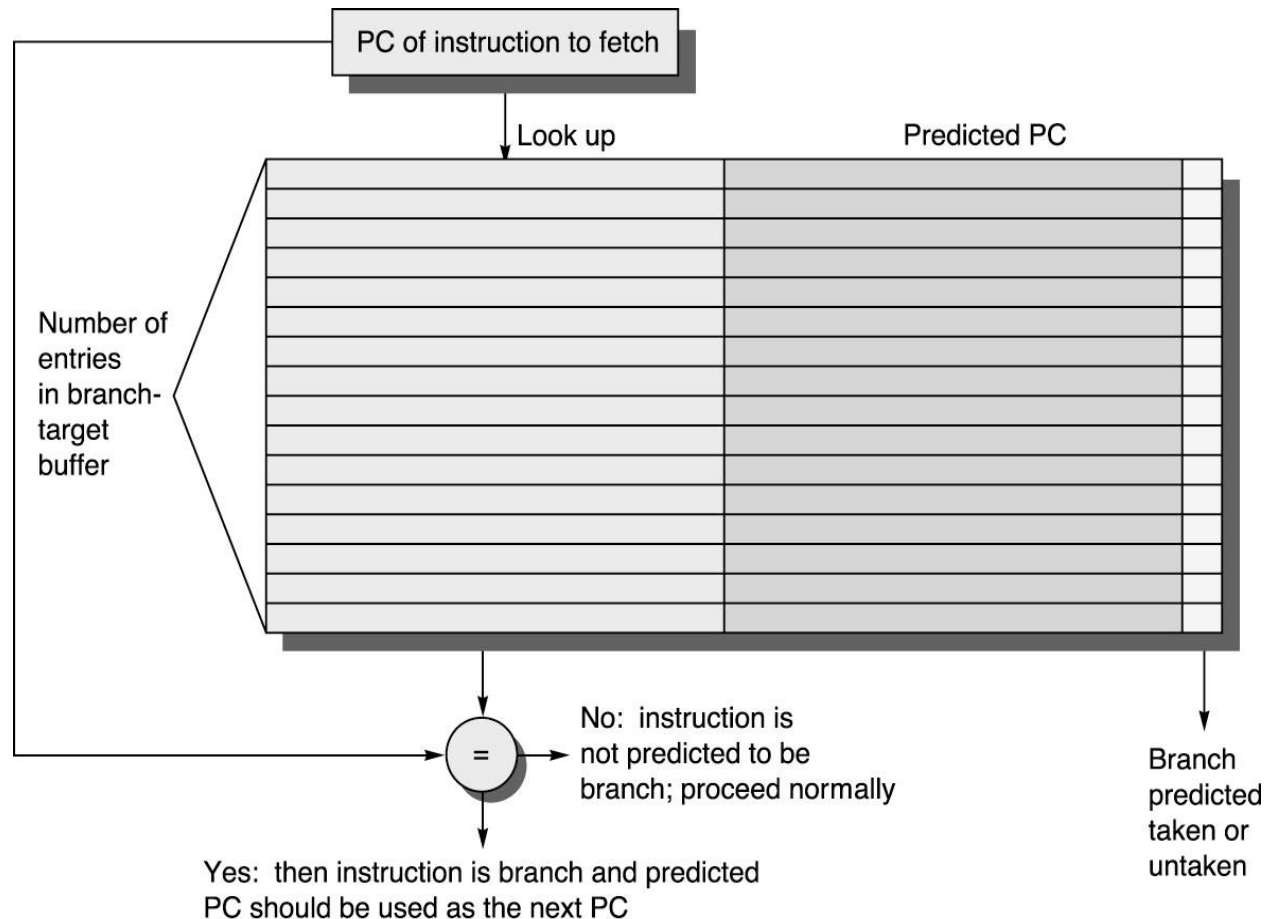
Increasing Instruction Fetch Bandwidth

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

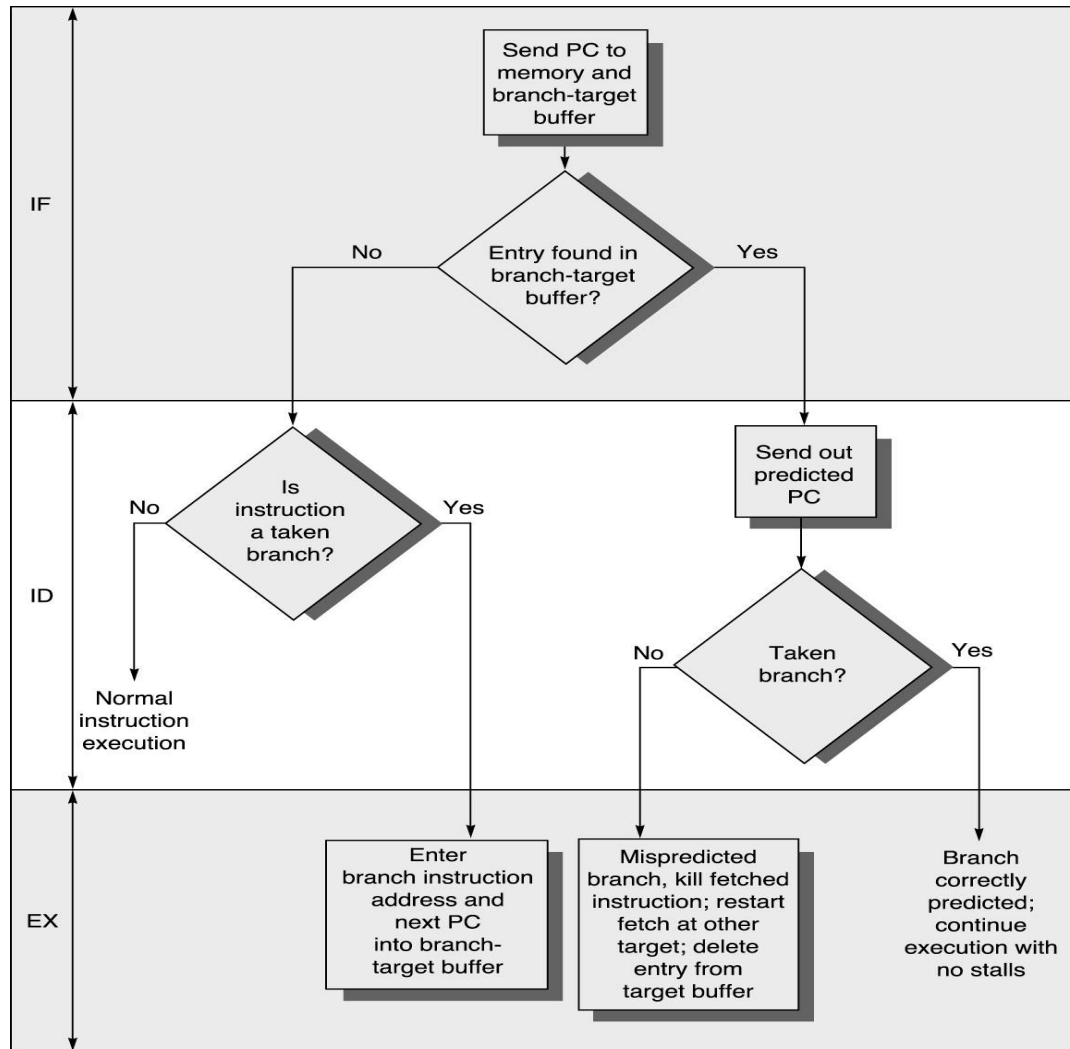
Branch Target Buffers

- 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*
- Branch-prediction buffer is accessed during the ID cycle
- Branch-target buffer may know the predicted instruction address at the end of the IF cycle, which is one cycle earlier than for a branch-prediction buffer
- If the PC of the fetched instruction matches a PC in the buffer, then the corresponding predicted PC is used as the next PC

A Branch-Target Buffer



Steps in Branch-Target Buffer



Penalties in 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

Example

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

Assume that 60% of the branches are taken.

Answer

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

Probability (branch not in buffer, but actually taken)
= $10\% \times 60\% = 0.06$

Branch penalty = $(0.09 + 0.06) \times 2 = 0.3$

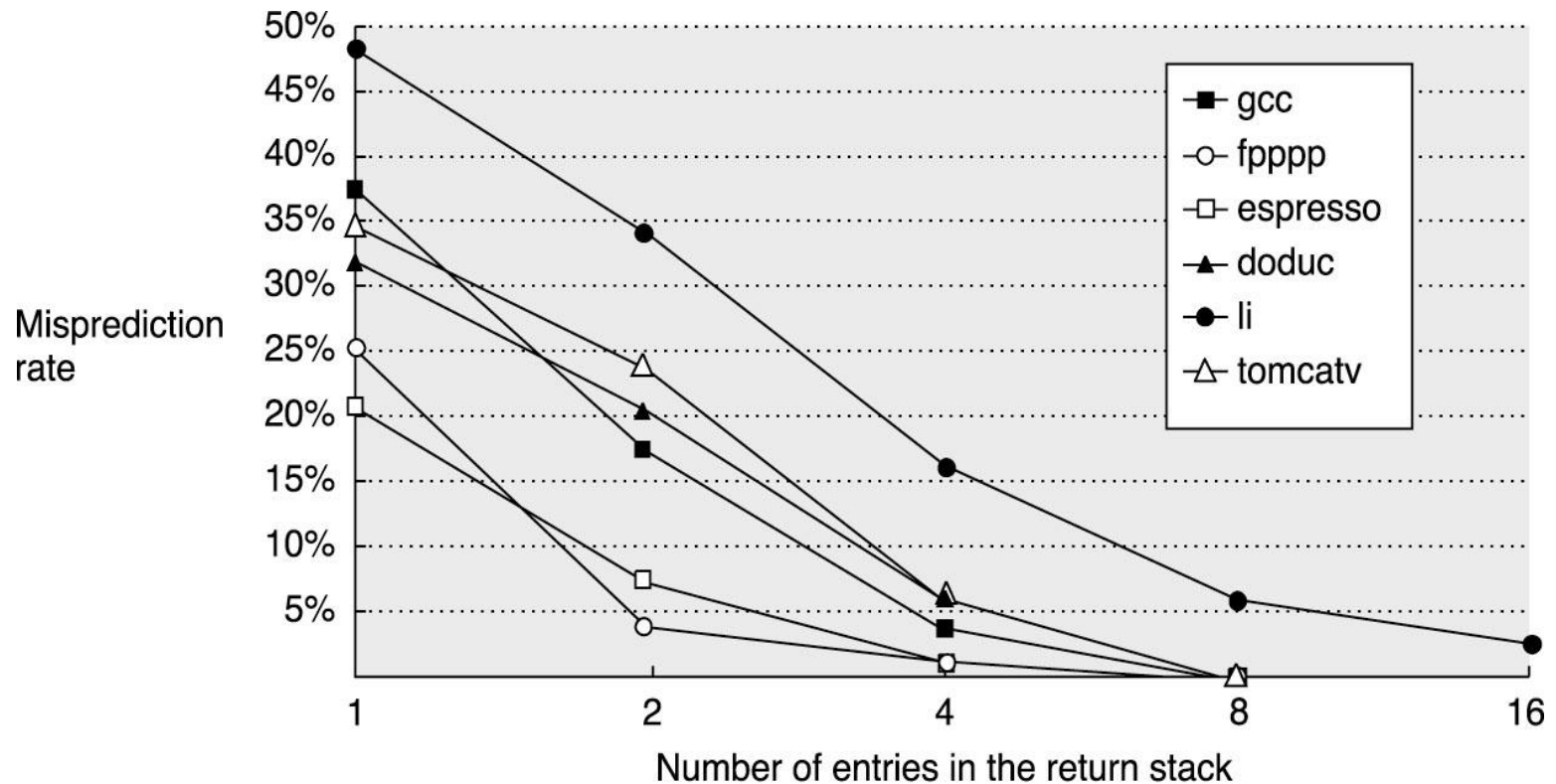
Variation on Branch-Target Buffer

- One variation on the branch-target buffer is to store one or more target instructions. This variation has two potential advantages
 - First, it allows the branch-target buffer access to take longer than the time between successive instruction fetches, possibly allowing a larger branch-target buffer
 - Second, buffering the actual target instructions allows us to perform an optimization called *branch folding*
- Branch folding can be used to obtain zero-cycle unconditional branches, and sometimes zero-cycle conditional branches

Return Address Predictors

- It is a technique for predicting indirect jumps, that is, jumps whose destination address varies at runtime
- For example, select or case statements, and FORTRAN-computed gotos, the vast majority of the indirect jumps come from procedure returns
- Prediction accuracy can be low if the procedure is called from multiple sites
- A small buffer of return addresses operating as a stack has been proposed to remedy this
- This structure caches the most recent return addresses

Prediction Accuracy for A Return Address Buffer



Integrated Instruction Fetch Units

- It's a separate autonomous unit that feeds instructions to the rest of the pipeline
- It characterizes instruction fetch as a simple single pipe stage given the complexities of multiple issue is no longer valid

Functions in Recent IIFUs

- Integrated branch prediction
 - The branch predictors become part of the IIFUs
- Instruction prefetch
 - Multiple issue
 - Prefetching management
 - Integration with branch prediction
- Instruction memory access and buffering
 - Fetches of multiple instruction lead to access multiple cache lines
 - Prefetching hides the cost of crossing cache blocks
 - Buffering provides on-demand instructions to the issue stage

Register Issues

- How do we ever know which registers are the architectural registers if they are constantly changing?
- There are clearly cases, where another process, such as the operating system, must be able to know exactly where the contents of a certain architectural register resides
- The mapping between the architecturally visible registers and physical registers will become stable
- Architectural registers can be removed when the value of any physical register not associated with an architectural register is unneeded

Renaming VS ROB

- One alternative to the use of a ROB is the explicit use of a larger physical set of registers combined with register renaming
- With the addition of speculation, register values may also temporarily reside in the ROB
- An advantage of the renaming approach is that instruction commit is simplified
 - Record the mapping between an architectural register number and physical register number is no longer speculative
 - Free up any physical registers being used to hold the “older” value of the architectural register
- With register renaming, deallocating registers is more complex

How Much to Speculate

- Advantages of speculation:
 - The ability to uncover events that would otherwise stall the pipeline early, such as cache misses
- Disadvantage:
 - The processor may speculate that some costly exceptional event occurs and begin processing the event, when in fact, the speculation was incorrect
- To overcome the disadvantage, we must use a low-cost exceptional event to be handled in speculative mode

Speculating through Multiple Branches

- Three different situations can benefit from speculating on multiple branches simultaneously:
 - a very high branch frequency
 - significant clustering of branches
 - long delays in functional units
- In the first two cases, achieving high performance may mean that multiple branches are speculated
- Save stalls to avoid long delays in function units

Value Prediction

- Value prediction predicts the values produced by instructions
- It works better for values changed infrequently
 - Loads and stores from a constant pool
 - A value is used for the source of a chain of dependent computations
- Much research focused on loads

Address Aliasing Prediction

- It predicts two stores or a load and a store refer to the same memory address

Introduction

- ILP is the primary focus of processor designs
- We examine the limitations of ILP
- We study the use of thread-level parallelism
- We compare the recent processors both in performance and in efficiency measure per transistor and per watt

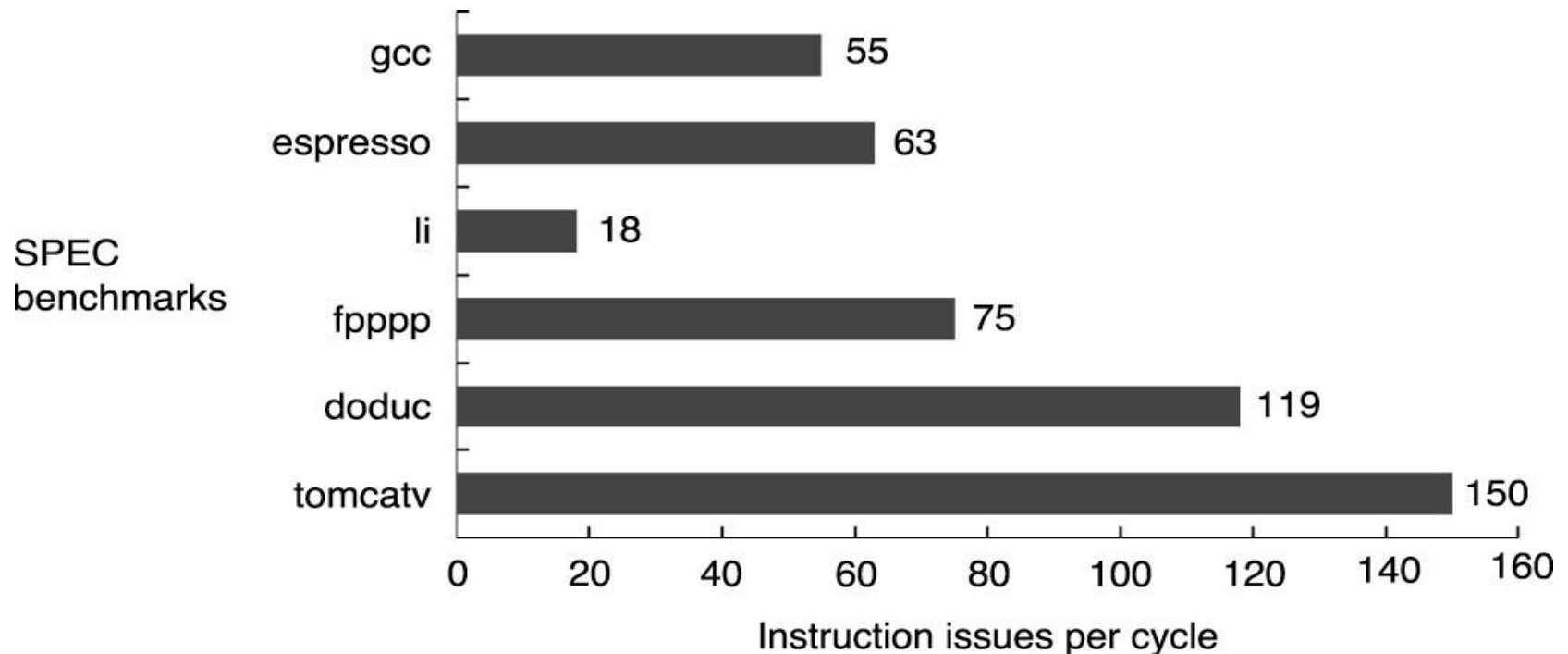
The Hardware Model for Limitations of ILP

- An ideal processor is one where all constraints on ILP are removed
- The only limits on ILP are those imposed by the actual data flows through registers and memory
- The assumptions made for an ideal or perfect processor are as follows:
 - Register renaming
 - Branch prediction
 - Jump prediction
 - Memory-address alias analysis
 - Perfect caches

The Hardware Model for Limitations of ILP (Cont'd)

- Assumptions 1 and 4 eliminate all but the true dependences
- Assumptions 2 and 3 eliminate control dependences
- For the unlimited case, there are infinite loads and stores issuing in one clock cycle
- All functional unit latencies are assumed to be one cycle, so that any sequence of dependent instructions can issue on successive cycles
- Perfect caches are assumed to let loads and stores can be completed in one cycle

ILP Available in a Perfect Processor for Six of the SPEC92 Benchmarks



Limitations on the Window Size and Maximum Issue Count

- How close could a real dynamically scheduled, speculative processor come to the ideal processor?
- The perfect processor must do:
 - Look arbitrarily far ahead to find a set of instructions to issue, predicting all branches perfectly
 - Rename all register uses to avoid WAR and WAW hazards
 - Determine whether there are any data dependencies among the instructions in the issue packet; if so, rename accordingly
 - Determine if any memory dependences exist among the issuing instructions and handle them appropriately
 - Provide enough replicated functional units to allow all the ready instructions to issue

Limitations on the Window Size and Maximum Issue Count (Cont'd)

- Obviously, this analysis is quite complicated
- For example, to determine whether n issuing instructions have any register dependences among them, assuming all instructions are register-register and the total number of registers is unbounded, requires

$$2n - 2 + 2n - 4 + \dots + 2 = 2 \sum_{i=1}^{n-1} i = 2 \frac{(n-1)n}{2} = n^2 - n$$

comparisons

Window

- In existing and near-term processors, the costs are not quite so high, since we need only detect dependence pairs and the limited number of registers allows different solutions
- The set of instructions that are examined for simultaneous execution is called the **window**
- Each instruction in the window must be kept in the processor
- The number of comparisons required every clock is equal to the maximum completion rate times the window size times the number of operands per instruction
- Today typically $6 \times 80 \times 2 = 960$

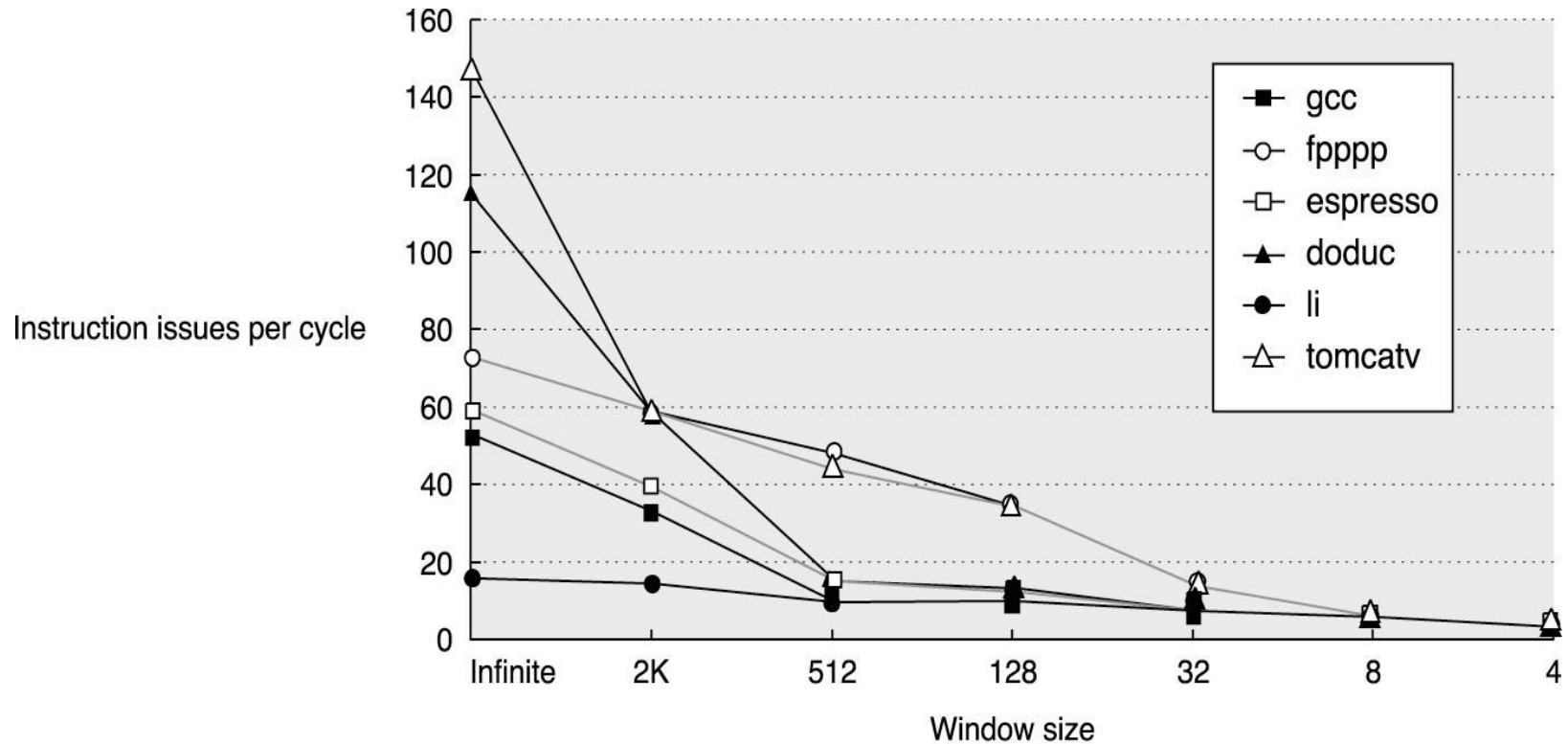
Window (Cont'd)

- To date, the window size has been in the range of 32 to 126, which can require over 2,000 comparisons
- The HP PA 8600 reportedly has over 7,000 comparators!
- The window size directly limits the number of instructions that begin execution in a given cycle
- In practice, real processors will have a more limited number of functional units as well as limited numbers of buses and register access ports
 - e.g., no processor has handled more than two memory references per clock or more than two FP operations

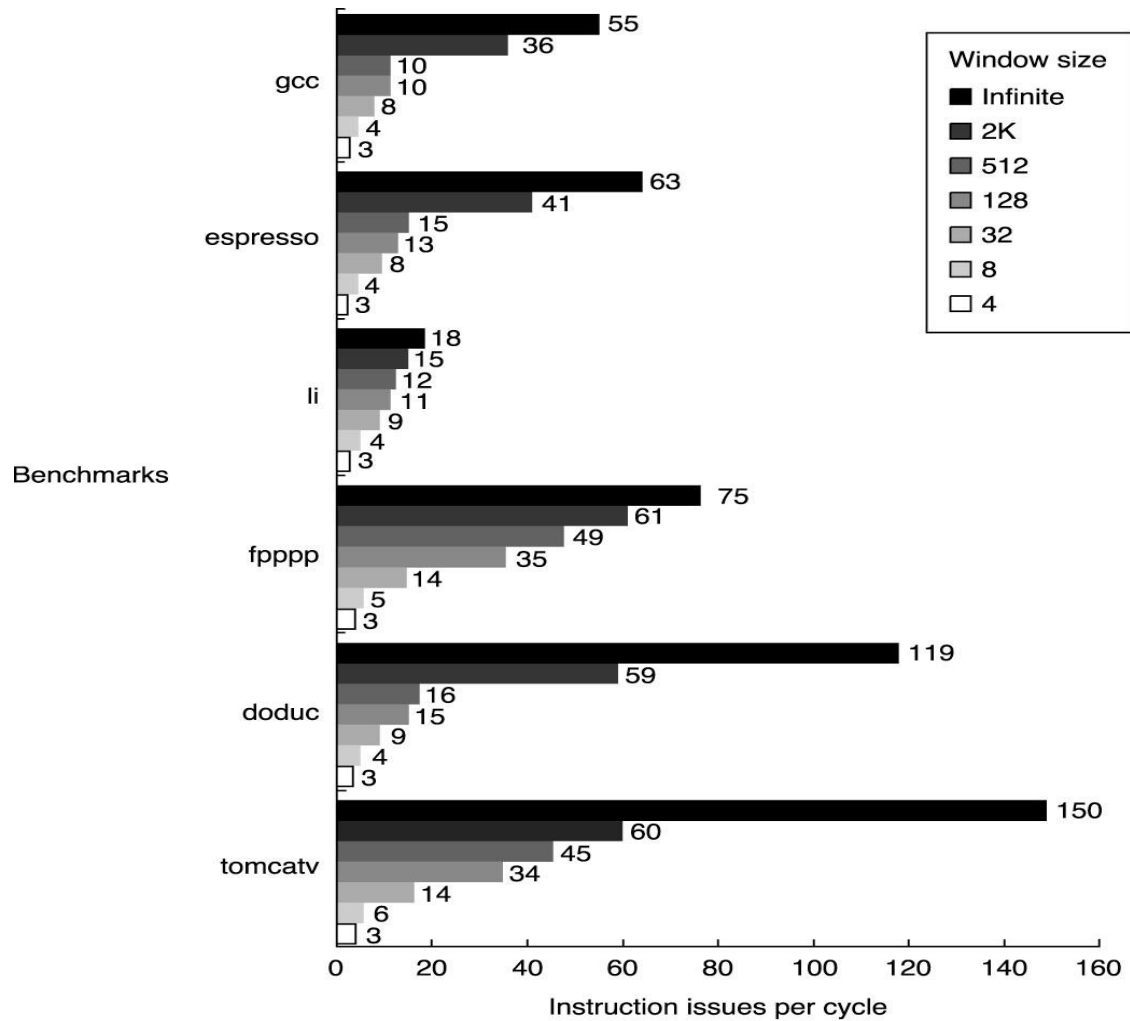
Window (Cont'd)

- The number of possible implementation constraints in a multiple issue processor is large, including:
 - issues per clock
 - functional units
 - unit latency,
 - register file ports
 - functional unit queues
 - issue limits for branches
 - limitations on instruction commit

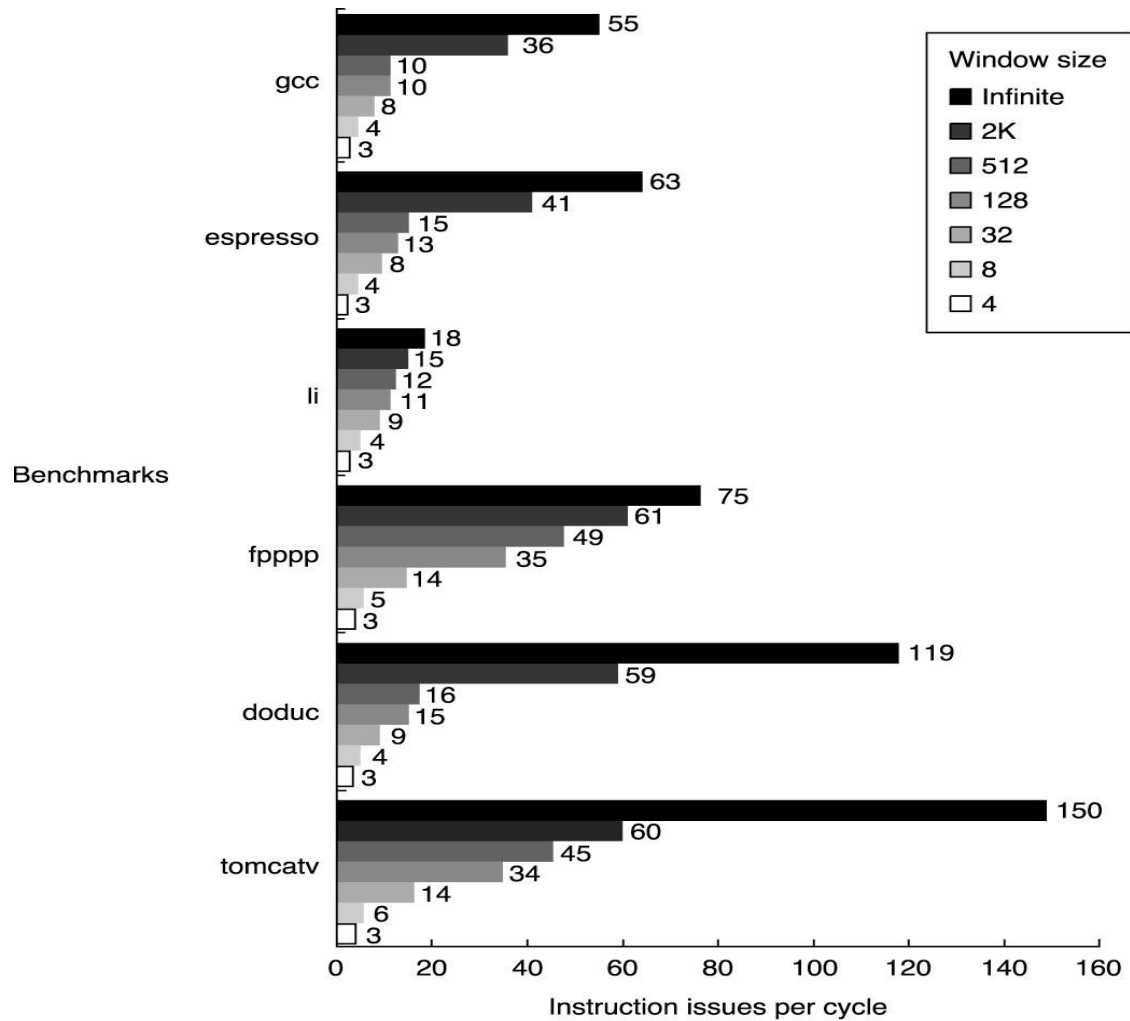
Effects of Reducing the Size of the Window



Effect of Window Size



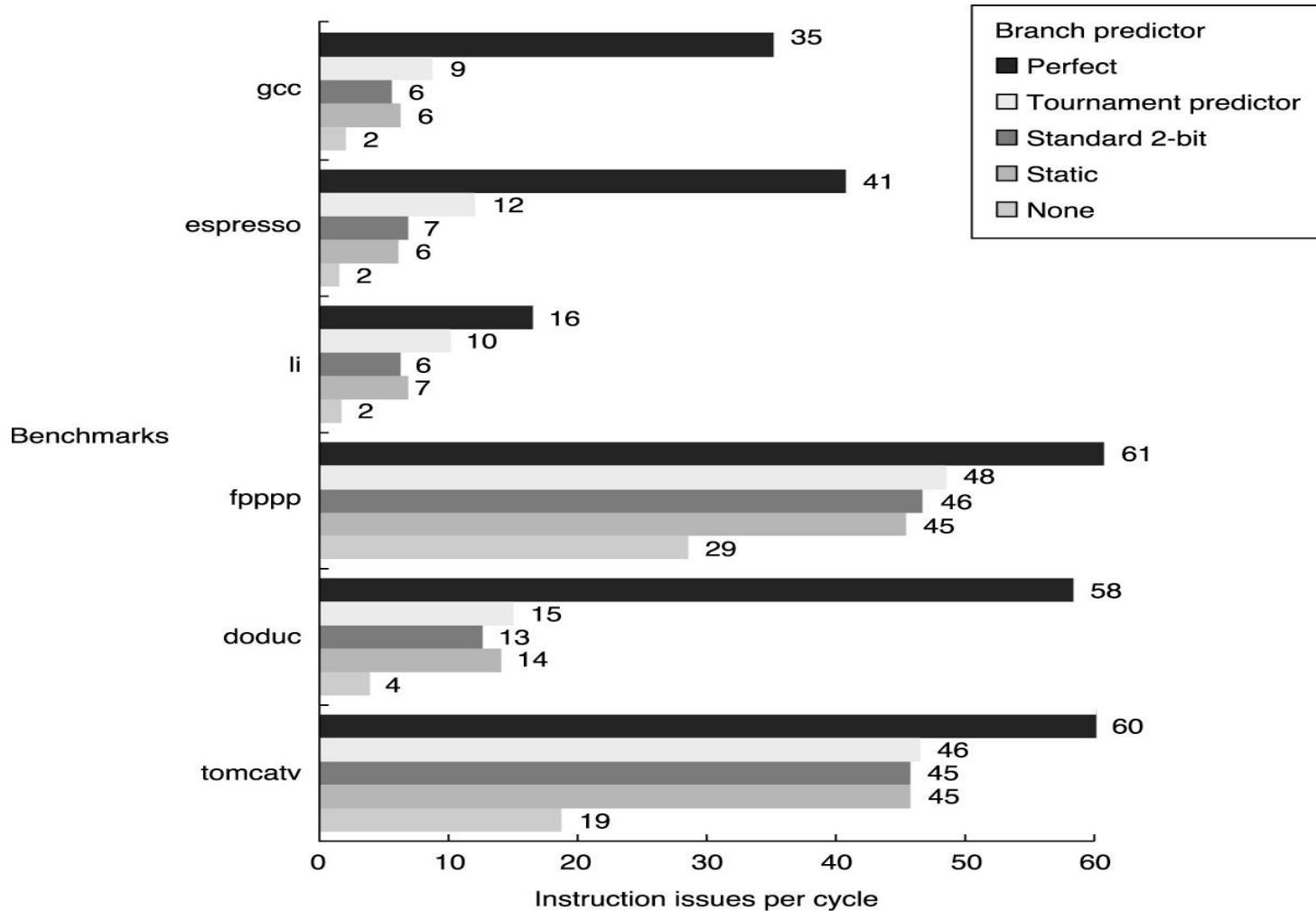
Effect of Window Size



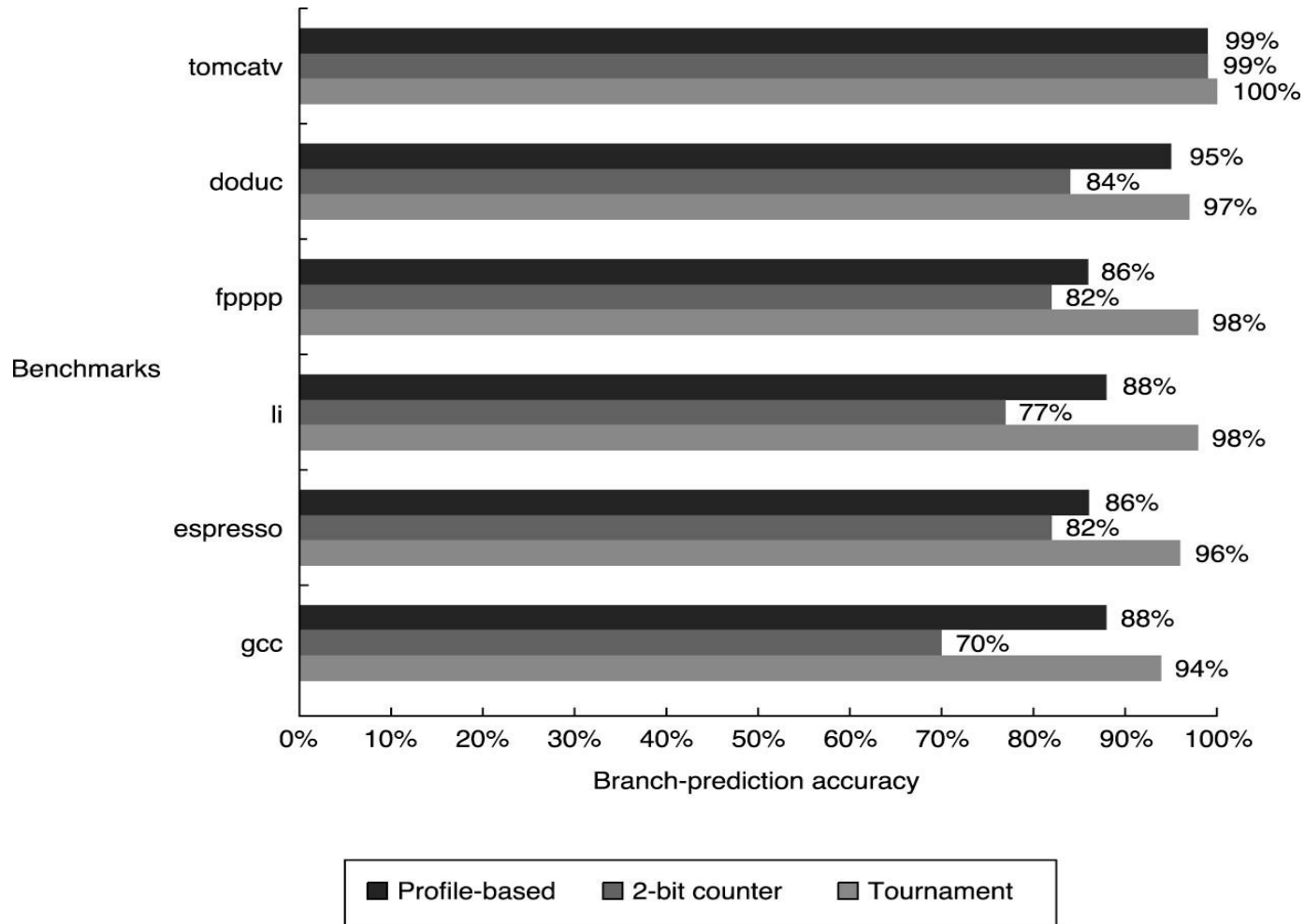
The Effects of Realistic Branch and Jump Prediction

- The five levels of branch prediction shown in these figures are
 - Perfect
 - Tournament-based branch predictor
 - Standard two-bit predictor with 512 two-bit entries
 - Static
 - None

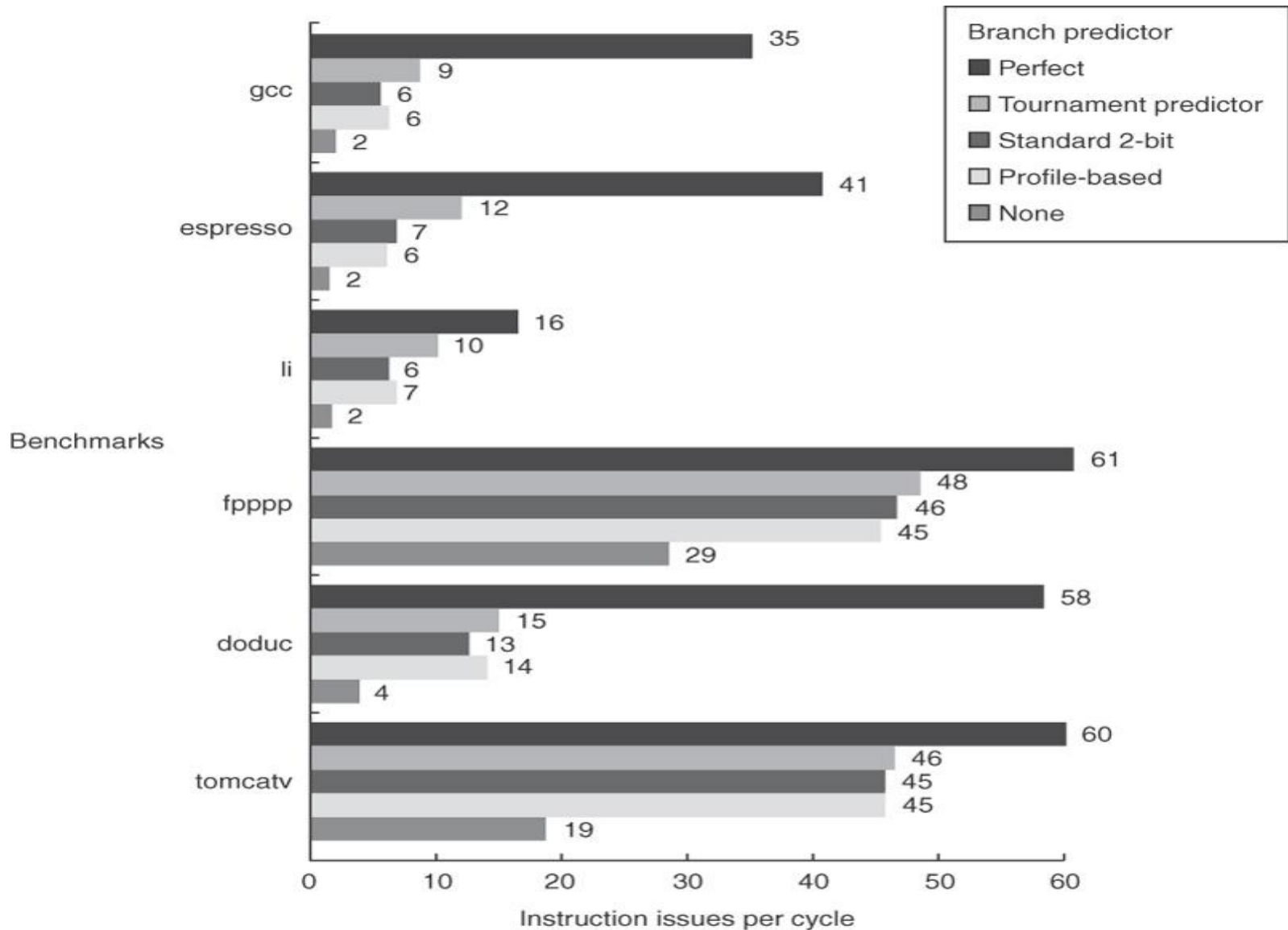
Effect of Branch-Prediction Schemes Sorted by Application



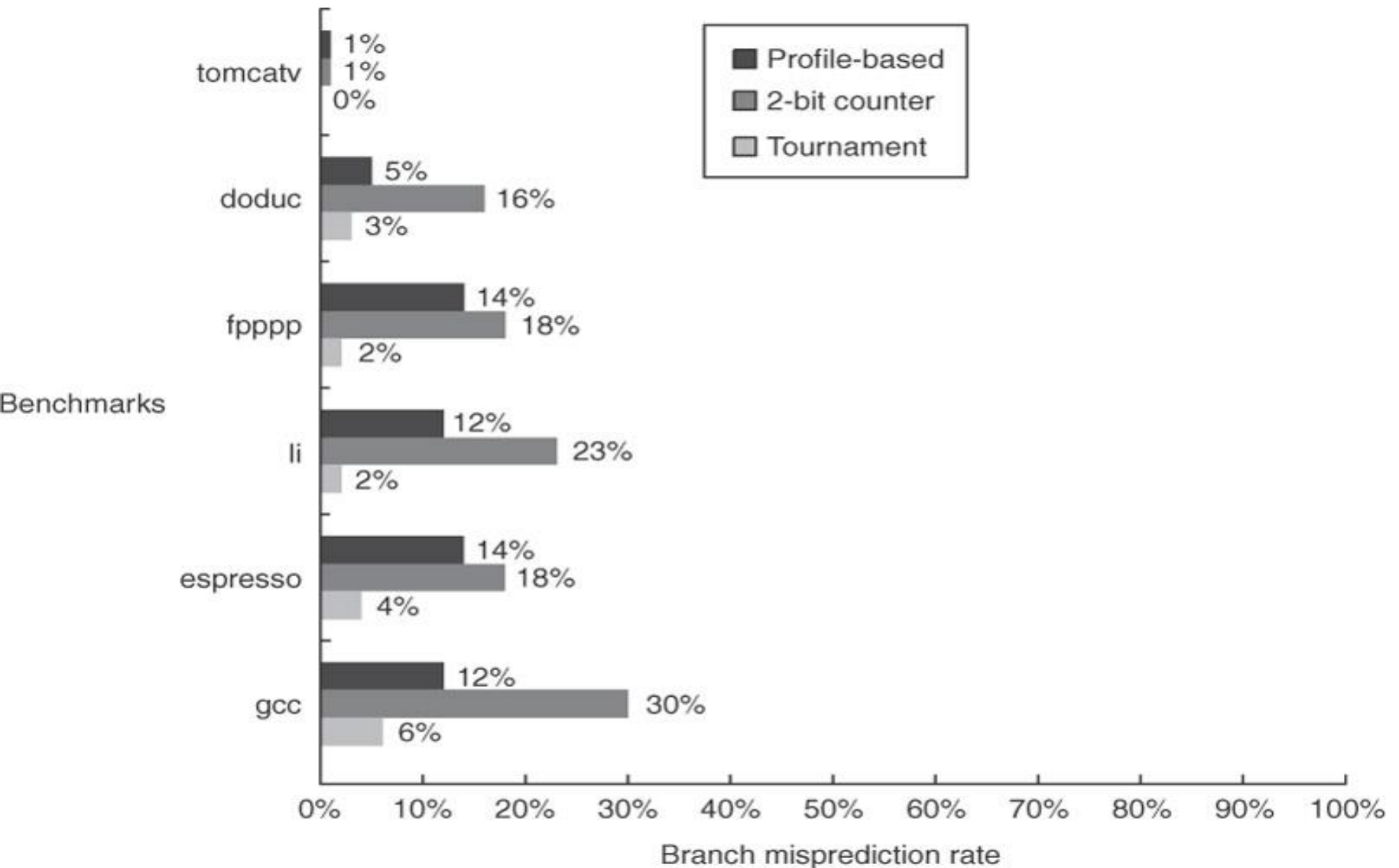
Branch Prediction Accuracy for the Conditional Branches in SPEC92



Effect of Branch-Prediction Schemes



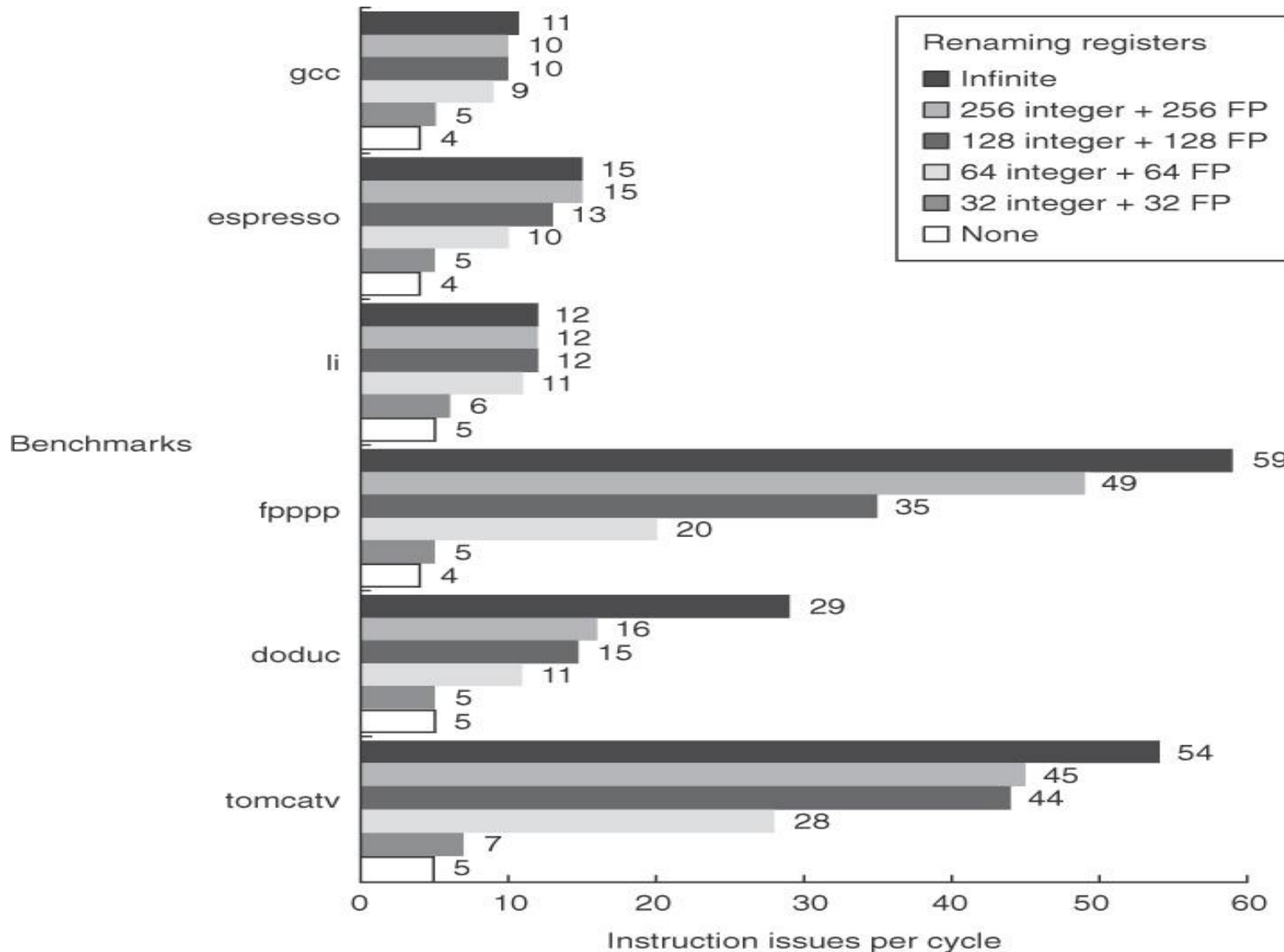
Branch Misprediction rate



The Effects of Finite Registers

- To date, IBM power5 has provided the largest number of virtual registers: 88 FP and 88 integer registers
- Exploiting large amounts of parallelism requires evaluating many possible execution paths, speculatively
- Many registers are used to hold live variables

Reduction in Available parallelism



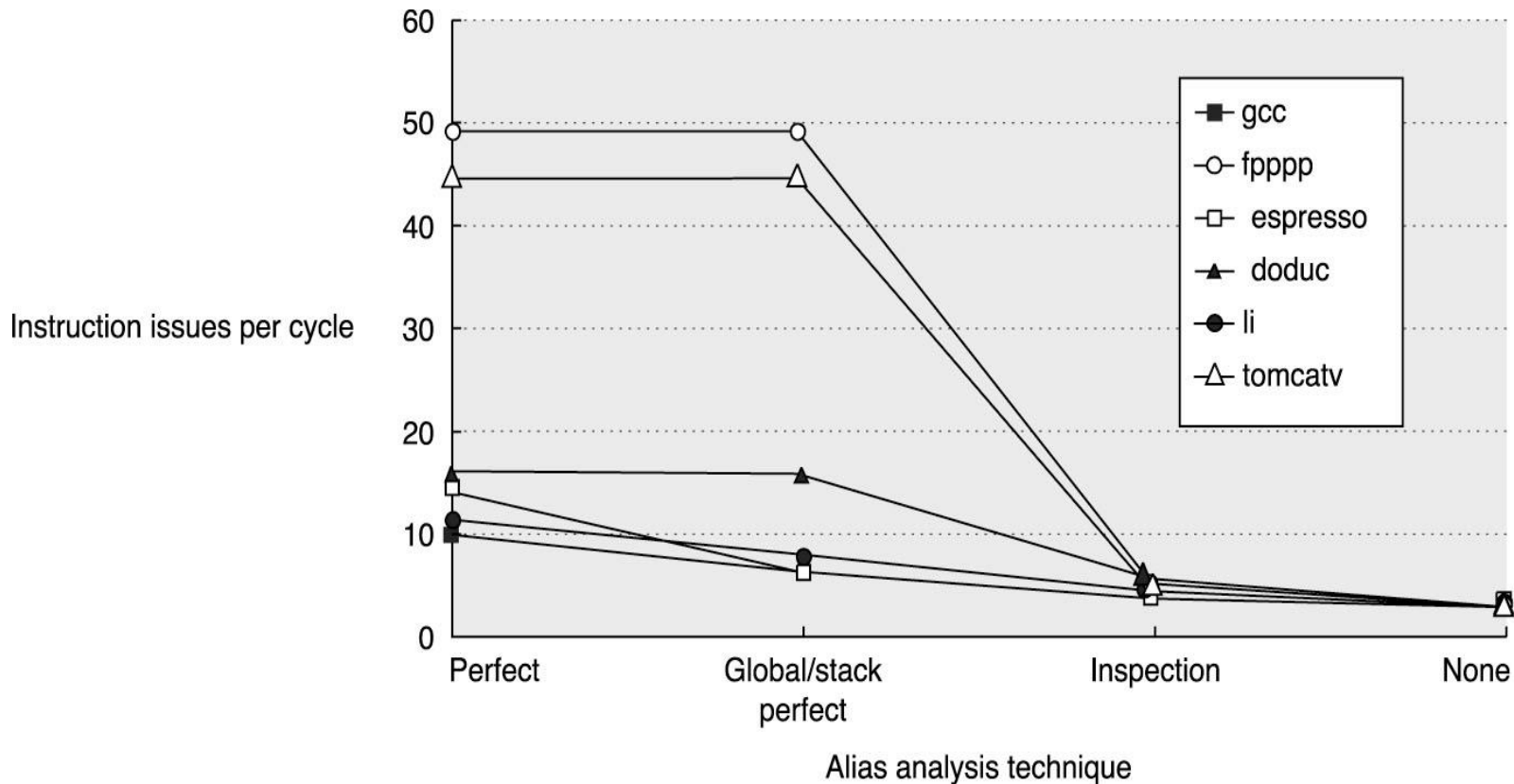
The Effects of Imperfect Alias Analysis

- The analysis cannot be perfect at compile time, and it requires a potentially unbounded number of comparisons at runtime

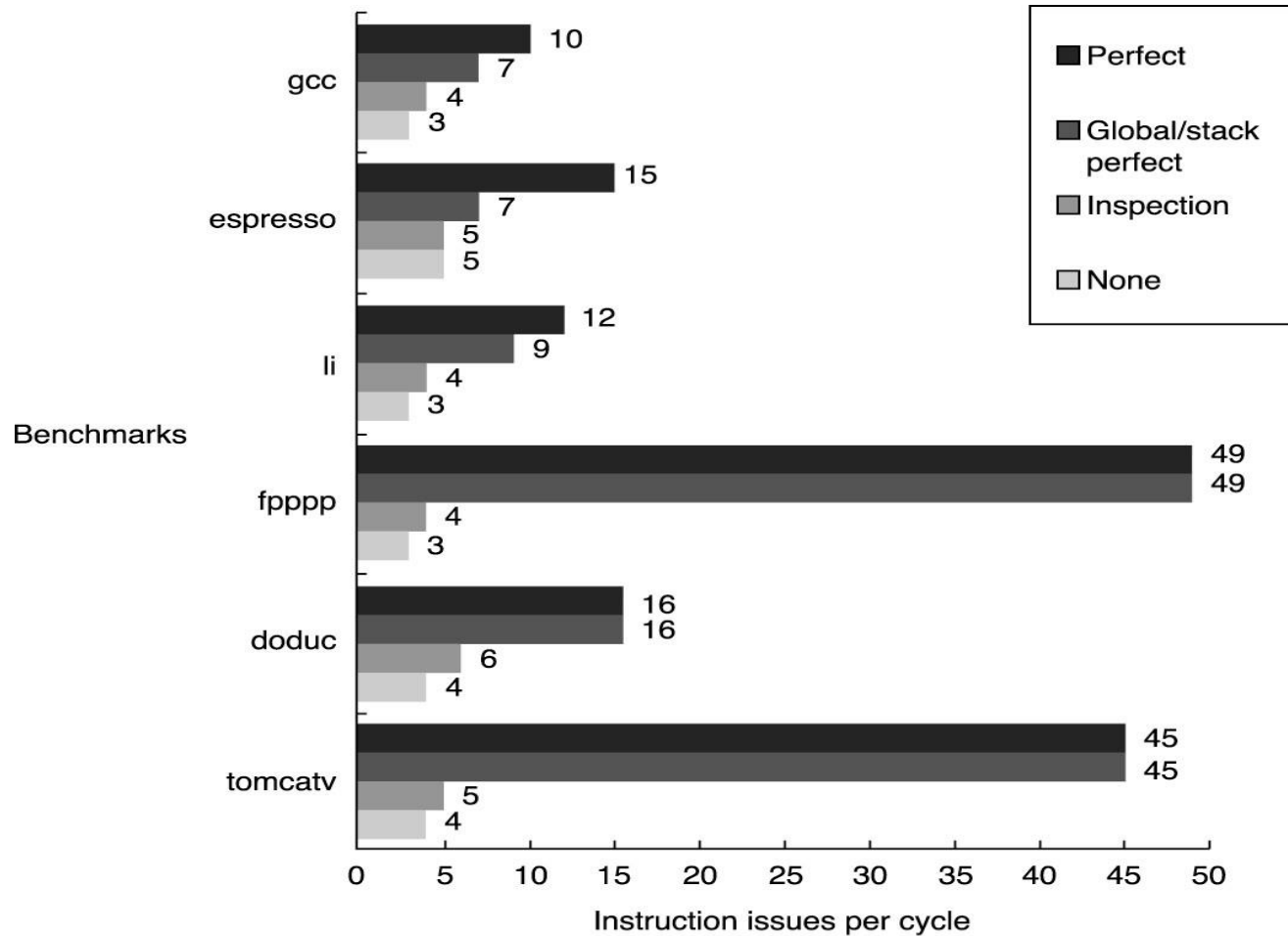
The Effects of Imperfect Alias Analysis (Cont'd)

- Three models of memory alias analysis
 - Global/stack perfect
 - Inspection
 - None

The Effect of Various Alias Analysis Techniques on the Amount of ILP



The Effect of Varying Levels of Alias Analysis



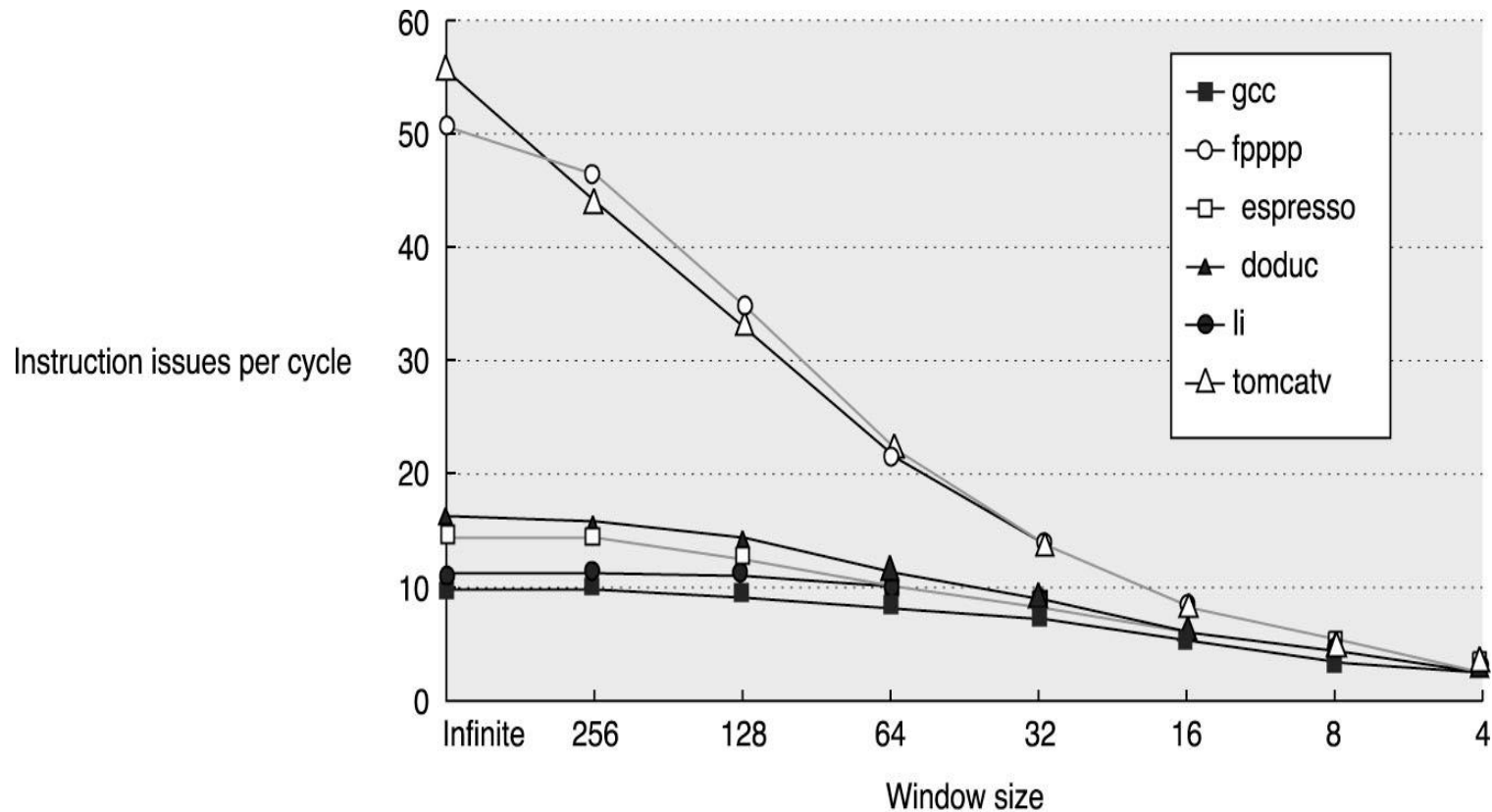
Three Factors on Effects of Imperfect Alias Analysis

- Dynamically scheduled processors rely on dynamic memory disambiguation and are limited by three factors:
 - To implement perfect dynamic disambiguation for a given load, we must know the memory addresses of all earlier stores that not yet committed
 - Only a small number of memory references can be disambiguated per clock cycle
 - The number of the load/store buffers determines how much earlier or later in the instruction stream a load or store may be moved

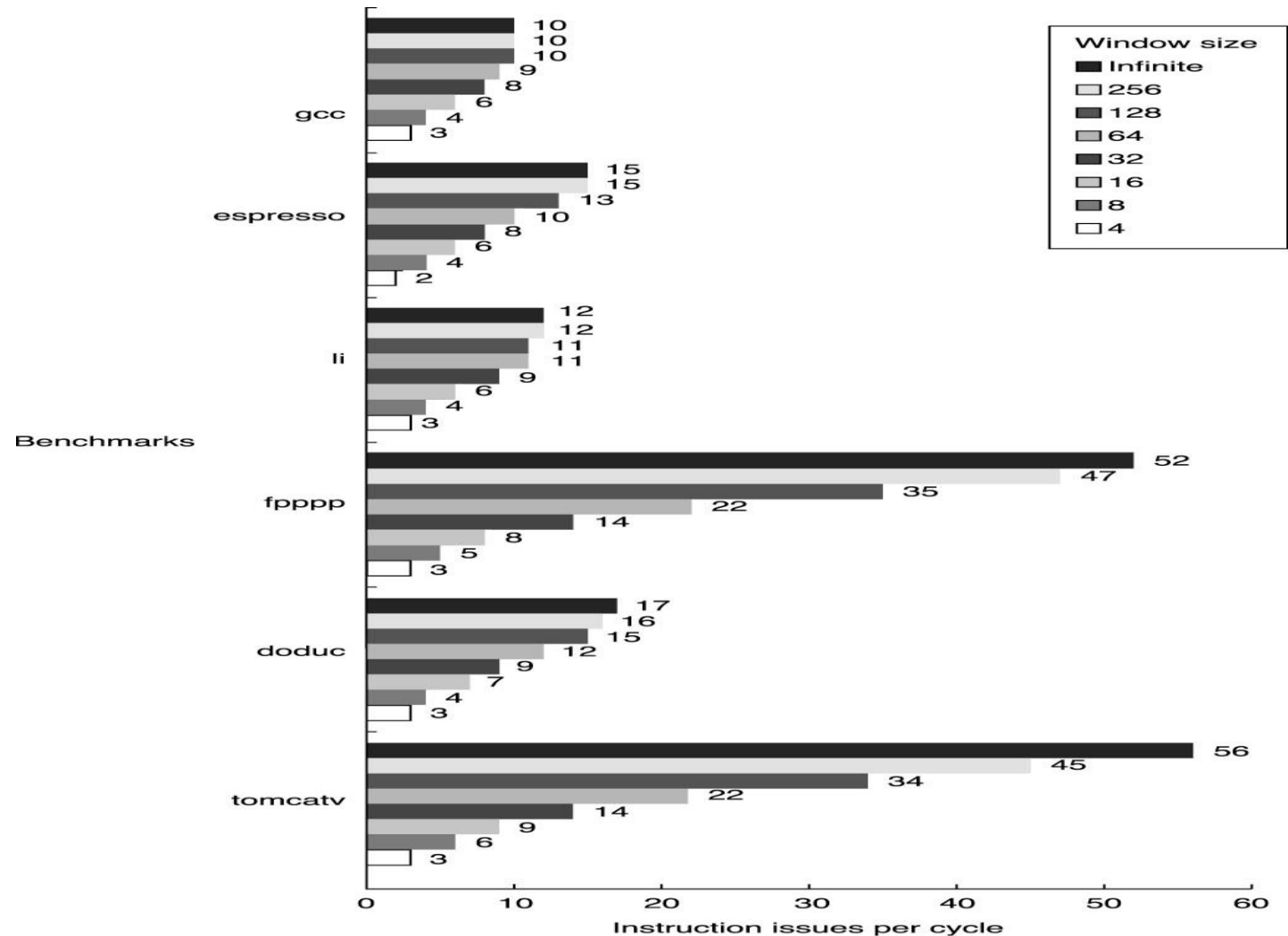
Limitations on ILP for Realizable Processors

- Assume the following fixed attributes
 - Up to 64 instruction issues per clock with no issue restrictions
 - A tournament predictor with 1K entries and a 16-entry return predictor
 - Perfect disambiguation of memory references done dynamically
 - Register renaming with 64 additional integer and 64 additional FP registers, which is roughly comparable to the IBM Power5

The Parallelism with Up to 64 Issues Per Clock



The Parallelism for a Variety Programs with up to 64 instruction Issues Per Clock



Deep Thinking

- It is too optimistic
 - It gives a bound for further implementations
 - No issue restrictions implies memory references
- Unfortunately, it is quite difficult to bound the performance of a processor with reasonable issue restrictions
 - Possible space is quite large
 - Existence of issue restrictions requires an accurate instruction scheduler
 - Studying processors with large numbers of issues very expensive implies higher ILP

Deep Thinking (Cont'd)

- Cache misses and nonunit latencies have not been taken into account, and both these effects may have significant impact!
- Loop-level parallelism existing in FP programs implies higher ILP
- For integer programs, branch prediction, register renaming, and less parallelism are all crucial limitations

Deep Thinking (Cont'd)

- Transaction processing and web servers heavily depend on the integer performance
- Designers face a challenge in deciding how best to use the limited resources available on a integrated circuit
- One of the most interesting trade-offs is between simpler processors
 - Larger caches and higher clock rates versus
 - More emphasis on instruction-level parallelism with a slower clock and smaller caches

Example

- Consider the following three hypothetical, but not atypical, processors, which we run with the SPEC gcc benchmark:
 - A simple MIPS two-issue static pipe running at a clock rate of 1 GHz and achieving a pipeline CPI of 1.0. This processor has a cache system that yields 0.01 misses per instruction.
 - A deeply pipelined version of MIPS with slightly smaller caches and a 1.2 GHz clock rate. The pipeline CPI of the processor is 1.2, and the smaller caches yield 0.015 misses per instruction on average.

Example (Cont'd)

- A speculative superscalar with a 64-entry window. It achieves one half of the ideal issue rate measured for this window size. This processor has the smallest caches, which leads to 0.02 misses per instruction, but it hides 10% of the miss penalty on every miss by dynamic scheduling. This processor has a 800-MHz clock.

Assume that the main memory time (which sets the miss penalty) is 100 ns. Determine the relative performance of these three processors.

Answer

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

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

We need to compute the miss penalties for each system:

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

Answer (Cont'd)

The clock cycle times for the processors are 1 ns, 0.83 ns, and 1.25 ns, respectively. Hence, the miss penalties are

$$\text{Miss penalty}_1 = \frac{100 \text{ ns}}{1 \text{ ns}} = 100 \text{ cycles}$$

$$\text{Miss penalty}_2 = \frac{100 \text{ ns}}{0.83 \text{ ns}} = 120 \text{ cycles}$$

$$\text{Miss penalty}_3 = \frac{0.9 \times 100 \text{ ns}}{1.25 \text{ ns}} = 72 \text{ cycles}$$

Applying this for each cache:

$$\text{Cache CPI}_1 = 0.01 \times 100 = 1.0$$

$$\text{Cache CPI}_2 = 0.015 \times 120 = 1.8$$

$$\text{Cache CPI}_3 = 0.02 \times 72 = 1.44$$

Answer (Cont'd)

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

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

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

$$\text{CPI}_1 = 1.0 + 1.0 = 2.0$$

$$\text{CPI}_2 = 1.2 + 1.8 = 3.0$$

$$\text{CPI}_3 = 0.22 + 1.44 = 1.66$$

Since this is the same architecture we can compare instruction execution rates to determine relative performance:

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

$$\text{Instruction execution rate}_1 = \frac{1000 \text{ MHz}}{2} = 500 \text{ MIPS}$$

$$\text{Instruction execution rate}_2 = \frac{1200 \text{ MHz}}{3.0} = 400 \text{ MIPS}$$

$$\text{Instruction execution rate}_3 = \frac{800 \text{ MHz}}{1.66} = 482 \text{ MIPS}$$

Beyond the Limits

- These is divided into two classes:
 - limitations that arise even for the perfect speculative processor
 - limitations that arise for one or more realistic models
- Limitations that apply even to the perfect model are:
 - WAR and WAW hazards through memory
 - Unnecessary dependences
 - Overcoming the data flow limit (Value prediction)

Beyond the Limits (Cont'd)

- Two ideas expose more ILP:
 - Address value prediction and speculation predicts memory address values and speculates by reordering loads and stores
 - Speculating on multiple paths

Hardware vs Software Speculation Mechanisms

- Run time disambiguation is difficult at compiler time
- Hardware-based speculation works better when control flow is unpredictable
- Hardware-based speculation maintains a completely precise exception model
- Compiler can result in better code scheduling
- Hardware-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementation
- Hardware-based speculation increases the complexity and cost

Multithreading: Using ILP Support to Exploit TLP

- ILP may very limited or hard to exploit in some applications. For example,
 - An online transaction-processing system has natural parallelism among the multiple queries and updates that are presented by requests
 - Scientific applications
 - Embedded applications like router processor
- This higher-level parallelism is called thread-level parallelism (TLP)

Thread Level Parallelism

- This higher level parallelism is called thread level parallelism
- A thread is a separate process with its own instructions and data
- Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute

Thread Level Parallelism (Cont'd)

- Unlike ILP, TLP is explicitly represented by the use of multiple threads of execution that are inherently parallel
- TLP could be more cost-effective primarily
- It is likely that ILP-based approaches will continue to be the primary focus for desktop-oriented processors

Using an ILP Datapath to Exploit TLP

- TLP and ILP exploit two different kinds of parallel structure in a program
- A processor oriented at ILP exploits TLP
- A datapath designed to exploit higher amounts of ILP, will find that functional units are often idle because of either stalls or dependences in the code
- Could TLP be used to keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

Multithreading

- It takes advantage of TLP
- It is a method for exploiting thread level parallelism
- It allows multiple threads to share the functional units of a single processor in an overlapping fashion
- Multiple threads are being executed with in single processor by duplicating the thread-specific state (program counter, registers, and so on)

Fine-Grained Multithreading

- It switches between threads on each instruction, causing the execution of multiple threads to be interleaved
- This interleaving is often done in a round-robin way, skipping any idle threads
- Thus the CPU must switch threads on every clock cycle
- A key advantage is that it can hide the throughput loss arising from stalls
- The disadvantage is that it slows down the execution of the individual threads

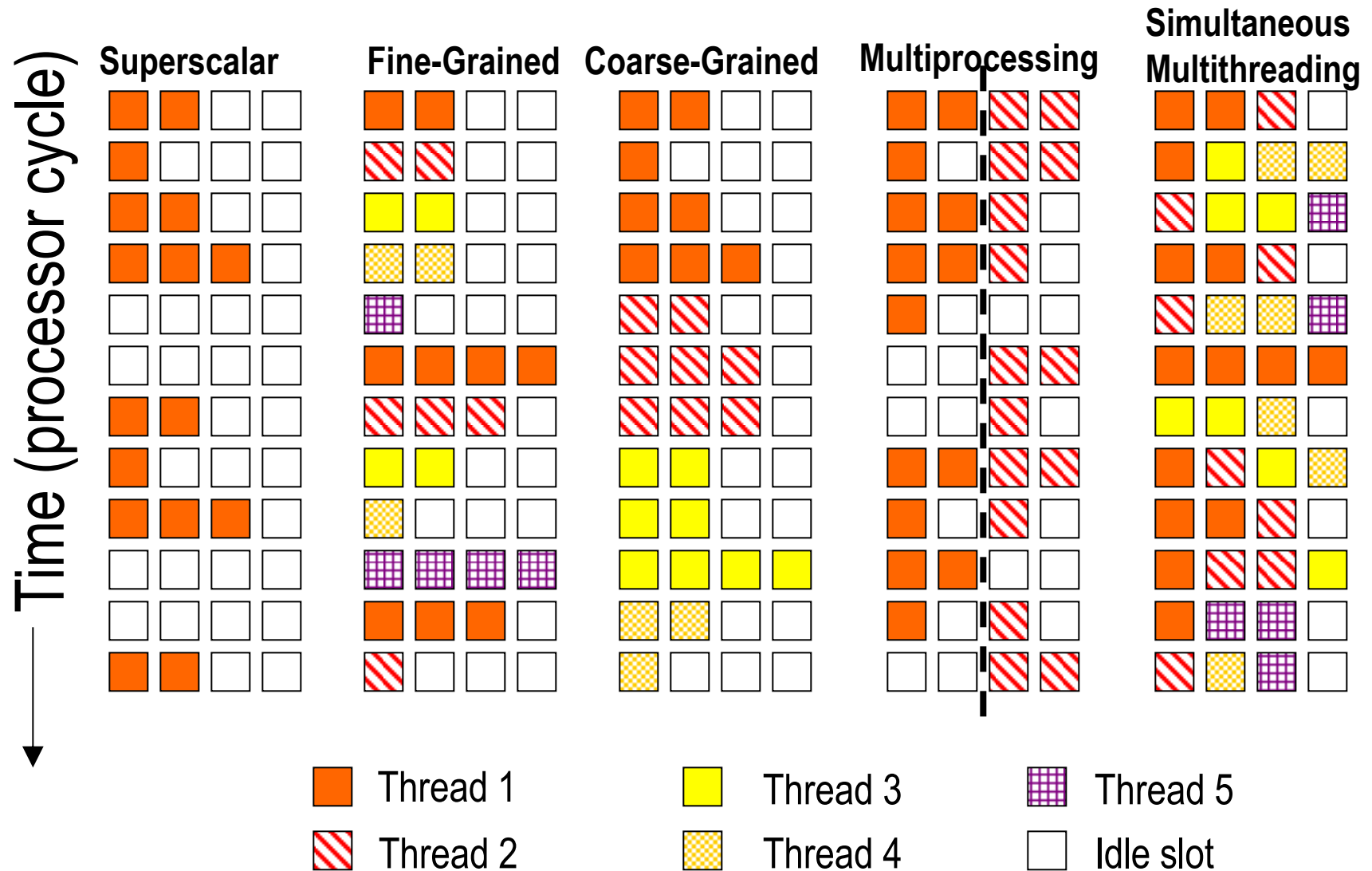
Coarse-Grained Multithreading

- It switches threads only on costly stalls, such as level 2 cache misses
 - It relieves the need to have thread-switching be essentially free and is much less likely to slow the processor down
- But it is limited to overcome throughput loss, especially from shorter stalls due to the pipeline start-up costs arising from thread-switching

Simultaneous Multithreading (SMT)

- It is a variation on multithreading to use the resources of a multiple-issue, dynamically scheduled processor to exploit TLP and ILP
- The motivation is that modern multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use
- With register renaming and dynamic scheduling, multiple instructions from independent threads can be issues without regard to the dependencies among them

Superscalar with Various Multithreadings



Design Challenges in SMT

- SMT will be unlikely to gain much performance if it were coarse-grained
- It makes sense only in a fine-grained implementation; thus fine-grained scheduling is very critical
- The effect can be minimized by having a preferred thread
- With a preferred thread, some performance may be sacrificed when encountering a stall
- Throughput is maximized by having a sufficient number of independent threads

Design Challenges in SMT (Cont'd)

- Unfortunately, mixing many threads will inevitably compromise the execution time of individual threads
- Similar problems exist in instruction fetch
- For current machines that issue four to eight instruction per cycle, it probably suffices to have a small number of active threads, and a even smaller number of preferred threads

Design Challenges in SMT (Cont'd)

- Challenges
 - Dealing with a larger register file needed to hold multiple contexts
 - Scheduling is challenging
 - Cache and TLB conflicts generated by SMT do not cause significant performance degradation
- Two important observations
 - For small multithreading, simple choices are good
 - Significant enhancement can be performed when the performance is low at the cost of some overhead

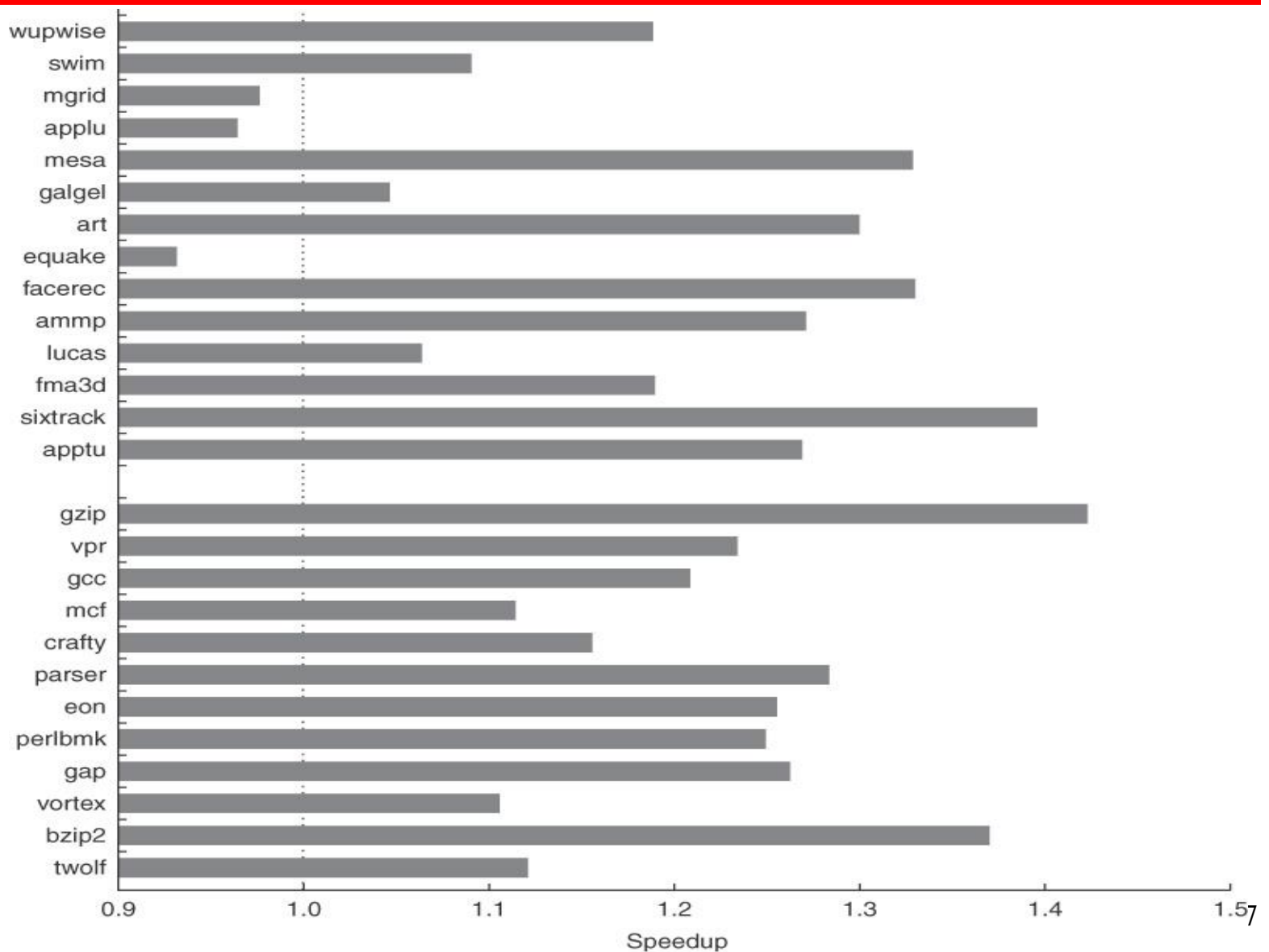
Design Challenges in SMT (Cont'd)

- Some support are needed (IBM Power5 as an example):
 - Increasing the associativity of L1 instruction cache and the instruction address translation buffers
 - Adding per-thread load and store queues
 - Increasing the size of the L2 and L3 caches
 - Adding separate instruction prefetch and buffering
 - Increasing the number of virtual registers from 152 to 240
 - Increasing the size of several issue queues

Potential Performance Advantages from SMT

- How much performance can be gained by implementing SMT?
- Gains for multiprogrammed workloads of two or more times are unrealistic. For example
 - Pentium 4 Extreme
 - IBM Power5

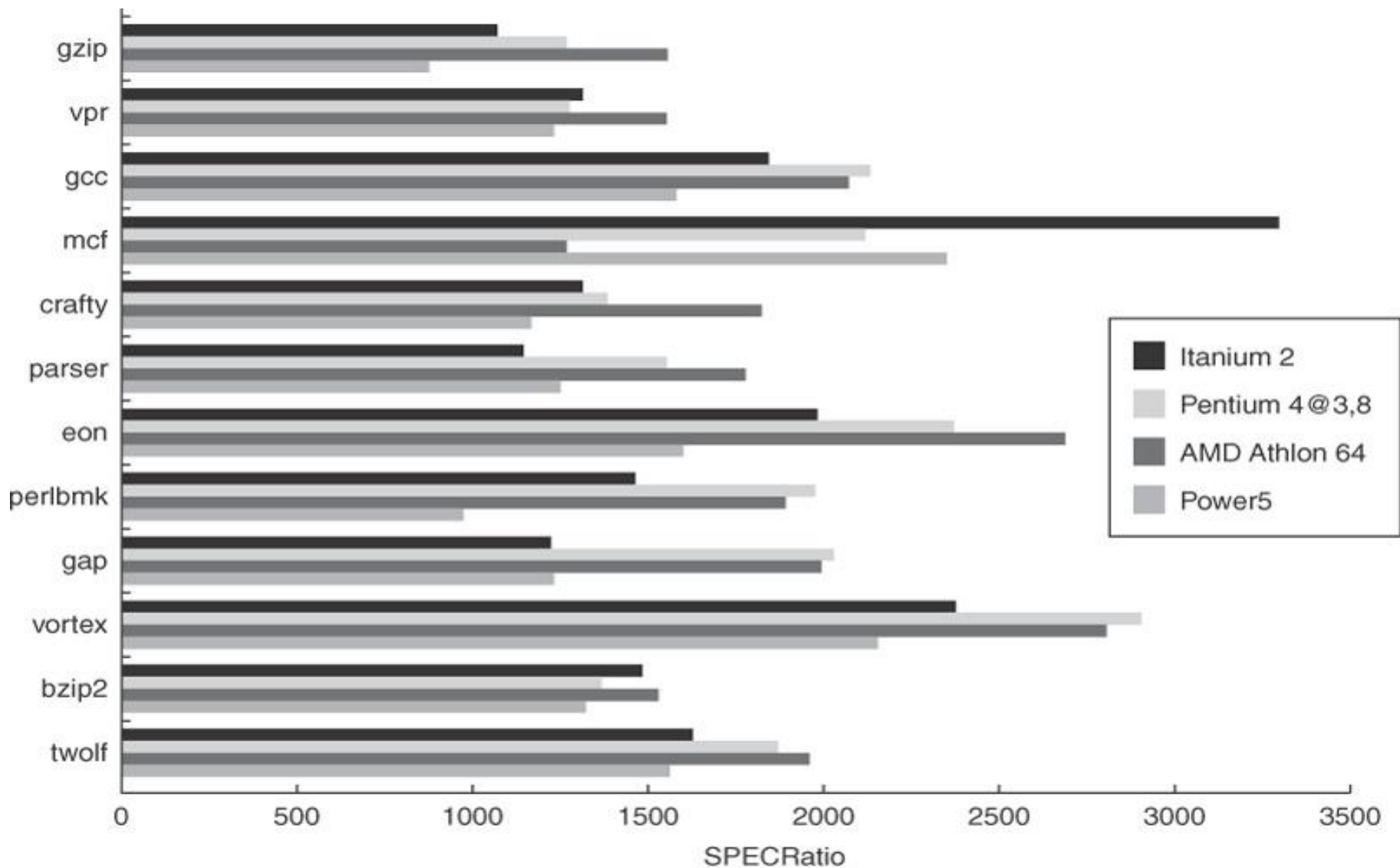
Performance Comparison fro SMT and ST on 8-processor Power5 Multiprocessor



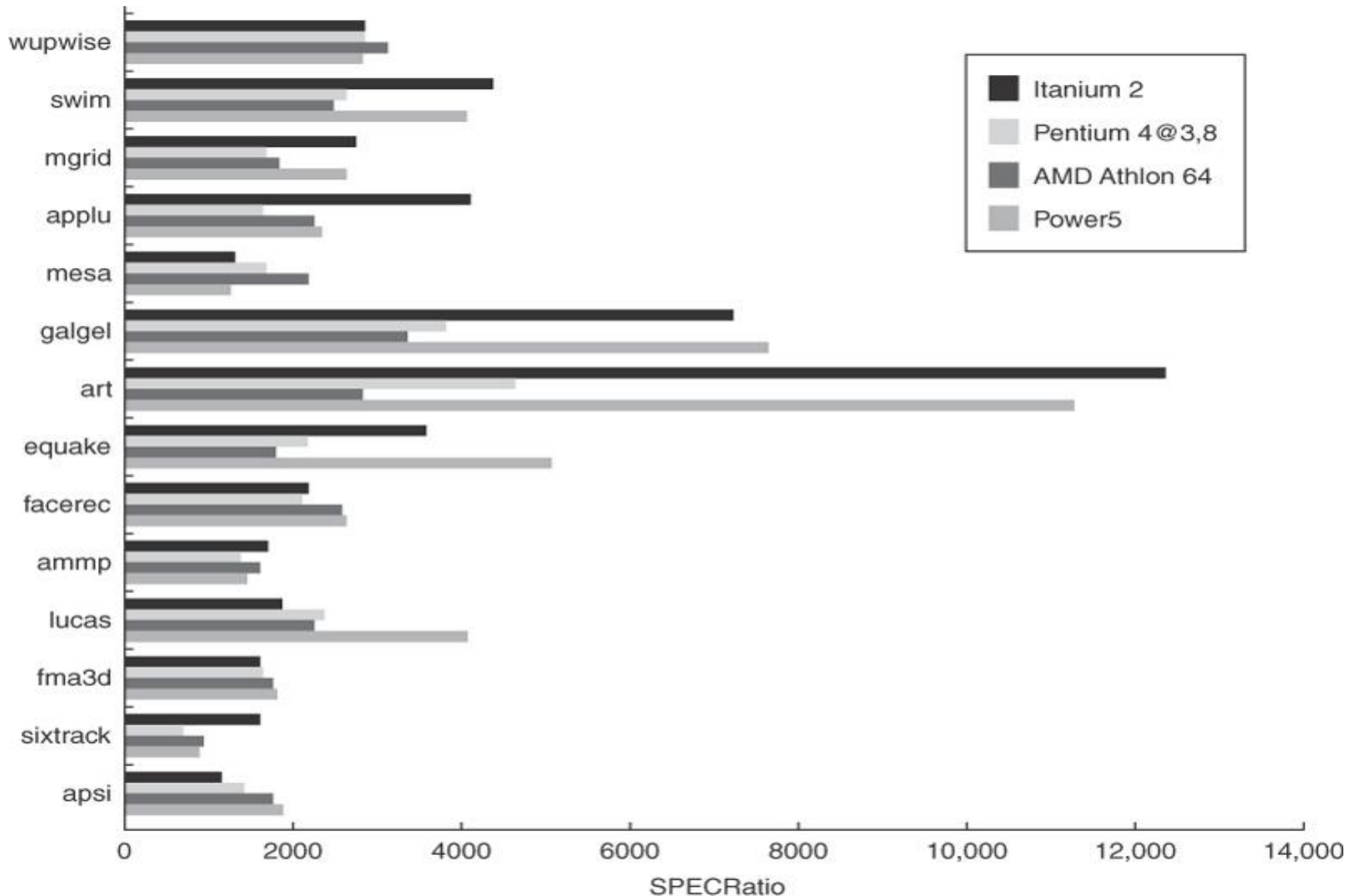
Performance and Efficiency in Advanced Multiple-Issue Processors

Processor	Micro architecture	Fetch / Issue / Execute	FU	Clock Rate (GHz)	Transistors Die size	Power
Intel Pentium 4 Extreme	Speculative dynamically scheduled; deeply pipelined; SMT	3/3/4	7 int. 1 FP	3.8	125 M 122 mm ²	115 W
AMD Athlon 64 FX-57	Speculative dynamically scheduled	3/3/4	6 int. 3 FP	2.8	114 M 115 mm ²	104 W
IBM Power5 (1 CPU only)	Speculative dynamically scheduled; SMT; 2 CPU cores/chip	8/4/8	6 int. 2 FP	1.9	200 M 300 mm ² (est.)	80W (est.)
Intel Itanium 2	Statically scheduled VLIW-style	6/5/11	9 int. 2 FP	1.6	592 M 423 mm ²	130 W

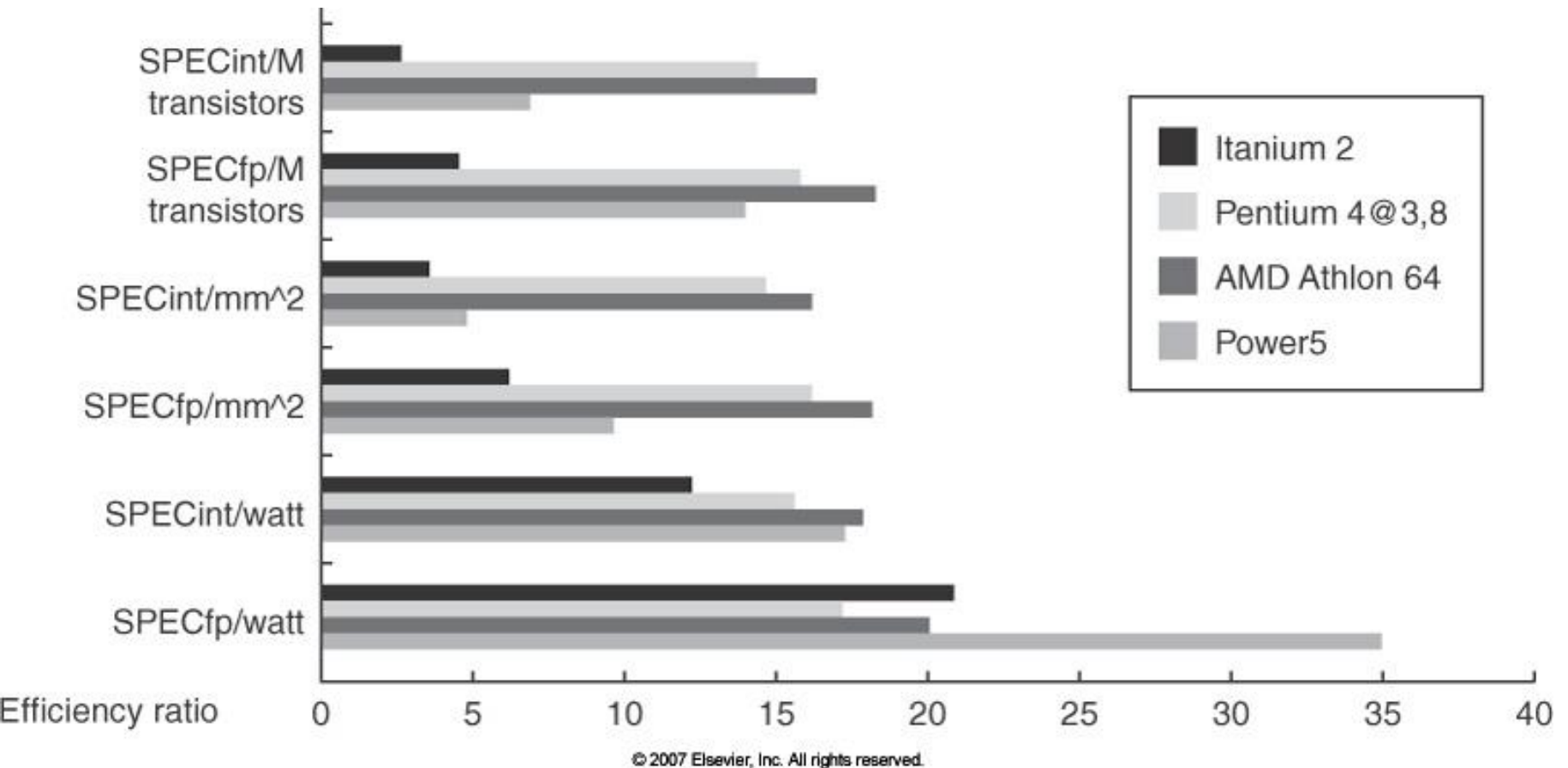
Performance Comparison for SPECint 2000



Performance Comparison for SPECfp 2000



Efficiency Measures for Multiple-Issue Processors



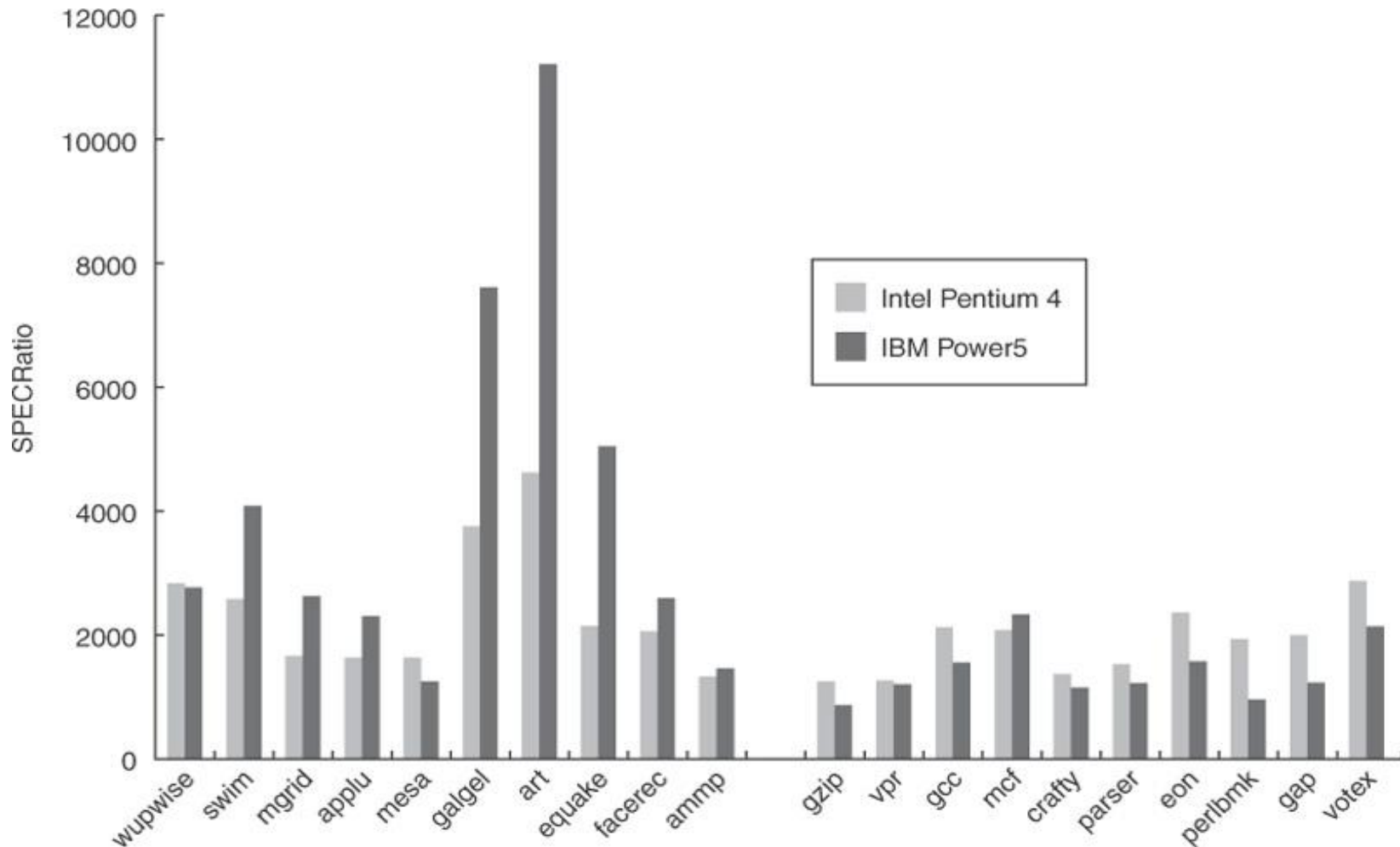
What Limits Multiple-Issue Processors?

- There is a tradeoff between the requirements of multiple-issue design and clock rate
- Modern microprocessors are primarily power limited
- Is a technique energy efficient?
 - Issuing multiple instructions incurs some overhead in logic that grows faster than the issue rate grows
 - The most important is the growing gap between peak issue rates and sustained performance
- Imperfect speculation is energy inefficient
- What about focusing on improving clock rate?
 - Deeper pipelines incur penalties and higher switching rates

Fallacy

- Processors with lower CPIs will always be faster.
- Processors with faster clock rates will always be faster
- Although a lower CPI is certainly better, sophisticated pipelines typically have slower clock rates than processors with simple pipelines
- In APs 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

Fallacy (Cont'd)



© 2007 Elsevier, Inc. All rights reserved.

Pitfall

- Sometimes bigger and dumber is better
- Advanced pipelines have focused on novel and increasingly sophisticated schemes for improving CPI
 - The 21264 uses a 29 Kbits sophisticated tournament predictor
 - the earlier 21164 uses a simple 2-bit predictor with 2K entries (or a total of 4 Kbits)
- For the SPEC95 benchmarks, the more sophisticated branch predictor of the 21264 outperforms the simpler 2-bit scheme on all but one benchmark

Pitfall

- On average, for SPECInt95, the 21264 has 11.5 mispredictions per 1000 instructions committed while the 21164 has about 16.5 mispredictions
- Somewhat surprisingly, the simpler 2-bit scheme works better for the transaction processing workload than the sophisticated 21264 scheme (17 mispredictions vs. 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

Pitfall (Cont'd)

- The transaction processing workload has a very large code size with a large branch frequency
 - The ability of the 21164 predictor to hold twice as many branch predictions based on purely local behavior (2K vs. the 1K local predictor in the 21264) seems to provide a slight advantage
- Different applications can produce different behaviors