ECE 486/586
Computer Architecture

Prof. Mark G. Faust

Maseeh College of Engineering
and Computer Science

PORTLAND STATE
UNIVERSITY

Instruction Level Parallelism and Its Dynamic Exploitation

- Reading:
  - Hennessy & Patterson: Chapter 2
- Homework:

ILP

- Instruction Level Parallelism
  - Evaluation/execution of instructions in parallel
  - Pipeline exploits ILP by recognizing that not all of the data path resources are in use all the time by each instruction – separate stages
- Static
  - Use software to identify and exploit parallelism
    - Re-arrange ("schedule") code to avoid hazards, exploit parallelism
  - Tends to be used in embedded processors
  - Intel IA-64 architecture is an exception
- Dynamic
  - Use hardware to identify and exploit parallelism
  - Penalty is increased complexity, additional logic, die area
  - Benefits
    - Binary compatibility (no tight coupling between S/W and H/W)
    - Many dependences not known at compile time (memory)
    - Some delays are unpredictable (e.g. cache misses)
  - Techniques dominate processors in desktop and server markets
    - Intel Pentium III, 4, AMD Athlon, MIPS R10000/12000, Sun UltraSPARC III, Power PC 603, Mac G3, G4

ILP

- "Basic block" – compiler term refers to code between control structures (i.e. no branches in except at entry, no branches out except at exit)
- Must be able to exploit parallelism across basic blocks
  - MIPS programs average dynamic branch frequency is 15% - 25%, meaning a branch is executed between every 4 and 7 instructions on average
  - Instructions within a basic block likely to have dependences (e.g. successor instruction awaiting write of register in previous instruction for its operand)

Saving registers

```
sort:   addi    $sp,$sp, -20        # make room on stack for 5 registers
        sw      $ra, 16($sp)        # save $ra on stack
        sw      $s3,12($sp)         # save $s3 on stack
        sw      $s2, 8($sp)         # save $s2 on stack
        sw      $s1, 4($sp)         # save $s1 on stack
        sw      $s0, 0($sp)         # save $s0 on stack
```

Procedure body

Move parameters		move	$s2, $a0	# copy parameter $a0 into $s2 (save $a0)
		move	$s3, $a1	# copy parameter $a1 into $s3 (save $a1)
Outer loop		move	$s0, $zero	# i = 0
	for1tst:	slt	$t0, $s0, $s3	# reg $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
		beq	$t0, $zero, exit1	# go to exit1 if $s0 ≥ $s3 (i ≥ n)
Inner loop		addi	$s1, $s0, -1	# j = i - 1
	for2tst:	slti	$t0, $s1, 0	# reg $t0 = 1 if $s1 < 0 (j < 0)
		bne	$t0, $zero, exit2	# go to exit2 if $s1 < 0 (j < 0)
		sll	$t1, $s1, 2	# reg $t1 = j * 4
		add	$t2, $s2, $t1	# reg $t2 = v + (j * 4)
		lw	$t3, 0($t2)	# reg $t3 = v[j]
		lw	$t4, 4($t2)	# reg $t4 = v[j + 1]
		slt	$t0, $t4, $t3	# reg $t0 = 0 if $t4 ≥ $t3
		beq	$t0, $zero, exit2	# go to exit2 if $t4 ≥ $t3
Pass parameters and call		move	$a0, $s2	# 1st parameter of swap is v (old $a0)
		move	$a1, $s1	# 2nd parameter of swap is j
		jal	swap	# swap code shown in Figure 2.34
Inner loop		addi	$s1, $s1, -1	# j -= 1
		j	for2tst	# jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1	# i += 1
		j	for1tst	# jump to test of outer loop

Restoring registers

```
exit1:  lw      $s0, 0($sp)         # restore $s0 from stack
        lw      $s1, 4($sp)         # restore $s1 from stack
        lw      $s2, 8($sp)         # restore $s2 from stack
        lw      $s3,12($sp)         # restore $s3 from stack
        lw      $ra,16($sp)         # restore $ra from stack
        addi    $sp,$sp, 20         # restore stack pointer
```

Procedure return

```
        jr      $ra                 # return to calling routine
```

Loop-Level Parallelism

- Simplest and most common is to exploit parallel iterations of a loop

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

- Opportunity for each loop iteration to operate in parallel but no opportunity within loop
- Vector processors designed for this but long out of vogue
- Perhaps a comeback for graphics, DSP and multimedia applications

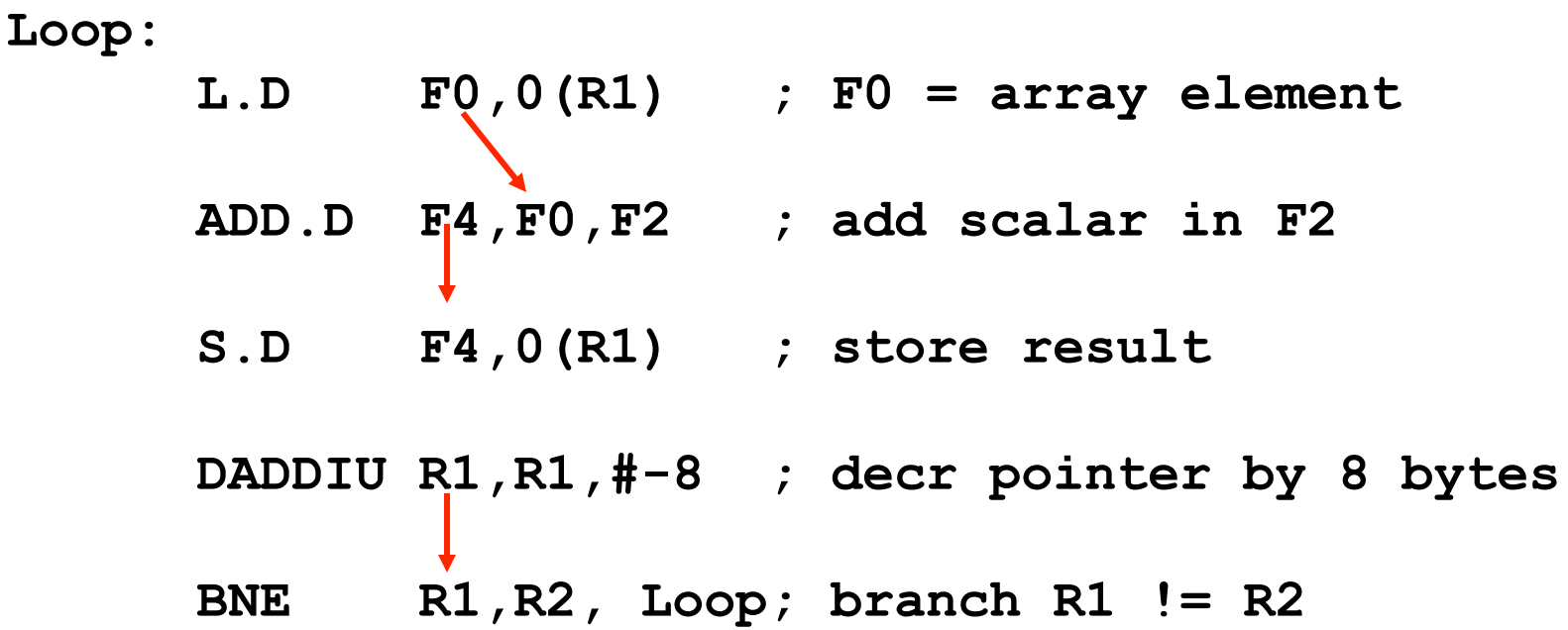
Data Dependences and Hazards

- Dependences between instructions limit parallelism
- Two instructions are “parallel” if there are no dependences between them. Parallel instructions can execute simultaneously in a pipeline with no stalls
- If two instructions are not parallel they must be executed in order though they may partially overlap

Dependences

- Three types of dependences
  - Data ("true data dependence")
  - Name dependences
  - Control dependences
- Data dependences
  - Instruction j is "data dependent on" i if
    - Instruction i produces a result used by instruction j, or
    - Instruction j is data dependent on instruction k which is data dependent on instruction i (transitive)

Data Dependences



Data Dependences

- Data dependences are properties of programs; they exist regardless of the implementation, pipeline details, etc
- Whether the dependence results in a detected hazard and whether that hazard causes a pipeline stall are consequences of the pipeline organization
- Example:

```
DADDIU  R1,R1,#-8    ; decr pointer by 8 bytes
BNE     R1,R2,Loop   ; branch R1 != R2
```

- Has a data dependence (RAW) through R1
- In original MIPS 5-stage pipeline the sequence wouldn't cause a stall because branch condition testing was in EX stage and result R1 would be available
- In revised MIPS pipeline, we moved condition testing to ID stage, creating a hazard for this example since R1 won't be available until end of EX phase occurring concurrently (even assuming forwarding/bypassing)

Data Dependences

- Because data dependences can have such severe consequences on performance of pipeline we want to overcome these limitations to exploiting ILP
- Scheduling is the primary method for doing this
  - Maintain dependence, but avoid the hazard
  - Eliminate dependence by transforming the code
- Dependence through registers is easier to detect than through memory… why?
  - Do 20(R4) and 20(R4) refer to the same memory location?
    - Not if R4 is incremented (auto-increment or otherwise)
  - Do 100(R4) and 20(R6) refer to the same memory location?
    - Need to do effective address computation to be sure

Name Dependences

- Consider two instructions, i and j where i precedes j
  - An *anti-dependence* between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads
    - WAR
  - An *output dependence* occurs when instruction i and instruction j write the same register or memory location
    - WAW
- Must preserve original ordering to ensure correctness
- But no data is being transferred between instructions
- Had compiler chosen different register names, there'd be no dependence (hence "name" not "data" dependence)
- Why might a compiler do this?
  - Register allocation (small number of general purpose registers)

Register Renaming

- Consider the following code sequence

```
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
```

True data dependence
Anti-dependence
Output dependence

- Using register renaming (S,T) we obtain

```
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
```

- Eliminating anti-dependence and output dependence and therefore possibility of WAR and WAW hazards

Dependence and Hazards

- A hazard is created when there's a dependence between instructions and they're close enough together (dynamically!) that the overlap caused by pipelining or the re-ordering of instructions would change order of access to the operands involved in the dependence
  - RAW (true data dependence)
  - WAW (output dependence)
  - WAR (anti-dependence)

Dependence and Hazards

- RAW (True data dependence)
- Most common. Must preserve program order
- MIPS 5-stage pipeline: results in hazard if load immediately precedes instruction attempting to use the register

```
LW    R1,0(R2)
ADD   R1,R1,#4
```

Dependence and Hazards

- WAW (output dependence)
- Only present in pipelines that do one or more of following
  - write in more than one stage
  - allow a later instruction to proceed when a previous instruction is stalled
  - include parallel execution units (e.g. integer and FP)
- MIPS 5-stage integer pipeline avoids this hazard because writes occur only in WB
- Saw examples in Appendix A where these dependences can occur (FP multiply following by a load of a floating point value into same destination register)
- Need to concern ourselves with these if we're going to allow pipelines with potential to re-order instructions

Dependence and Hazards

- WAR (anti-dependence)
- Don’t occur in most static issue pipelines (even deep ones) because all reads are early (ID) and all writes are late (WB)
- Occur if instructions are re-ordered

Control Dependence

- Consider code segment

```
if p1 {
        S1;
};

if p2 {
        S2;
}
```

- S1 is “control dependent” on p1
- S2 is “control dependent” on p2
  - Not control dependent on p1 – assuming S1 doesn’t affect p2
  - If it did, there’d be a data dependence anyway

Control Dependence

- In general
  - If instruction i is control dependent on a branch it cannot be moved before the branch (so that it's no longer controlled by the branch)
  - If instruction i is not control dependent on a branch it cannot be moved after the branch (so that its execution is controlled by the branch)

Control Dependence

- Control dependence preserved by two properties of simple pipelines
  - Instructions execute in order: an instruction occurring before a branch is executed before the branch
  - Detection of control (branch) hazards ensures that an instruction control dependent on a branch isn't executed until the branch outcome is known
- But control dependence is not the critical property
  - May be willing to execute instructions we shouldn't (violating control dependence) provided we don't affect correctness of the program!
- Two properties **critical** to program correctness
  - data flow
  - exception behavior

Control Dependence

- Consider following code sequence

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

Assume no branch delay slot

- Obvious that if we don't maintain the data dependence involving R2 we can change the result of the program
- Less obvious is that if we ignore the control dependence and move the load instruction before the branch, we can cause a memory protection exception (important: no data dependence prevents us from interchanging BEQZ and LW, only control dependence)

Control Dependence

- Consider following code sequence

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

Assume no branch delay slot

- Value of R1 in **OR** instruction is dependent upon the branch
- It's data dependent on **DADDU** and **DSUBU** instructions but that's insufficient – preserving the order alone won't guarantee correct execution
- Need to ensure data flow (right R1 result to R1 in OR)

Control Dependence

- Consider following code sequence

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

- If register destination of **DSUBU** (R4) is unused after the instruction at label L ("dead") and the instruction is not capable of generating an exception, we can move **DSUBU** before the branch (violating control dependence, but preserving data flow)
- Worst case is that we'd unnecessarily execute an instruction
- This kind of scheduling based upon branch prediction is called "speculation"

Dynamic Scheduling

- Simple 5-stage pipeline of Appendix A[1]
  - In order issue and in order execution (if instruction stalls no subsequent instructions can begin execution)
  - Data dependence between closely spaced instructions results in hazard and stall

```
DIV.D   F0,F2,F4
ADD.D   F10,F0,F8   ; stalls on RAW for F0
SUB.D   F12,F8,F14  ; execution prevented
```

  - May leave execution units idle
  - Structural and data hazards (RAW) could both be checked in ID stage

[1]before scoreboard introduced

Dynamic Scheduling

- If we are to allow **`SUB.D`** to proceed we need to separate issue into two parts
  - Check for structural hazards
  - Await absence of any data hazards
- Still check for structural hazards at issue
  - In order issue (issue in program order)
  - Out of order execution
  - Out of order completion
  - Out of order execution introduces possibility of WAW and WAR hazards

Dynamic Scheduling

- Example

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

- Anti-dependence between **ADD.D** and **SUB.D** resulting in WAR hazard if pipeline executes **SUB.D** before **ADD.D** (which is waiting for **DIV.D**)
- Output dependence on F6 results in potential WAW

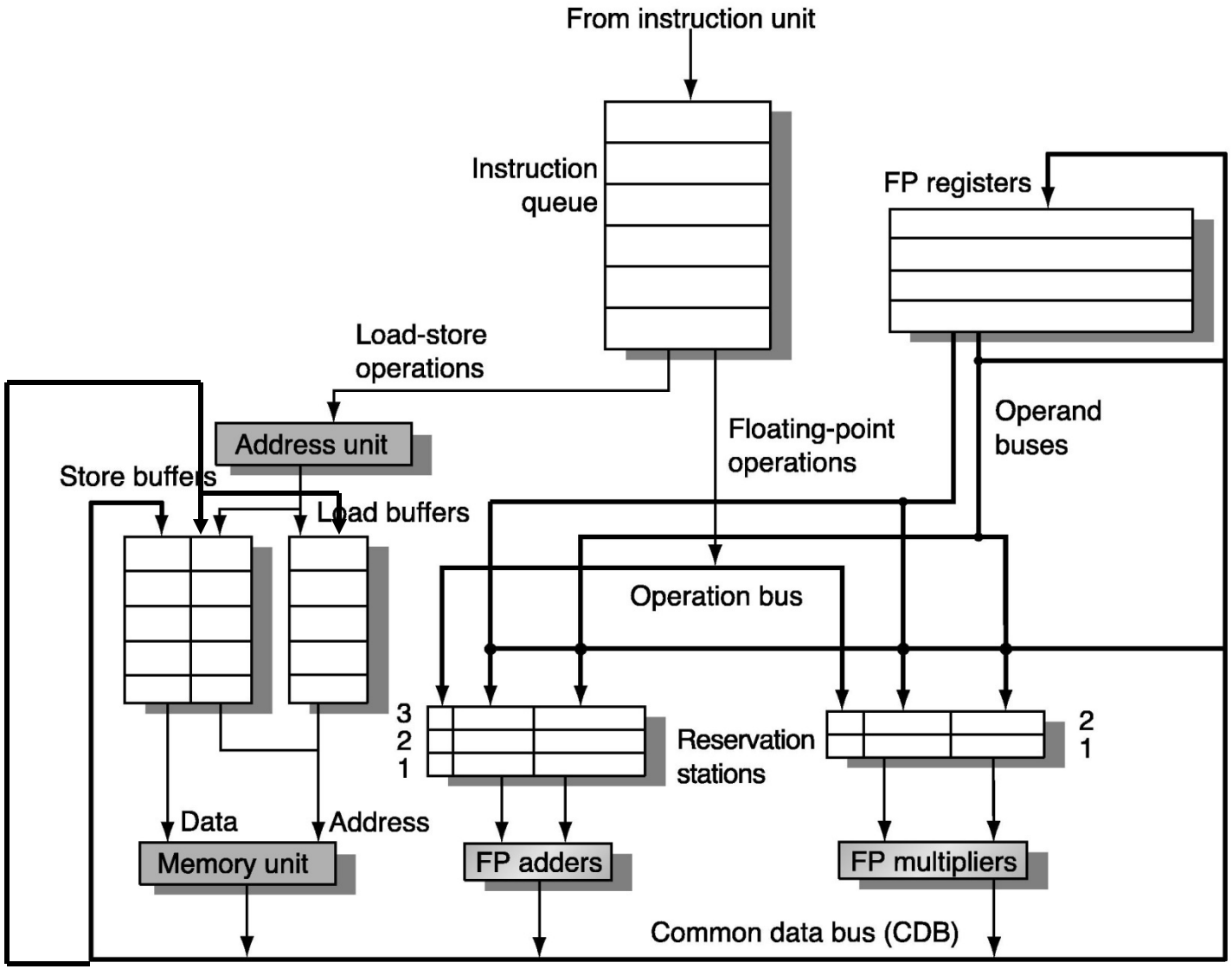
Exceptions

- Preserve exception behavior
  - Only allow those exceptions to occur that would have occurred had instructions executed sequentially
  - Exceptions must occur in same order as they would in absence of pipeline (issue even without dynamic scheduling)
- Can still be imprecise exceptions
  - Processor state not necessarily the same as if instructions executed sequentially
  - Undesirable since makes restart after exception difficulty

Tomasulo's Algorithm

- Used in IBM 360/91 (1967!)
  - Employed in floating point unit
  - Before caches available in commercial processors
  - Minimize RAW hazards
  - Uses register renaming to eliminate WAR and WAW hazards
  - Designed for register-memory architecture
    - Easily handles register-register architecture
  - Designed for pipelined functional unit
    - Effectively equivalent to multiple functional units

Tomasulo's Algorithm



Tomasulo's Algorithm

- Issue
  - Get next instruction from head of queue
  - If matching reservation station empty and operands available
    - Issue instruction to reservation station with values
  - If not an empty reservation station then structural hazard exists
    - Stall instruction
  - If empty reservation station but operands not available, keep track of functional units producing them
    - Issue instruction to reservation station indicating FU that will provide operands
  - Effectively "renames" registers, eliminating WAR and WAW hazards

Tomasulo's Algorithm

- Execute
  - If one or more operands unavailable, monitor CDB for them
  - When an operand becomes available, it's placed in corresponding reservation station
  - When all operands are available, operation can be executed at the functional unit
  - Several instructions could become ready in same clock cycle for the same functional unit
  - Arbitrary choice OK for non-memory access functional units
  - Addresses RAW hazards

Tomasulo's Algorithm

- Write Result
  - When result is available, write it on the CDB
  - CDB distributes to register file and all reservation stations
  - Stores write data to memory during this step

Load/Store Buffers

- Load Buffers
  - Hold components of effective address until it's computed
  - Track outstanding loads waiting on memory
  - Hold results of completed loads waiting on CDB
- Store Buffers
  - Hold components of effective address until its computed
  - Hold destination addresses for stores waiting for data value
  - Hold address and value until memory is available

Load/Store Operations

- Proceed in Two Steps
  - Compute effective address when base register is available
  - Effective address is placed in the load or store buffer
  - Loads in load buffer execute as quickly as memory is available
  - Stores in store buffer must wait for value being stored

Tags

- Register names are effectively replaced with tags
  - Identify reservation station which will produce operand
  - Values broadcast on CDB include the tag
    - Reservation stations with pending instructions awaiting operands monitor CDB for tags and values

Data Structures

- Reservation Station
  - Op — operation to perform on source operands
  - Qj, Qk — reservation stations that will produce source operands (0 → source operand already in V field or not required)
  - Vj, Vk — actual value of source operands if available
  - A — used for effective address calculation for loads/stores (initially holds immediate field of instruction, holds effective address once calculated)
  - Busy — indicates reservation station is busy and FU occupied
- Register Status
  - Qi — Reservation station that will write this register

Data Structures

		Instruction status		
Instruction		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			

Tomasulo's Algorithm Details

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].Qi=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};` rt
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;`

How Does Tomasulo's Algorithm Differ from Basic Scoreboard?

- Data structures are distributed among reservation stations rather than centrally located
- Common Data Bus (CDB) permits bypass rather than await writes to register file
  - Bandwith issues if multiple execution units complete in same cycle
- Scoreboard stalls instruction until all operands are available while Tomasulo's algorithm can assign the instruction to an execution unit (reservation station) and operands are read from CDB as they become available (has effect of eliminating WAR because source operand of earlier instructions will be read if available).
- Scoreboard stalls instruction to prevent WAW hazard while Tomasulo's algorithm effectively renames the output register and allows the instruction to proceed
  - Overwrites RegisterStat[rd] even if earlier instruction recorded its FU there
  - OK because subsequent instructions which needed it recorded the FU producing it
  - When FU of earlier instruction writes register to CDB
    - Reservation stations needing it, retrieve it
    - Register file not written since RegisterStat[reg] is no longer that FU

Difference: WAW

Scoreboard: stall

Instruction status	Wait until	Bookkeeping
Issue	Not Busy [FU] and not Result [D]	Busy[FU]←yes; Op[FU]←op; Fi[FU]←D; Fj[FU]←S1; Fk[FU]←S2; Qj←Result[S1]; Qk← Result[S2]; Rj← not Qj; Rk← not Qk; Result[D]←FU;

Tomasulo's Algorithm: register rename!

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].Qi=r;

Difference: WAR

Scoreboard: stall WB until register to be written is read by any instruction waiting for it

Write result	$\forall f$((Fj[f] ≠ Fi[FU] or Rj[f] = No) & (Fk[f] ≠ Fi[FU] or Rk[f] = No))	$\forall f$(if Qj[f]=FU then Rj[f]←Yes); $\forall f$(if Qk[f]=FU then Rk[f]←Yes); Result[Fi[FU]]← 0; Busy[FU]← No

Tomasulo's Algorithm: read registers immediately if available

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].Qi=r;

Loops and Tomasulo's Algorithm

- Example

```
LOOP: L.D     F0,0(R1)
      MUL.D   F4,F0,F2
      S.D     F4,0(R1)
      DADDIU  R1,R1,#-8
      BNE     R1,R2,LOOP
```

- Multiplies each element in an array by a scalar (F2) using pointer arithmetic (pointer in R1), decrementing pointer by size of double (8 bytes) until end of array reached (signaled by pointer to one element past end of array).
- Assume branches all taken

Loops and Tomasulo's Algorithm

Integer ADDs completed

Multiple iterations of loop in flight in pipeline!

Addresses for L.D

```
LOOP:   L.D     F0,0(R1)
        MUL.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDIU  R1,R1,#-8
        BNE     R1,R2,LOOP
```

Instruction status

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

Reservation stations

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

Register status

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

Where are we?

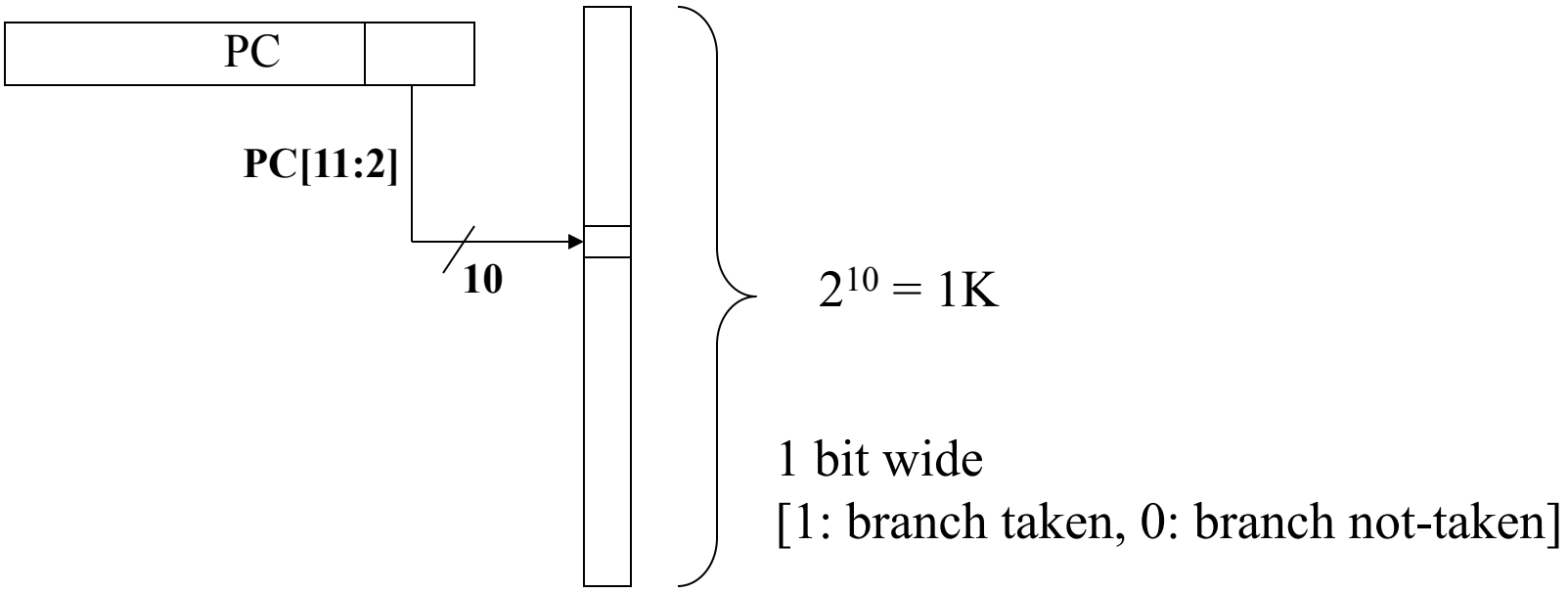
- Identified and categorized data hazards
- Effective means to reduce stalls due to data hazards
  - Static scheduling (compiler)
  - Forwarding/bypassing
  - Dynamic scheduling (out of order execution)
  - Register renaming
- Control dependences
  - Briefly discussed four static techniques
    - Stall until outcome known
    - Assume taken
    - Assume not-taken
    - Delayed branch (statically schedule branch slot)
  - Need to improve performance in face of control dependences
    - Branch instruction frequency
    - Impact on deep pipeline, out of order execution
  - Dynamic techniques

Branch Prediction

"Prediction is hard, especially when it involves the future."
-- Yogi Berra

Basic Branch Prediction

- Branch prediction buffer (branch history table)
- Memory indexed by low order bits of branch instruction address
- Stores previous branch outcomes to predict next outcome
- Memory is not tagged (unlike cache)
- Consequence: entry may reflect a different branch (aliased)



Prediction Performance

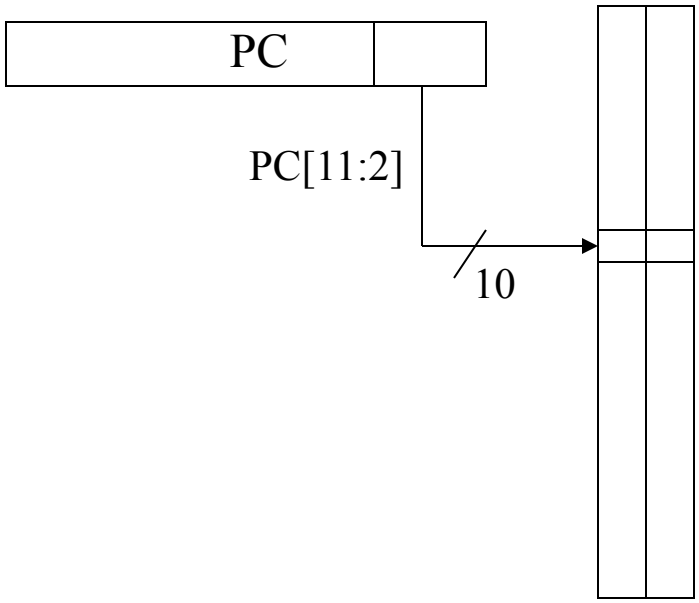
- Consider a `for` loop which is executed multiple (5) times
  - May be in a nested loop
  - May be in a function which is called repeatedly

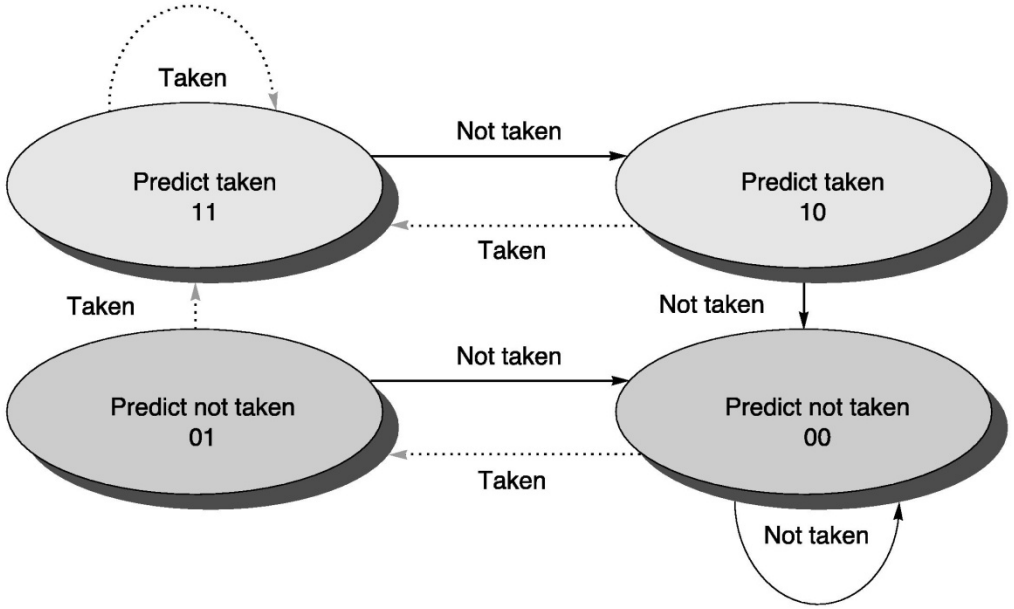
```
Outcome          11111011111011111011111101…
Prediction        11111011111011111011111110…
Misprediction    ....11....11....11....11....11…
```

Misprediction rate: 2/6 = 33%!

2-bit Prediction Scheme

- 2-bit prediction scheme remedies shortcoming of 1-bit scheme
- 2-bit entry in prediction buffer represents state machine
- Prediction must miss twice before it's changed





Prediction Performance

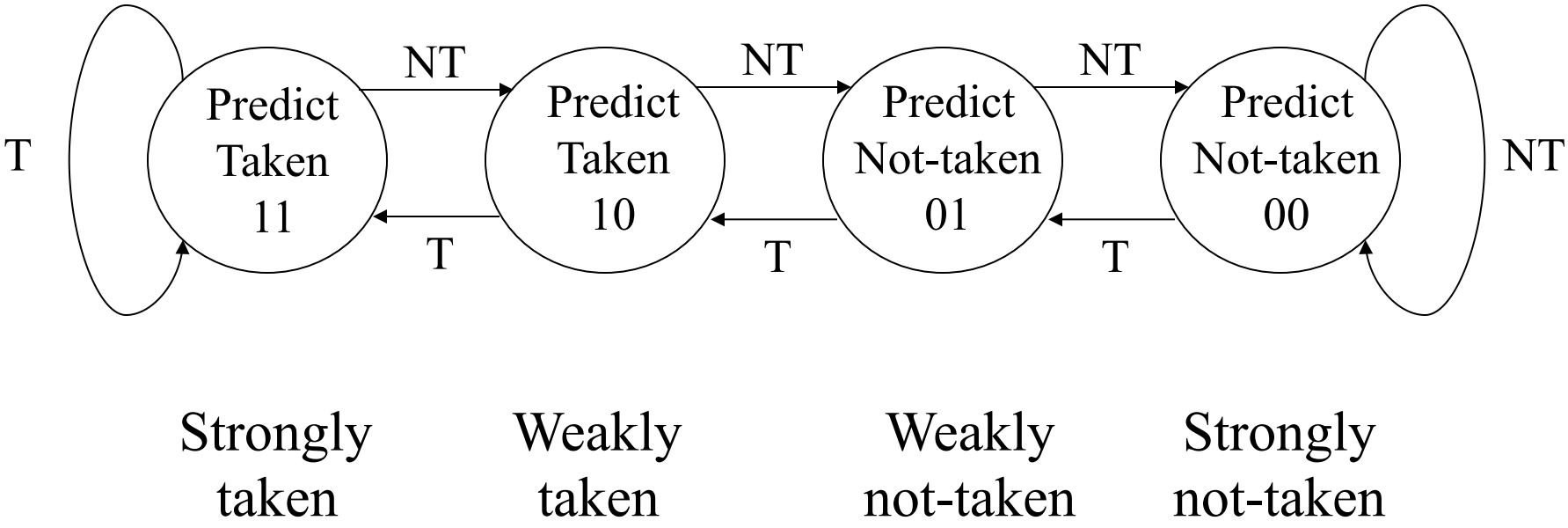
- Consider same for loop which is executed multiple (5) times

```
Outcome          11111011111011111011111101…
Prediction       11111111111111111111111111…
Misprediction     ....1.....1.....1.....1.....…
```

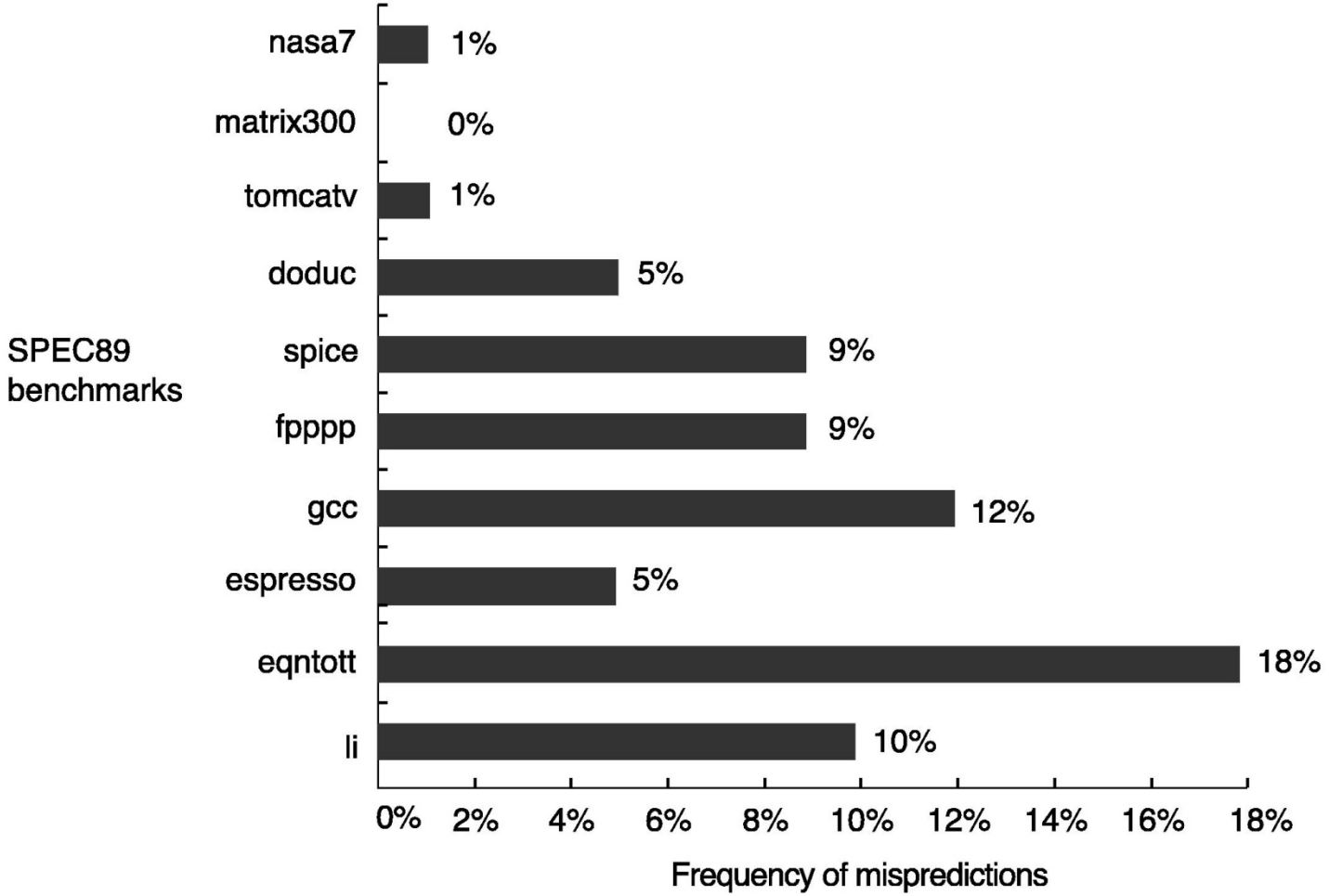
Misprediction rate: 1/6 = 16%

Prediction Performance

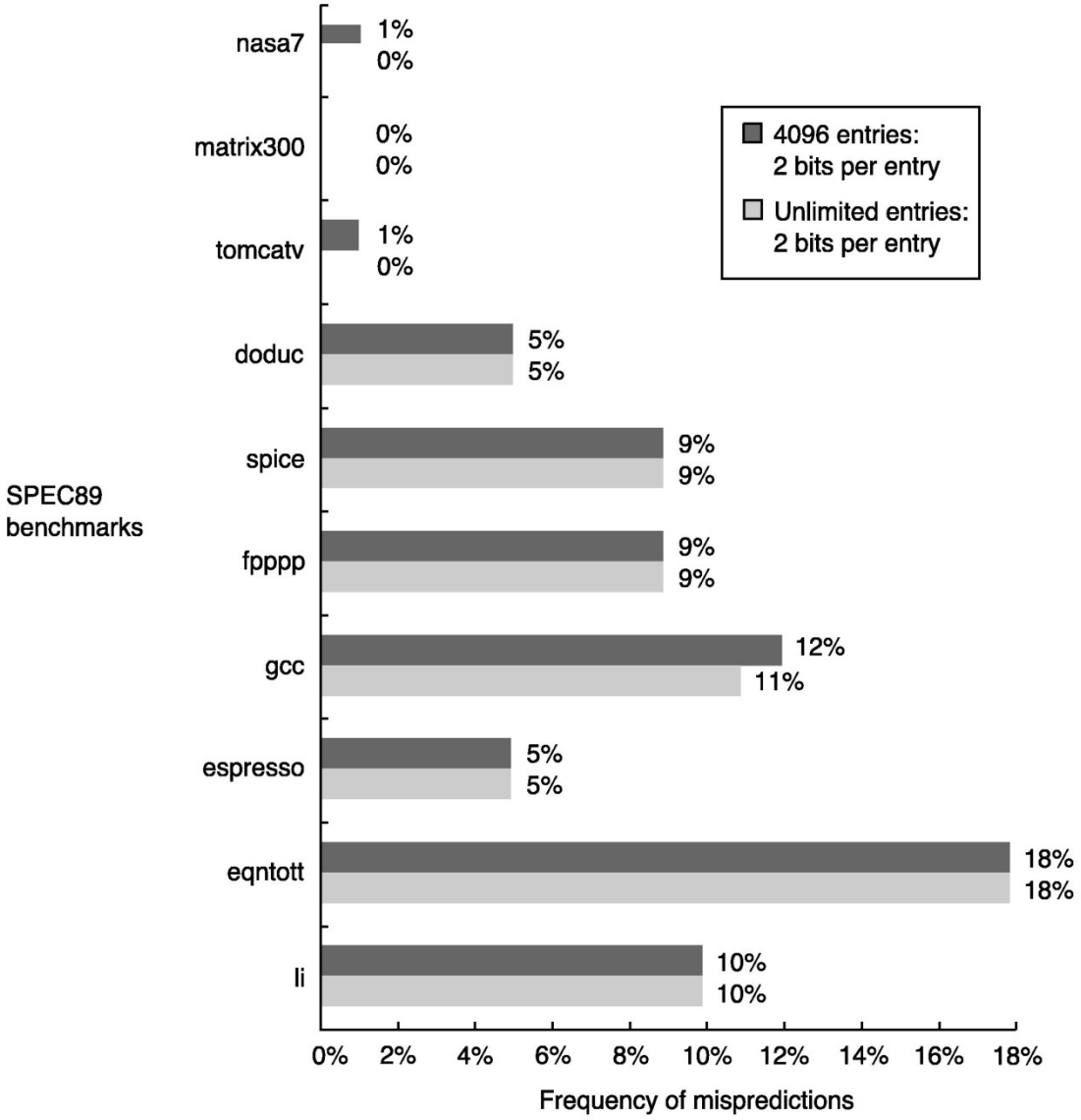
- A variation: 2-bit saturating counter



Prediction Accuracy of 4K 2-bit Prediction Buffer



Increasing Buffer Size Isn't the Solution



Correlating Branch Predictors

- Simple 2-bit prediction schemes use branch history of single branch (assuming no aliasing) to predict future behavior of that branch
- Behavior of other branches may have impact on current branch
- Outcomes of different branches often correlate

```
if (a == 2)
     a = 0;
if (b == 2)
     b = 0;
if (a != b){
     …
}
```

```
        DADDI   R3,R1,-#2
        BNEZ    R3,L1   ; a != 2
        DADD    R1,R0,R0
L1:     DADDI   R3,R2,-#2
        BNEZ    R3,L2   ; b != 2
        DADD    R2,R0,R0
L2:     DSUB    R3,R1,R2
        BEQZ    R3,L3   ; a == b
```

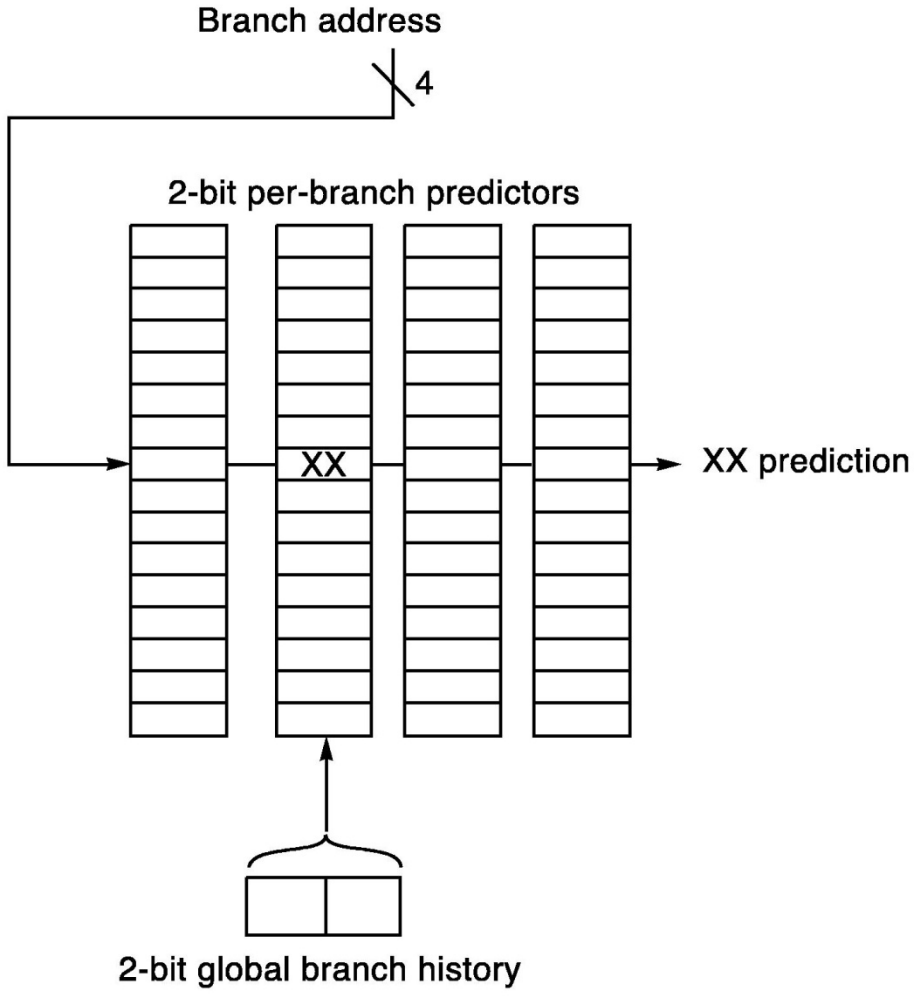
If first two branches untaken – third one taken

Correlating Branch Predictor with 2-bit Global History Register

Can extend branch history as n-bits, recording history of last n branches

Requires 2^n tables of length $2^{(\text{branch address bits used})}$

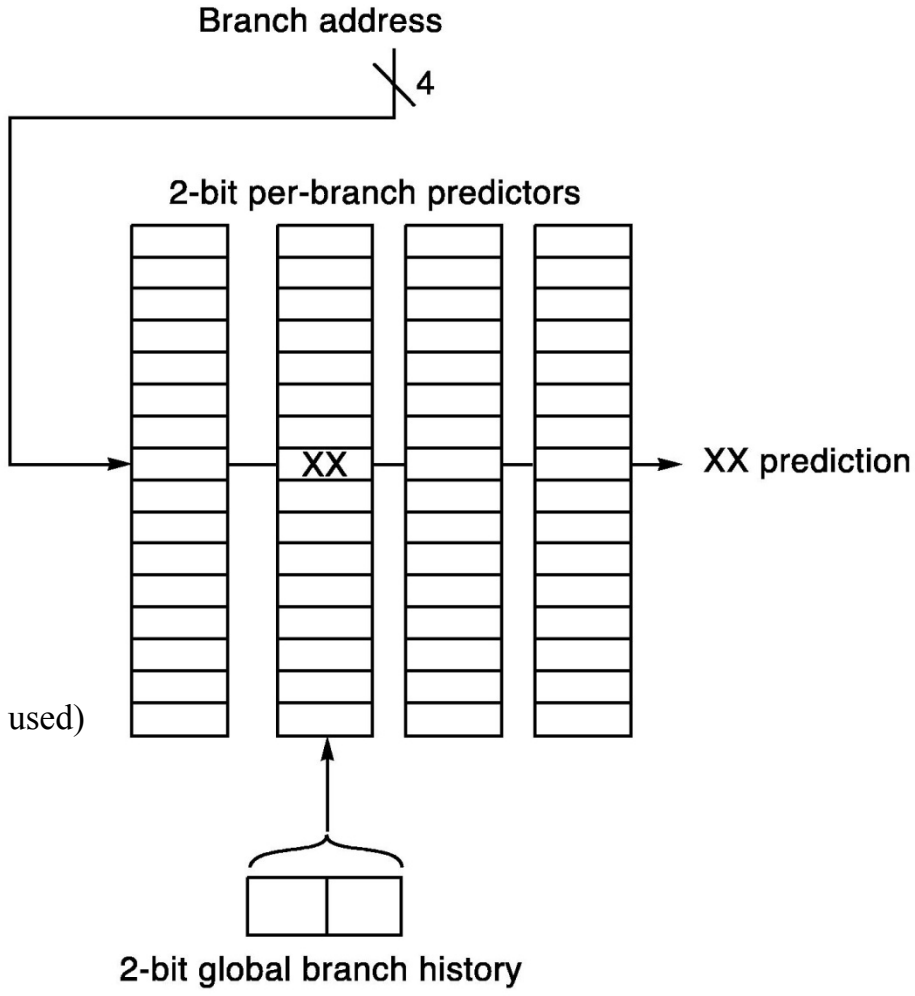
Can implement branch history as n-bit shift register



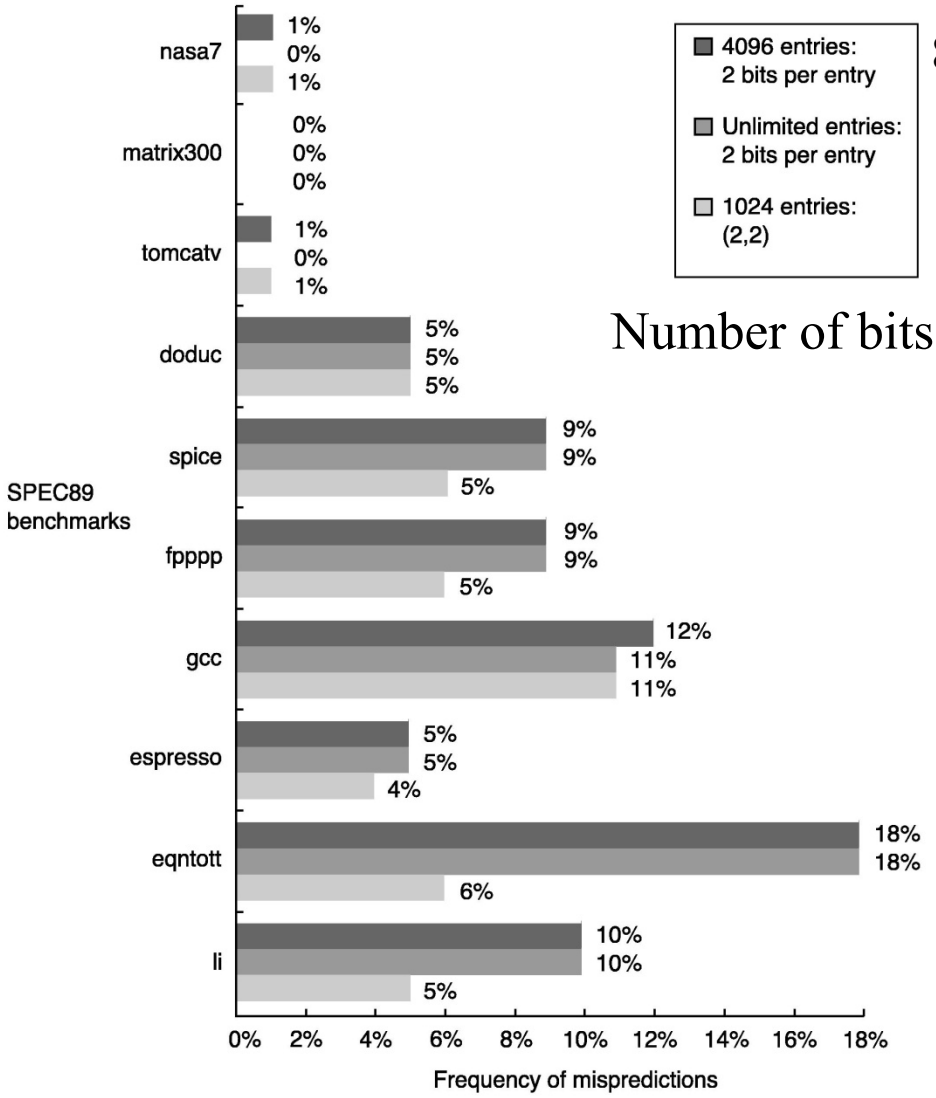
Correlating Branch Predictor with 2-bit Global History Register

(m,n) predictor uses behavior of last m branches to choose from 2^m branch predictors, each of which is an n bit predictor

Number of bits = 2^m x n x $2^{(\text{branch address bits used})}$



Comparison of 2-bit Predictors

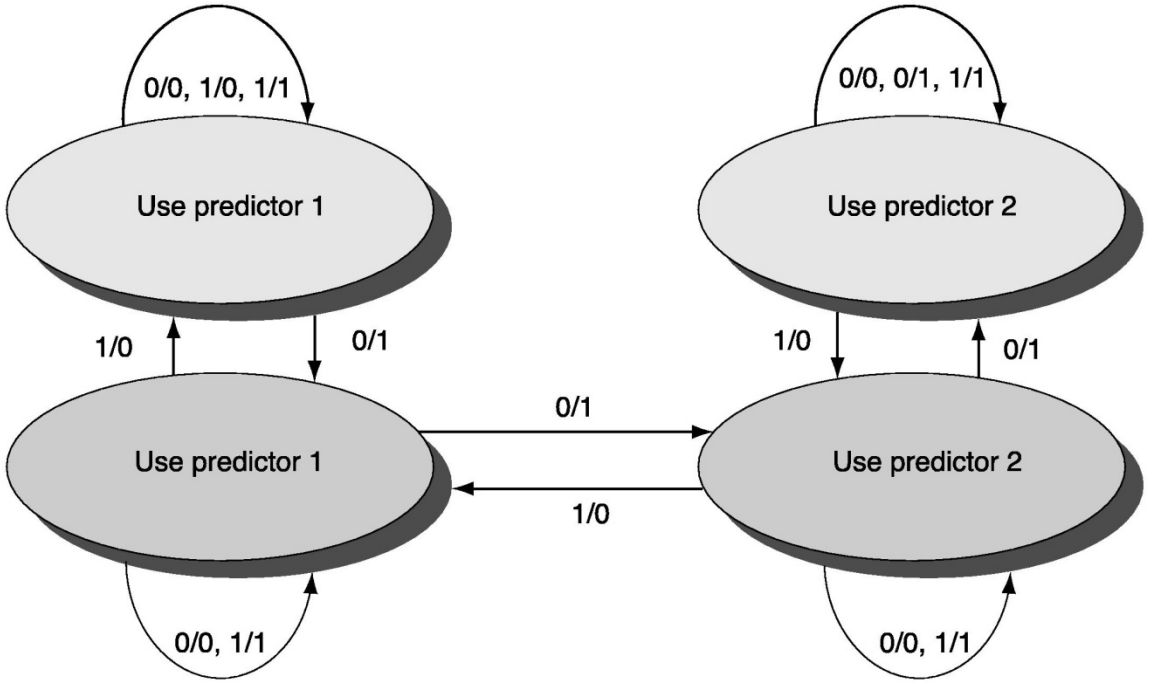


Number of bits = 2^m x n x $2^{(\text{branch address bits used})}$

Tournament Predictors: Adaptively Combining Local and Global Predictors

- Some branches are predicted more accurately with global predictors
- Some branches are better predicted with local predictors
- Combine both types of predictors and dynamically select the right predictor for the right branch
- The selector is yet another predictor with 2-bit state machine

Tournament Predictors: Adaptively Combining Local and Global Predictors



P1/P2 where P1 = 0 if predictor 1 mispredicted, 1 if correct

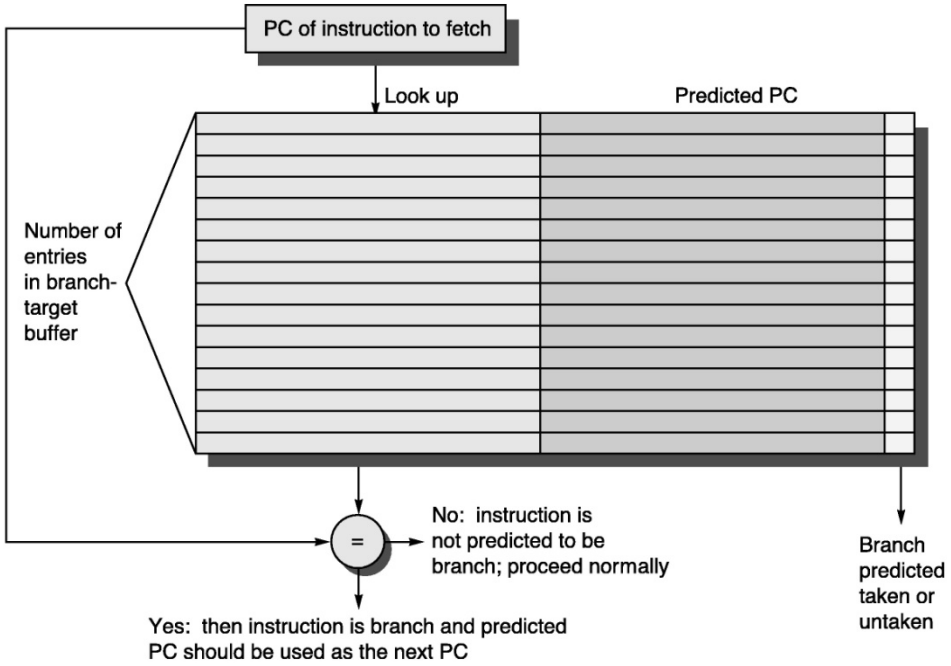
P2 = 0 if predictor 2 mispredicted, 1 if correct

Predicting Branch Targets

- To avoid branch penalty in 5-stage pipeline we need to know which address to fetch next instruction from before end of IF stage
- Requires us to know whether the (as-yet undecoded) instruction is a branch and, if so, what the next PC should be
- Branch prediction buffer is used during ID phase so that at end of ID we know branch target address (computed during ID), fall through address (computed during IF), and branch prediction
- Branch target buffer is used during IF phase to predict target address

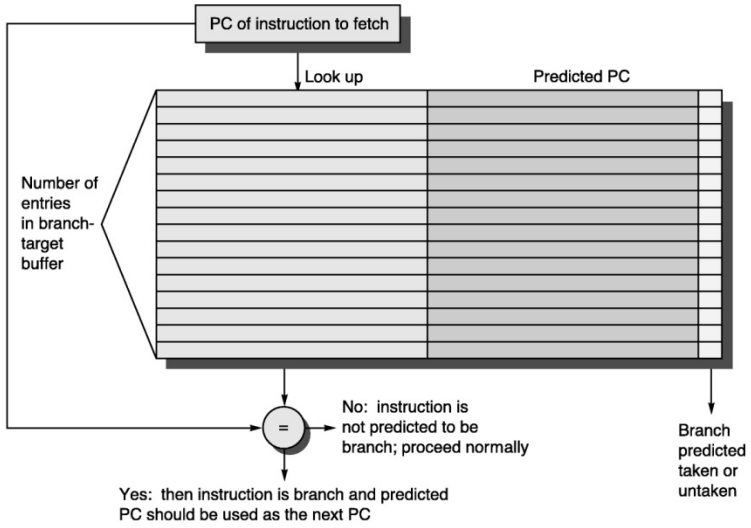
Predicting Branch Targets

- Branch target buffer is used during IF phase
- Uses PC of current instruction (possible branch) to search cache of predicted target PCs of branch
  - Any cache organization OK (e.g. direct-mapped, fully-associative)
  - But tag bits for full PC
- Fetch of predicted target begins at start of next cycle

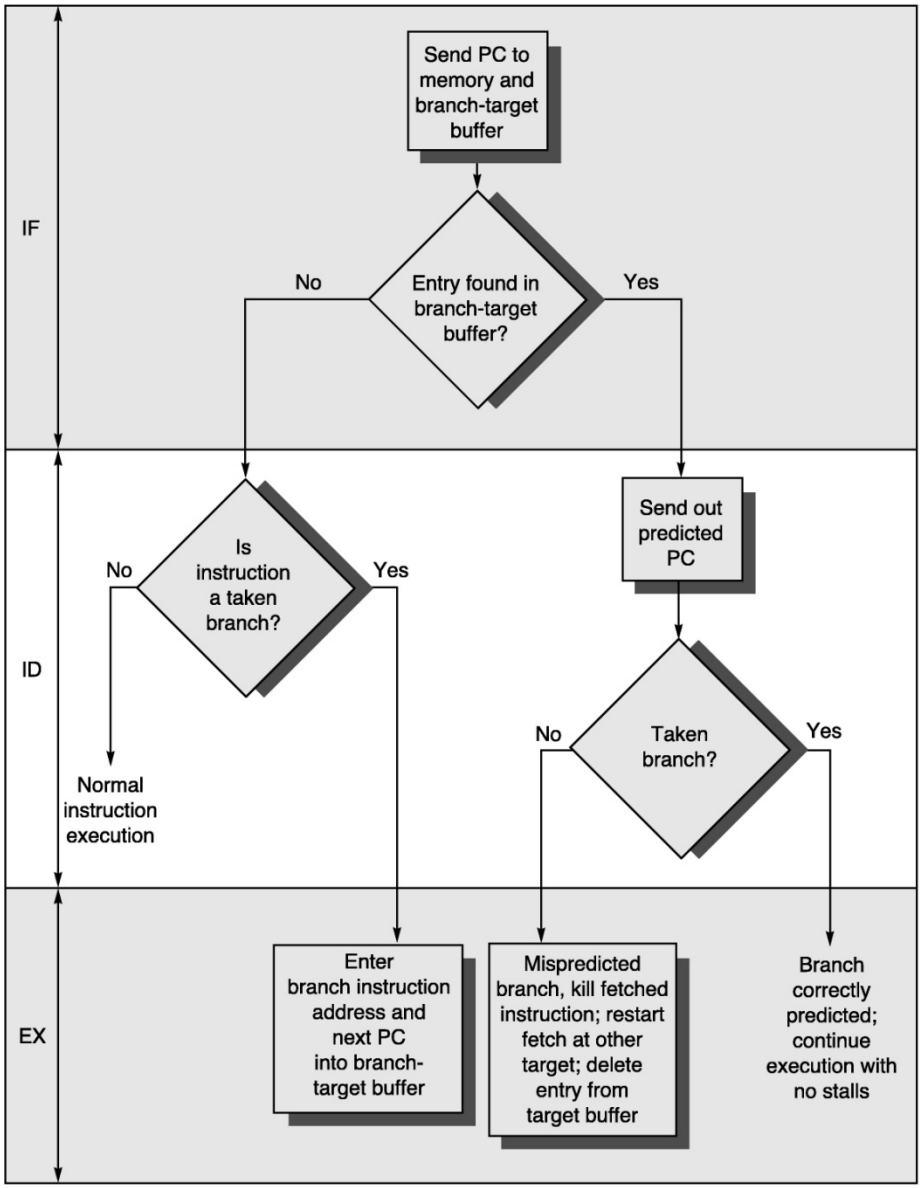


Predicting Branch Targets

- Unlike branch prediction buffer, we cannot permit aliasing but must match the PC (otherwise we would fetch predicted targets for non-branch instructions, impacting performance)
- If branch is later resolved to be not-taken, entry will be removed from target buffer
  - Fetch for (predicted) not-taken branch is same as non-branch: sequential
- If using two-bit predictor
  - Can retain in table but use prediction bits in table



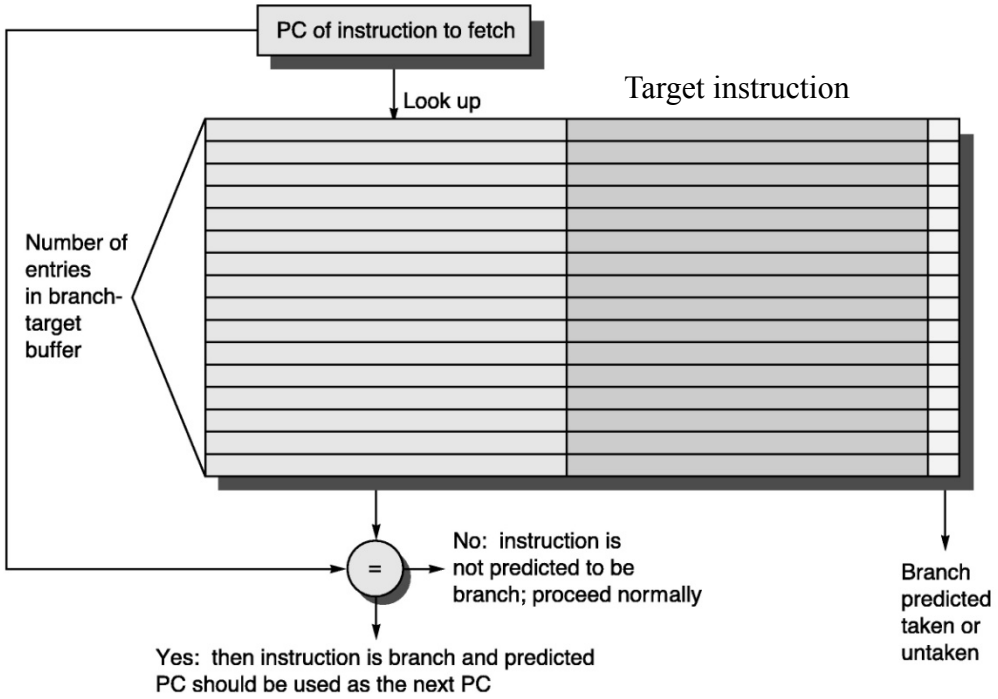
Branch Target Buffer Behavior



Instruction in Buffer	Prediction	Actual Branch	Penalty Cycles
Yes	Taken	Taken	0
Yes	Taken	Not-taken	2
No	-	Taken	2
No	-	Not-taken	0

Branch Folding

- Store the actual target instruction rather than its PC
  - Saves a memory fetch (cycle)
  - May permit a larger table (additional latency covered by savings)
- Zero-cycle unconditional branches
  - Branch target buffer signals hit and unconditional branch
  - Target instruction substituted for current instruction (branch)

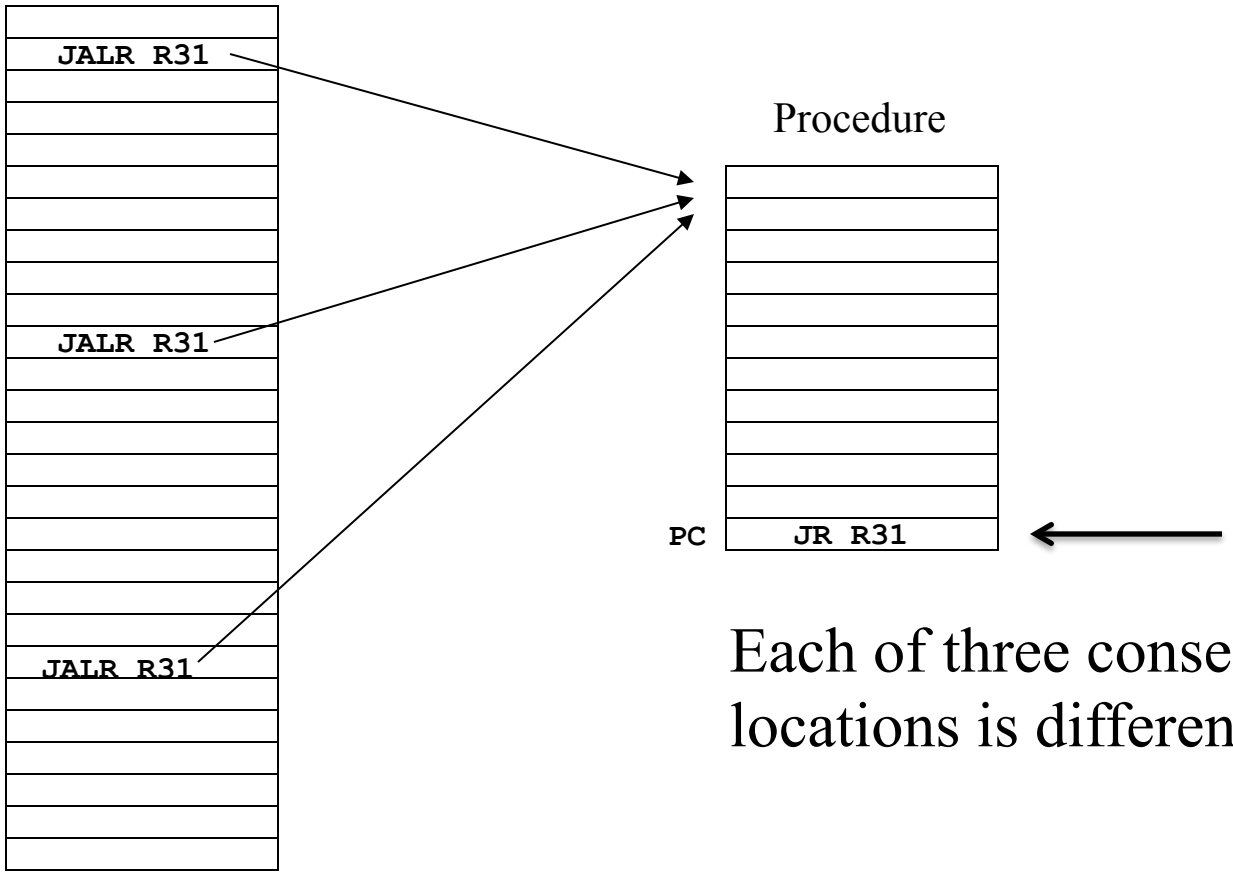


Can we do better?

- Have been predicting unconditional and conditional branches using PC-relative (immediate operand) mode
  - BEQ R1, R2, LOOP LOOP encoded as displacement(PC)
  - Target of the branch can be associated with the branch's address
- Can also branch (jump) where target address is in a register (indirect) mode
  - JMP R1
  - Target of branch can change as contents of R1 changes
  - Complicates branch prediction
  - Target of branch no longer (uniquely) associated with branch's address
- Although also used for dispatch tables, "long jumps", most indirect mode jumps are used for procedure calls

Returning from Procedure Calls

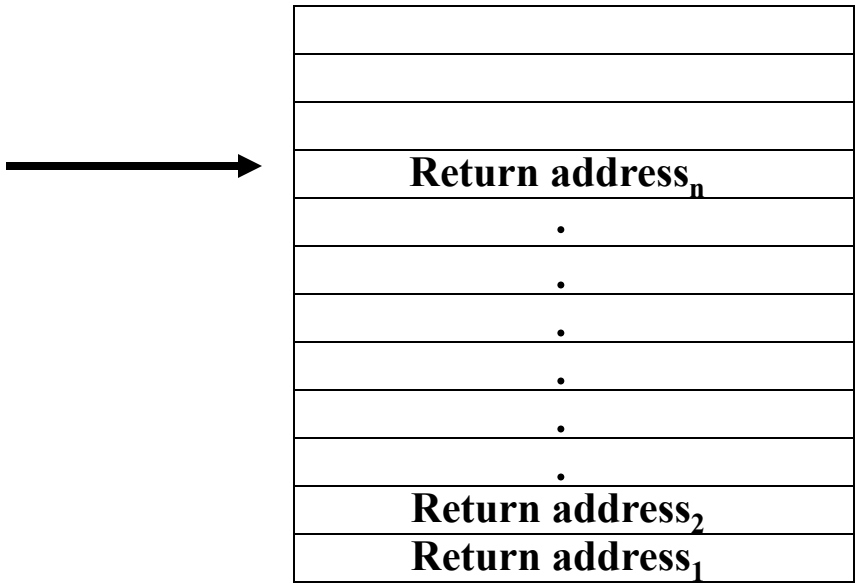
- Procedures return to their callers via a Jump instruction
  - Procedures may be called from different points in a program
  - Makes branch target prediction inaccurate if using previous return address



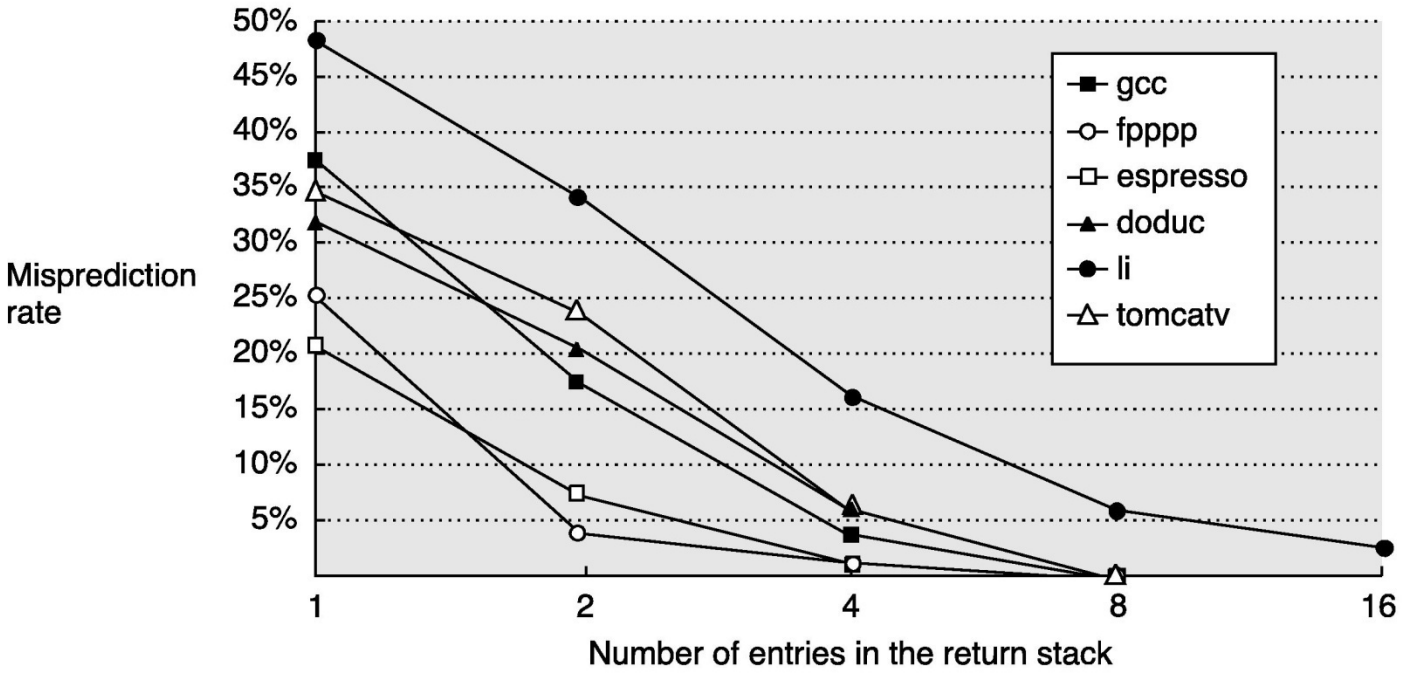
Each of three consecutive return locations is different for same address

Branch Prediction: Return Address Stacks

- When procedure call occurs (JALR), push return address on RAS
- When return instruction encountered, pop return address from RAS
- 100% “prediction” accuracy if stack is deep enough



Prediction Accuracy for Return Address Buffer (Arranged as Stack)



Returns account for average 81% of indirect jumps in these benchmarks

Multiple Issue

- Reducing data and control stalls using these techniques help us to approach ideal CPI = 1
- Further improvement requires CPI < 1
- Two primary alternatives
  - Superscalar
    - Statically (compiler) Scheduled (in-order execution)
    - Dynamically Scheduled (out-of-order execution)
    - Parallel pipelines, not just some parallel FUs
  - VLIW (Very Long Instruction Word)
    EPIC (Explicitly Parallel Instruction Computers)
    - Inherently Statically Scheduled

Superscalar Pipeline

- Requires issuing multiple instructions per clock cycle
  - Retrieve "packet" of instructions from fetch unit
  - Some of which may issue
  - What if earliest (sequential) instruction in packet is branch?
    - Don't issue following instruction
  - What if branch target is not first instruction in packet?
    - Don't issue prior instructions in packet

Instruction type	Pipe stages							
Integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	EX	EX	WB		
Integer instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	EX	EX	WB	
Integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	EX	EX	WB
Integer instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	EX	EX

Multiple Instruction Issue with Dynamic Scheduling

- Consider following simple code sequence (adds a scalar in F2 to each element of a vector in memory)

```
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
      DADDI    R1,R1,#-8   ; 8 bytes/double
      BNE      R1,R2,Loop  ; branch R1 != R2
```

Multiple Instruction Issue with Dynamic Scheduling

- Assume
  - A FP and integer operation can issue every clock cycle
  - Integer FU used for ALU operations and effective address calculations
  - Issue and Write take one cycle each
  - Dynamic branch prediction hardware
  - Separate FU to evaluate branch conditions
  - Latency for results (result available to succeeding instruction) is
    - Integer ALU: 1 cycle
    - Loads: 2 cycles
    - FP add: 3 cycles
  - Branches single issue
  - Branch prediction is perfect
    - But don't know outcome until EX

Multiple Issue Example

Iteration number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for L.D
1	S.D	F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU	R1,R1,#-8	2	4		5	Wait for ALU
1	BNE	R1,R2,Loop	3	6			Wait for DADDIU
2	L.D	F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D	F4,F0,F2	4	10		13	Wait for L.D
2	S.D	F4,0(R1)	5	8	14		Wait for ADD.D
2	DADDIU	R1,R1,#-8	5	9		10	Wait for ALU
2	BNE	R1,R2,Loop	6	11			Wait for DADDIU
3	L.D	F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD.D	F4,F0,F2	7	15		18	Wait for L.D
3	S.D	F4,0(R1)	8	13	19		Wait for ADD.D
3	DAADIU	R1,R1,#-8	8	14		15	Wait for ALU
3	BNE	R1,R2,Loop	9	16			Wait for DADDIU

Multiple Issue

- Consider same example but with multiple FUs so that effective address calculation and integer ALU can proceed in parallel

Iteration number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for L.D
1	S.D	F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU	R1,R1,#-8	2	3		4	Executes earlier
1	BNE	R1,R2,Loop	3	5			Wait for DADDIU
2	L.D	F0,0(R1)	4	6	7	8	Wait for BNE complete
2	ADD.D	F4,F0,F2	4	9		12	Wait for L.D
2	S.D	F4,0(R1)	5	7	13		Wait for ADD.D
2	DADDIU	R1,R1,#-8	5	6		7	Executes earlier
2	BNE	R1,R2,Loop	6	8			Wait for DADDIU
3	L.D	F0,0(R1)	7	9	10	11	Wait for BNE complete
3	ADD.D	F4,F0,F2	7	12		15	Wait for L.D
3	S.D	F4,0(R1)	8	10	16		Wait for ADD.D
3	DADDIU	R1,R1,#-8	8	9		10	Executes earlier
3	BNE	R1,R2,Loop	9	11			Wait for DADDIU

Hardware-Based Speculation

- In high performance pipeline with multiple issue, control dependency becomes bottleneck
- Branch prediction allows pipeline to partially continue until branch outcome known
- Instructions continue to be fetched/issued but cannot be executed until prior branch outcome known
- Stall fetched/decoded instructions until branch outcome known
- With multiple issue, this control dependency becomes a bottleneck
- Solution: “Speculatively” execute instructions based upon predicted branch outcome

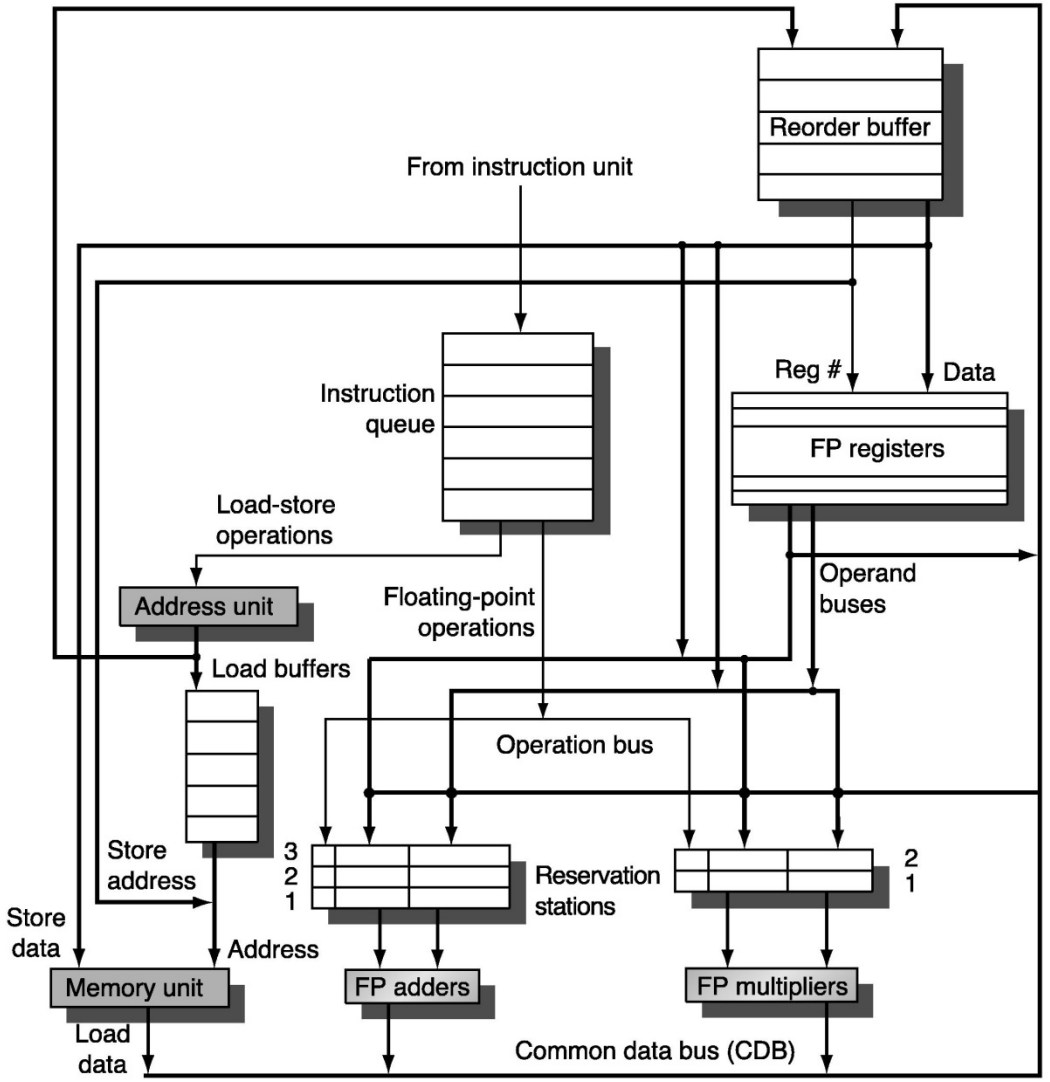
Hardware-Based Speculation

- Resulting Problem: If prediction wrong, must “undo” effects of wrongly executed instructions (also must deal with potential exceptions arising from wrongly executed instructions)
- Solution: Separate execution and write results from “commit”

Hardware-Based Speculation

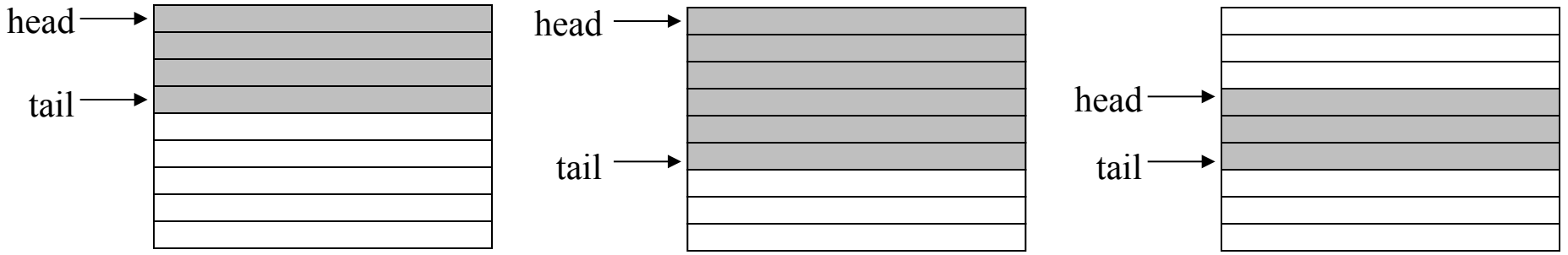
- Combines three key ideas
  - Dynamic branch prediction
    - Choose which instructions to execute
  - Speculation
    - Allow execution of instructions before control dependences are resolved
    - Ability to undo effects of incorrectly executed sequence
  - Dynamic scheduling
    - Schedule different combinations of basic blocks
- Used in most modern high-performance processors
  - PowerPC 603/604/G3/G4
  - MIPS R10000/12000
  - Intel Pentium II, III, 4
  - Alpha 21264
  - AMD K5, K6, Athlon
- Relies on extensions to Tomasulo's algorithm
  - Separate bypassing from actual instruction completion ("commit")

Extending Tomasulo's Algorithm to Handle Speculation and Precise Exceptions



Re-Order Buffers

- Re-Order buffer is FIFO queue
  - Slots allocated to instructions in order they are issued
- Facilitate "Commit" Operation
  - Don't write results until instruction commits (reaches head of queue)
  - Results of branches known
  - No prior instructions caused exception
- If Exception Occurs
  - Note it in re-order buffer but continue
  - Exceptions checked at time of commit
- Implemented with "head" and "tail" pointers (circular buffer)



Steps in Instruction Execution

- Issue
  - Get instruction from instruction queue (IF unit)
  - Issue if empty ROB slot and empty reservation station
  - Update entries to indicate buffers in use
  - ROB ID allocated to instruction sent to reservation stations
    - Will be used to tag results when placed on CDB
  - Stall if no available reservation station or no empty ROB slot
  - Sometimes called “dispatch”
- Execute
- Write Result
- Commit

Steps in Instruction Execution

- Issue
- Execute
  - If one or more source operands not available, monitor CDB for it
  - Effectively handles RAW hazards
  - When both operands available, execute instruction
  - May require multiple cycles
  - Sometimes called “issue”
- Write Result
- Commit

Steps in Instruction Execution

- Issue
- Execute
- Write Result
  - When result available, write it to CDB with tag of ROB entry
  - Written from CDB to ROB and any waiting reservation stations
  - Handle store instructions: write Value field of CDB if value available
  - Mark reservation station as available
- Commit

Steps in Instruction Execution

- Commit
  - Depends upon type of instruction
    - Branch with incorrect prediction
    - Store
    - Any other instruction (normal commit)
  - Normal commit
    - Instruction reaches head of ROB and result present in buffer
    - Update register with result
    - Remove instruction from ROB (advance head pointer)
  - Commit Store instruction
    - Same as above but write to memory
  - Commit incorrect branch
    - Indicates speculation was wrong
    - Flush ROB
    - Restart execution at correct branch successor
  - Sometimes called “completion” or “graduation”

Reorder Buffer

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	

Reorder buffer

Entry	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 × 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

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

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 destination, rs and rt are the sources, r is the reservation station allocated, b is the assigned ROB entry, and h is the head entry of the ROB. RS is the reservation station data structure. The value returned by a reservation station is called the result. RegisterStat is the register data structure, Regs represents the actual registers, and ROB is the reorder buffer data structure.

Interactions between instructions in fetch packet

Action or bookkeeping	Comments

`if (RegisterStat[rs1].Busy)/*in-flight instr. writes rs*/`
`  {h ← RegisterStat[rs1].Reorder;`
`  if (ROB[h].Ready)/* Instr completed already */`
`  {RS[r1].Vj ← ROB[h].Value; RS[r1].Qj ← 0;}`
`  else {RS[r1].Qj ← h;} /* wait for instruction */`
`} else {RS[r1].Vj ← Regs[rs]; RS[r1].Qj ← 0;};`
`RS[r1].Busy ← yes; RS[r1].Dest ← b1;`
`ROB[b1].Instruction ← Load; ROB[b1].Dest ← rd1;`
`ROB[b1].Ready ← no;`
`RS[r].A ← imm1; RegisterStat[rt1].Reorder ← b1;`
`RegisterStat[rt1].Busy ← yes; ROB[b1].Dest ← rt1;`
	Updating the reservation tables for the load instruction, which has a single source operand. Because this is the first instruction in this issue bundle, it looks no different than what would normally happen for a load.

`RS[r2].Qj ← b1;} /* wait for load instruction */`	Since we know that the first operand of the FP operation is from the load, this step simply updates the reservation station to point to the load. Notice that the dependence must be analyzed on the fly and the ROB entries must be allocated during this issue step so that the reservation tables can be correctly updated.

`if (RegisterStat[rt2].Busy) /*in-flight instr writes rt*/`
`  {h ← RegisterStat[rt2].Reorder;`
`  if (ROB[h].Ready)/* Instr completed already */`
`  {RS[r2].Vk ← ROB[h].Value; RS[r2].Qk ← 0;}`
`  else {RS[r2].Qk ← h;} /* wait for instruction */`
`} else {RS[r2].Vk ← Regs[rt2]; RS[r2].Qk ← 0;};`
`RegisterStat[rd2].Reorder ← b2;`
`RegisterStat[rd2].Busy ← yes;`
`ROB[b2].Dest ← rd2;`
	Since we assumed that the second operand of the FP instruction was from a prior issue bundle, this step looks like it would in the single-issue case. Of course, if this instruction was dependent on something in the same issue bundle the tables would need to be updated using the assigned reservation buffer.

`RS[r2].Busy ← yes; RS[r2].Dest ← b2;`
`ROB[b2].Instruction ← FP operation; ROB[b2].Dest ← rd2;`
`ROB[b2].Ready ← no;`
	This section simply updates the tables for the FP operation, and is independent of the load. Of course, if further instructions in this issue bundle depended on the FP operation (as could happen with a four-issue superscalar), the updates to the reservation tables for those instructions would be effected by this instruction.

Register Renaming

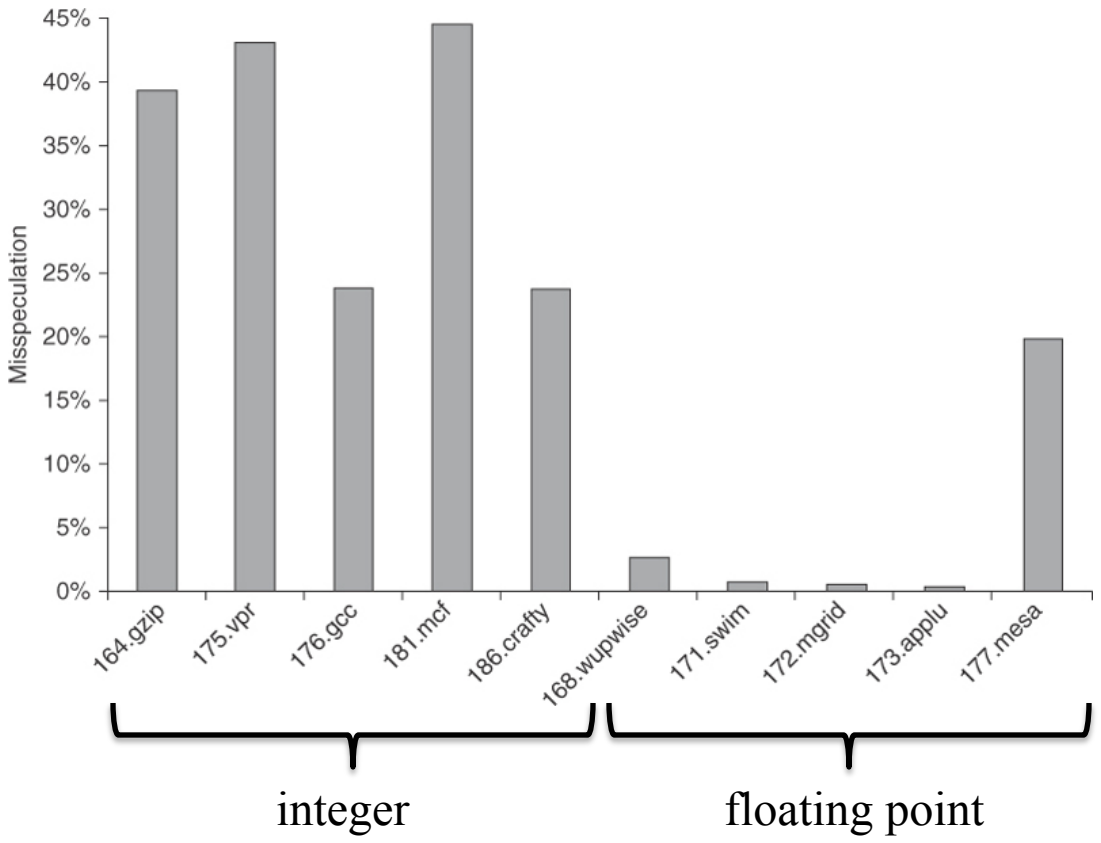
- Alternative to Reorder Buffer
  - RAT: Register Alias Table
  - More physical registers than architectural registers
  - Maps architectural registers to physical registers

Cost of speculation

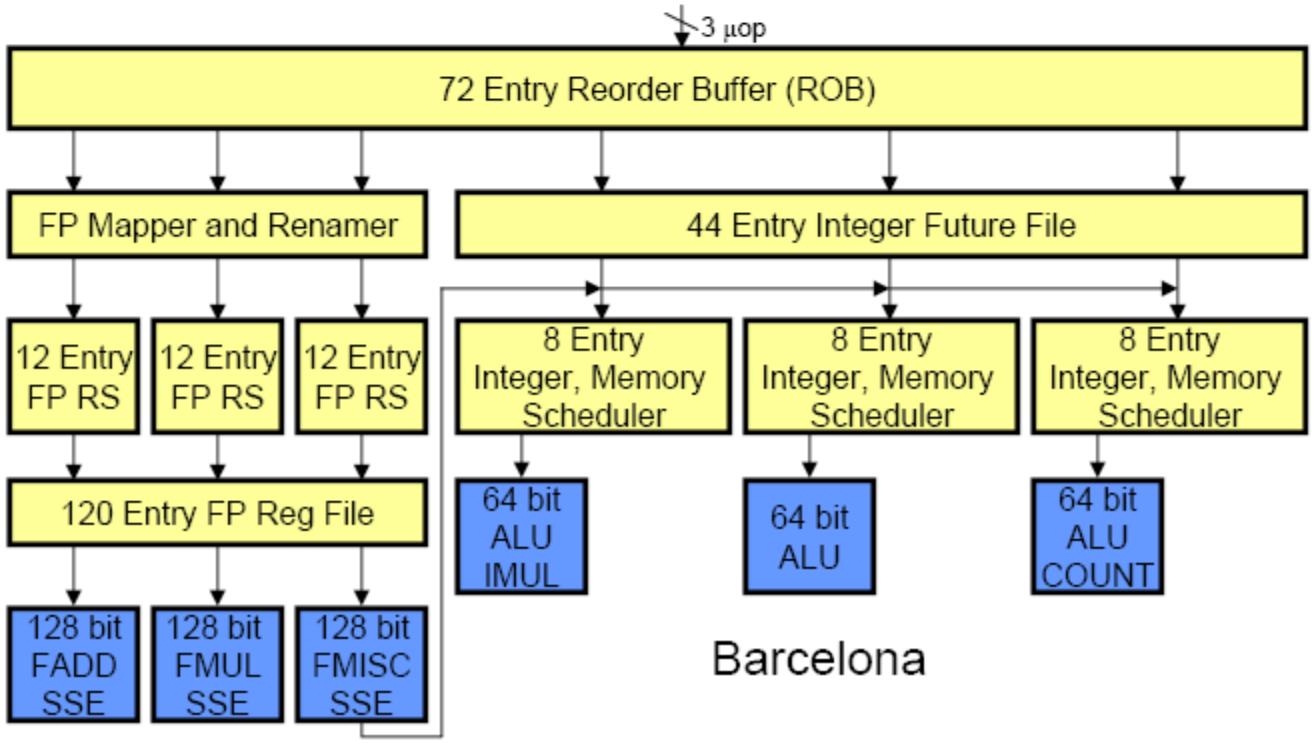
- Takes time and energy
- Time to recover from incorrect speculation
- Die area and complexity to implement
- Speculation may cause exceptional events
  - Cache line miss (initiate DRAM transaction)
  - TLB miss
- Some processors only allow "low-cost" exceptional events
  - First level cache misses, but not second (no DRAM access)

Cost of speculation

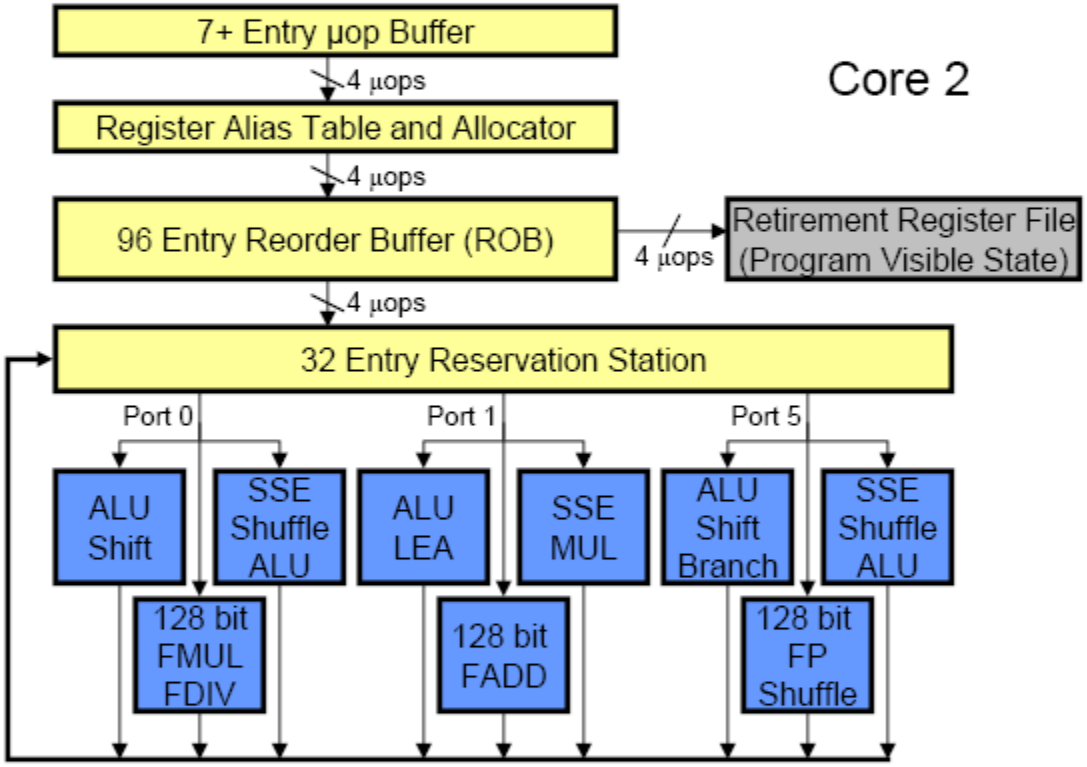
- Fraction of instructions executed as result of mis-speculation



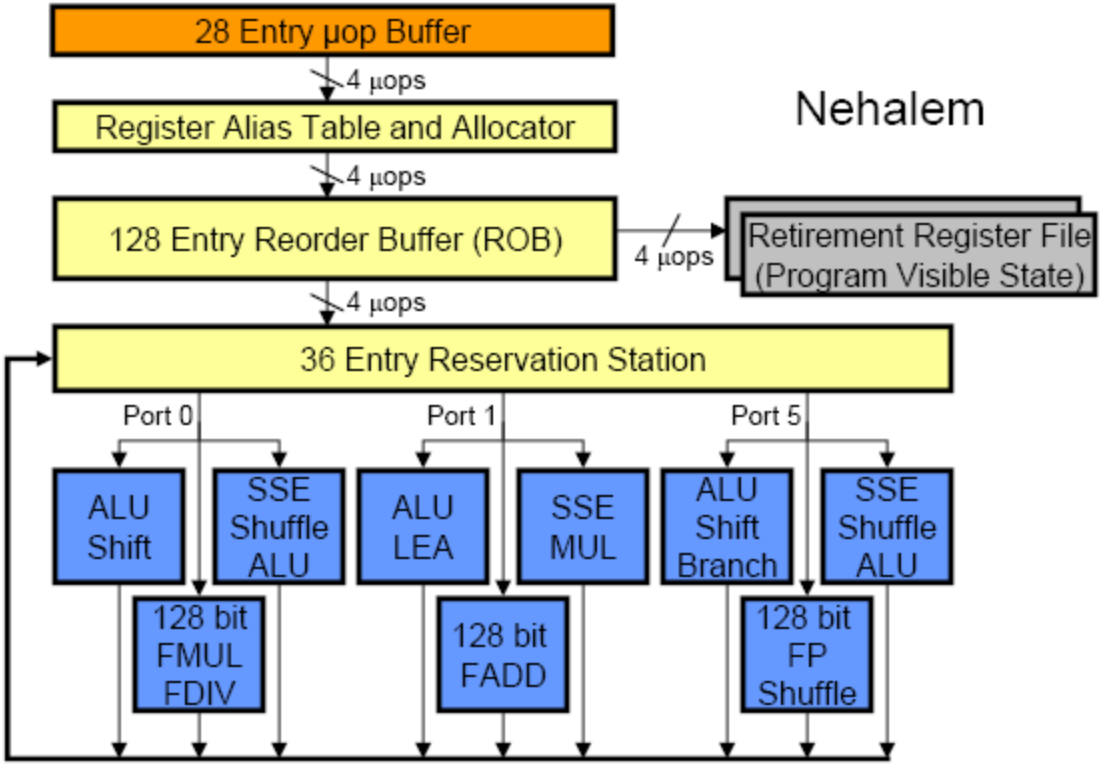
Some Examples



Some Examples



Some Examples



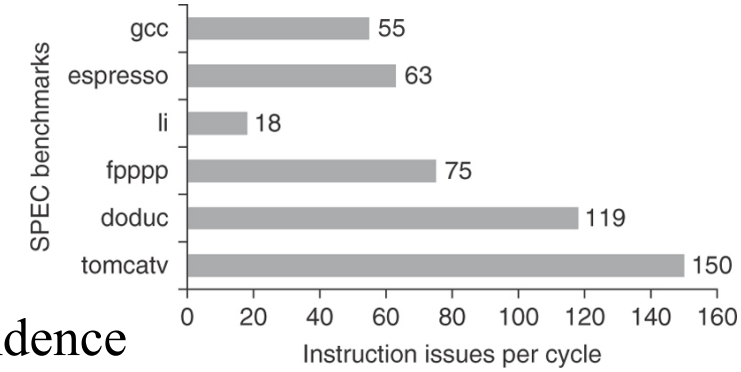
Terminology Review

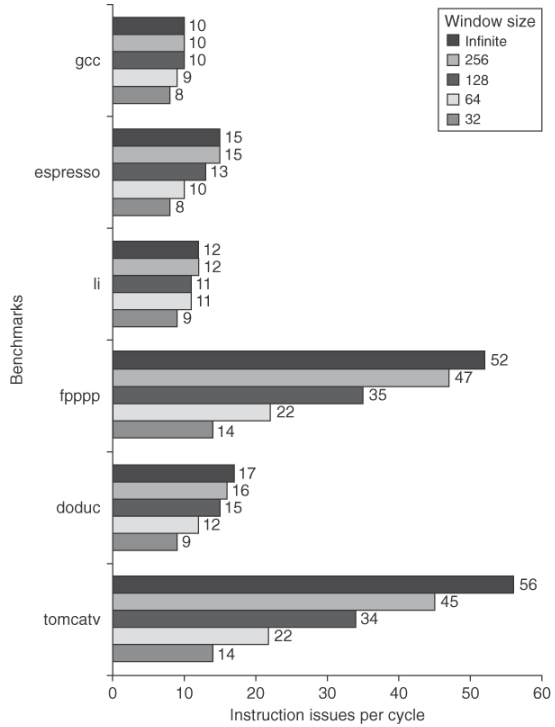
- Issue, Dispatch
  - Removing instruction from fetch queue or assigning a reservation station
- Completion
  - Usually when instruction completes and result is available (e.g. in ROB) but instruction has not committed (sometimes used synonymously with commit).
- Commit, Graduate, Retire
  - Instruction complete and branch outcome known so result is written to destination (e.g. memory or register file)

Limits to ILP

- Studies based on “ideal” processor
  - All predictions correct, unlimited registers
  - Perfect caches
  - Effectively no control, WAW, WAR dependence
  - Only RAW
  - One cycle latency for dependence
  - Maximal parallelism exploited
- Realizable processors
  - Up to 64 instruction issues/clock, no restrictions
  - Tournament predictor 1K entries
  - 16-entry return predictor
  - Register renaming (additional 64 integer, 64 FP)
  - Pipeline latency of one cycle

  - Conclusions
    - Integer programs not constrained by window size
    - Instead: branch prediction, registers, less inherent ILP

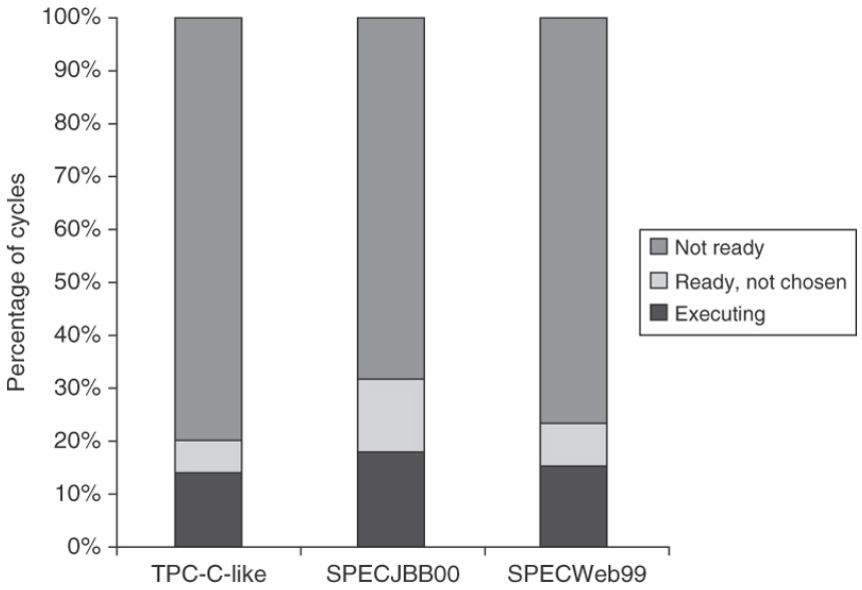


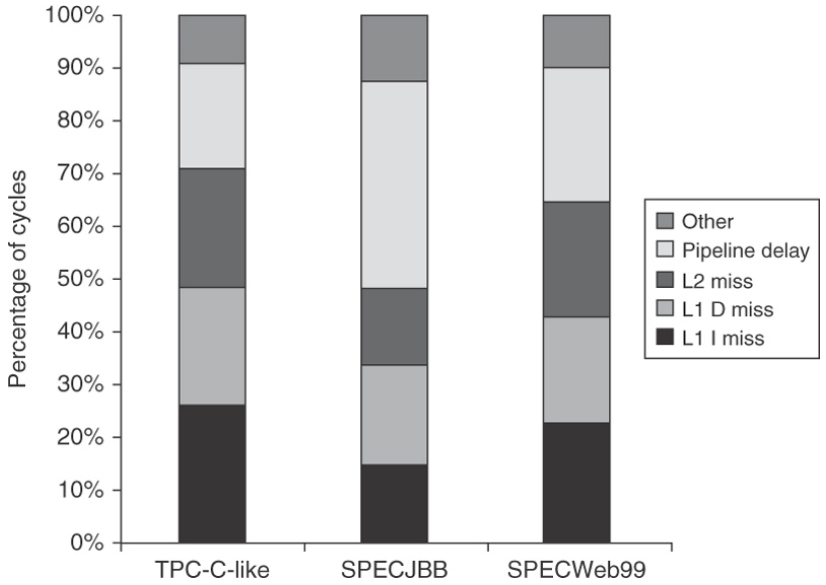


Thread-Level Parallelism

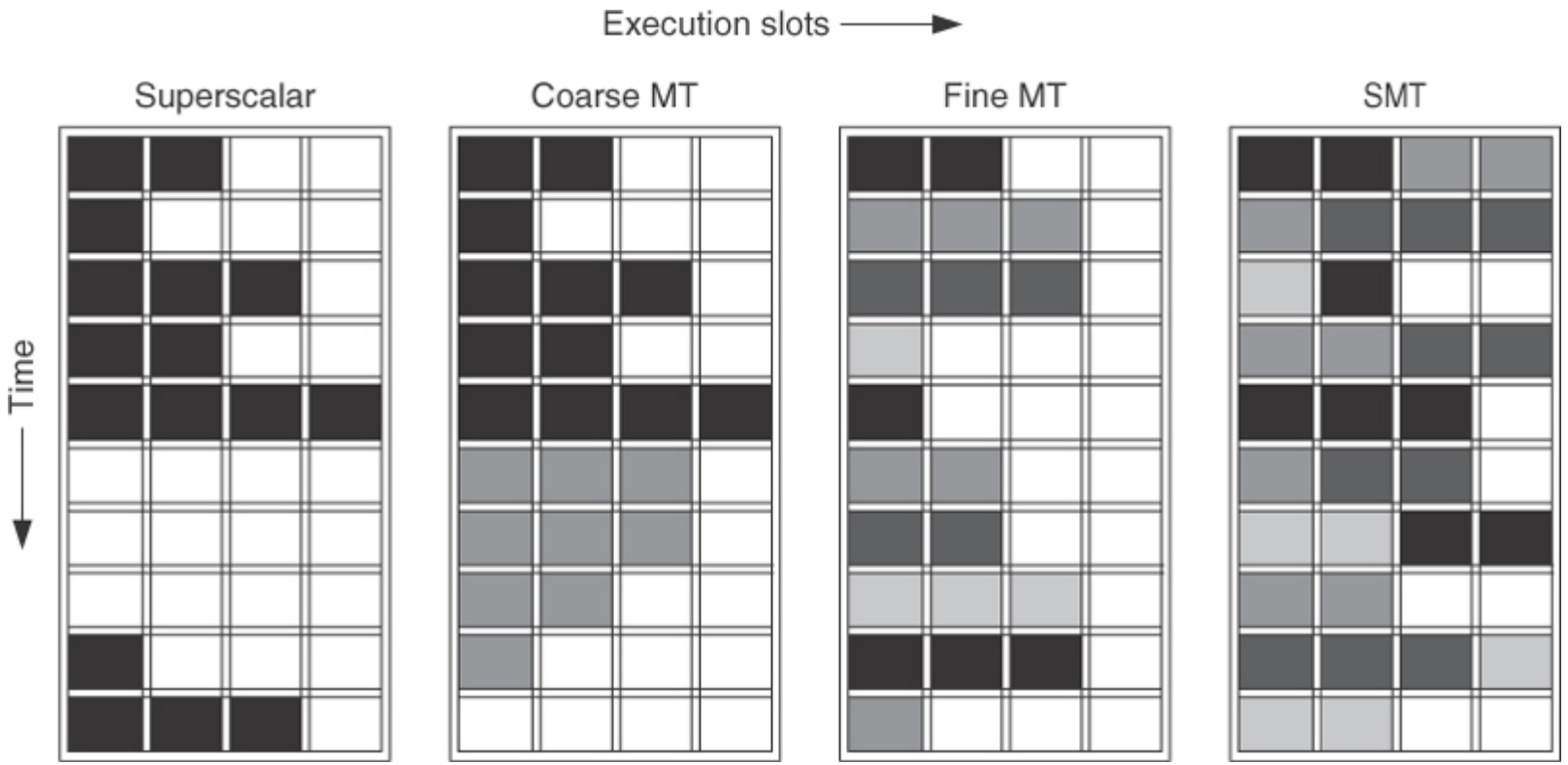
- Threads
  - Smallest unit of processing schedulable by an operating system
  - Similar to process
    - Processes typically independent, threads are subset of a process
    - Threads typically share process state, memory, other resources
    - Processes typically have own address spaces; threads share
    - Processes typically interact only through OS-provided IPC
  - Servers: multiple "threads" in common code
- Idea: not enough ILP in one thread: exploit parallelism across threads
- Cost of context switching: saving/restoring state when scheduling
  - Relatively high for process (page table, TLB, PC, processor status, general purpose registers)
  - Relatively low for thread (PC, processor status, general purpose registers)
- Sun T1 multicore processor
  - Introduced in 2005 for servers
  - Relies on thread level parallelism instead of ILP
    - 8 cores, each supporting 4 threads
    - Only single-issue desktop or server microprocessor introduced in > 5 years?
    - 6-stage, single-issue pipeline (stage for thread switch)
    - Processor idle only when all four threads are stalled
    - Loads and branches incur 3-cycle latency -- masked only by other threads
    - One set of FP units shared by all 8 cores

Sun Niagara


100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
Percentage of cycles
TPC-C-like
SPECJBB00
SPECWeb99
Not ready
Ready, not chosen
Executing


100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
Percentage of cycles
TPC-C-like
SPECJBB
SPECWeb99
Other
Pipeline delay
L2 miss
L1 D miss
L1 I miss

Superscalar and Threads



Sun Niagara

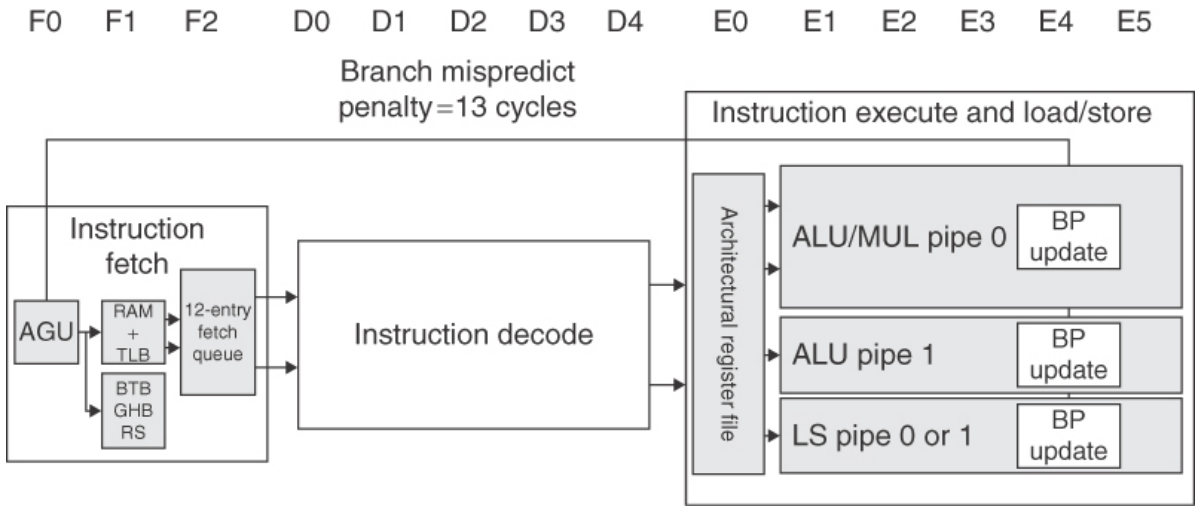
Intel Core i7
IBM Power7

ARM Cortex-A8

- Basis for Apple's A9 (iPad, iPhone 3GS/4), Motorola Droid
  - Dual-issue, statically scheduled superscalar
  - Dynamic branch predictor
  - 512 entry, 2-way set associative branch target buffer (BTB)
  - 4K entry global history buffer (GHB) (2-bit saturating counters)
    - Indexed by 10-bit global history register (GHR) and least two significant bits of PC
    - Organized as 256 x 32 bits. Upper 8 bits of GHR select 32 bits (16 two-bit predictors)
    - Low order 2 bits of GHR and 2 bits of PC used to select predictor
    - GHB only referenced when GHR changes to save power
  - 8 entry return stack

ARM Cortex-A8

- Basis for Apple's A9 (iPad, iPhone 3GS/4), Motorola Droid
  - 13-stage pipeline (ARM doesn't count F0)
    - 3 cycle fetch, 4 cycle decode, 5 cycle integer pipeline
  - 13-cycle mis-predict penalty
  - Can issue 0, 1, or 2 instructions per cycle (if single issue, always to pipe 0)
  - Checks for interactions between pair of instructions in fetch packet
    - Permits
      - one load/store and one ALU
      - two ALU
      - older multiply with younger load/store or ALU
      - only one instruction can change PC (e.g. branch or load/ALU with PC as destination register)



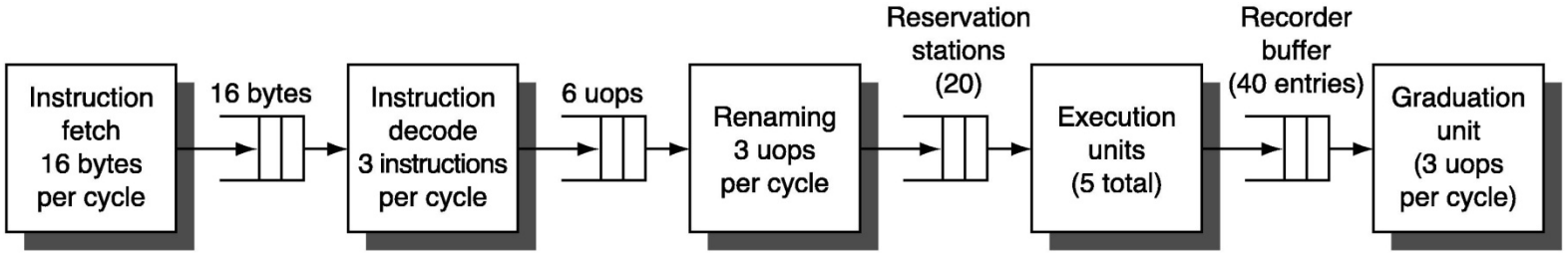
Intel P6 Microarchitecture

- P6 is basis for Pentium Pro, Pentium II, Pentium III
- Netburst microarchitecture is basis for Pentium 4
- Why "microarchitecture"?
  - Same ISA, Different Microarchitecture
    - e.g. pipeline, caches, functional units

Processor	First ship date	Clock rate range	L1 cache	L2 cache
Pentium Pro	1995	100–200 MHz	8 KB instr. + 8 KB data	256 KB–1024 KB
Pentium II	1998	233–450 MHz	16 KB instr. + 16 KB data	256 KB–512 KB
Pentium II Xeon	1999	400–450 MHz	16 KB instr. + 16 KB data	512 KB–2 MB
Celeron	1999	500–900 MHz	16 KB instr. + 16 KB data	128 KB
Pentium III	1999	450–1100 MHz	16 KB instr. + 16 KB data	256 KB–512 KB
Pentium III Xeon	2000	700–900 MHz	16 KB instr. + 16 KB data	1 MB–2 MB

Intel P6 Microarchitecture

- Dynamically scheduled
  - Translates each IA-32 instruction into series of micro-operations (μops) which are then pipelined
    - Translates CISC binary to RISC-like instructions
    - Retains binary compatibility
  - Up to 3 IA-32 instructions fetched, decoded, translated into μops every cycle
    - At most 1 "complex" instruction and 0-3 "simple" instructions
  - Why? Variable length instructions!
    - Instruction cache "back annotated" with instruction delimiters



CISC Instructions to μOps Example

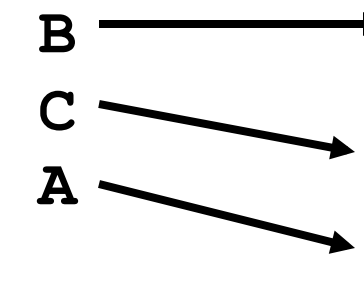
```
A = B + C
```

- CISC

```
Load  R1, B
Add   R1, C
Store R1, A
```

- μOps (RISC)

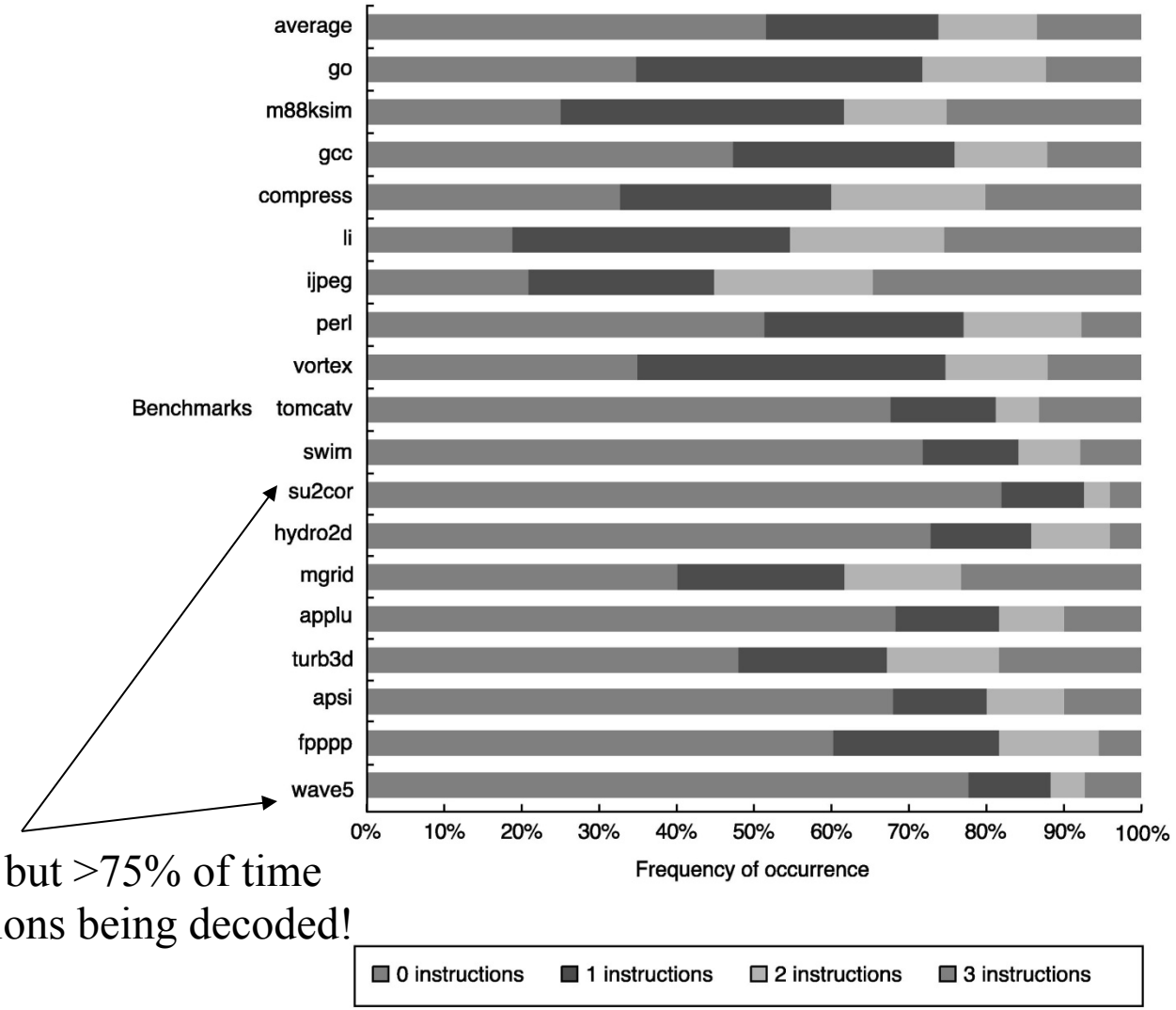
```
Load  R1, B
Load  R2, C
Add   R1,R1,R2
Store R1, A
```



Intel P6 Microarchitecture

Instruction name	Pipeline stages	Repeat rate
Integer ALU	1	1
Integer load	3	1
Integer multiply	4	1
FP add	3	1
FP multiply	5	2
FP divide (64-bit)	32	32

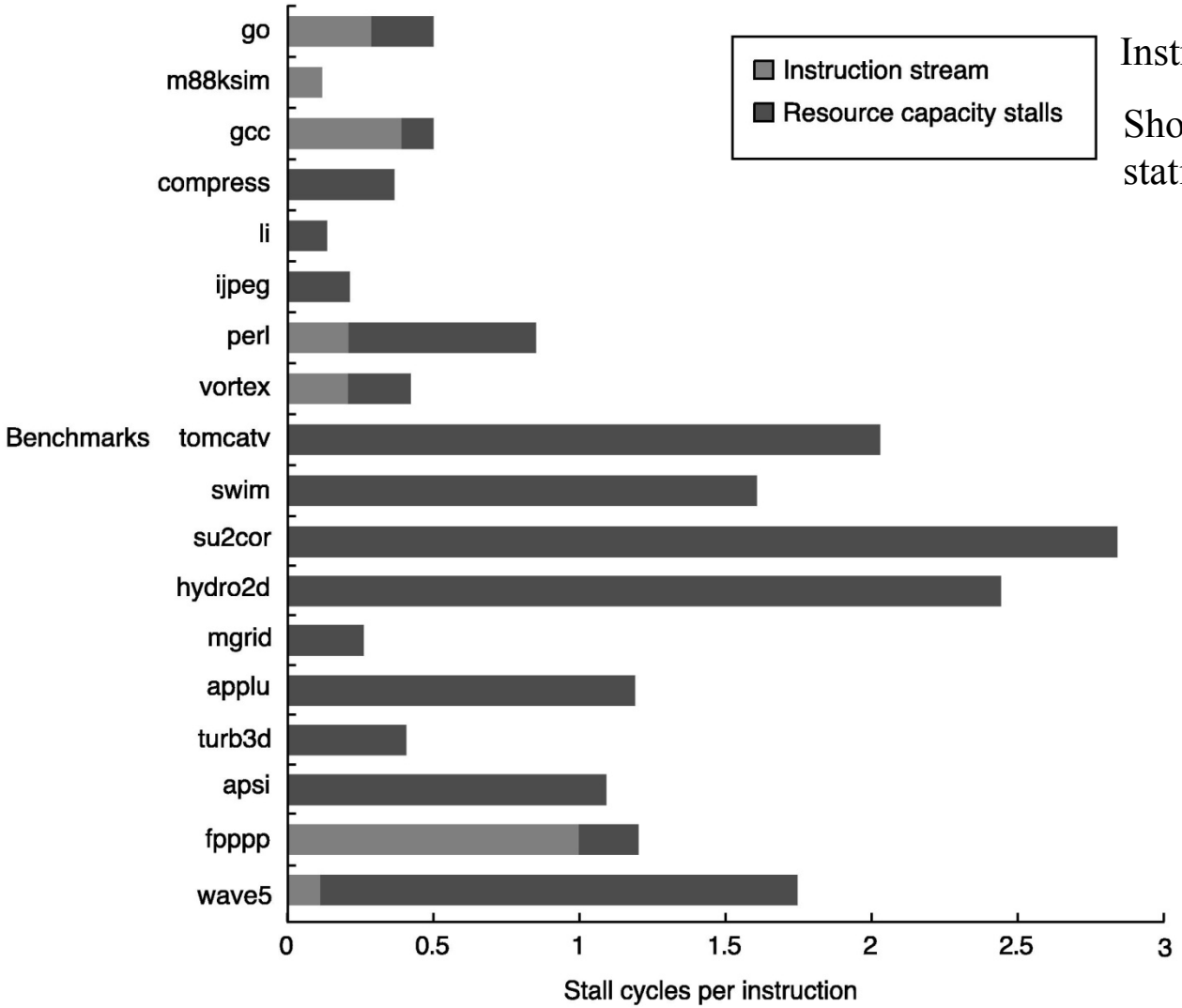
Evaluation of P6 Performance
Instructions Decoded per Clock



Avg = 0.87

3 decoders but >75% of time
no instructions being decoded!

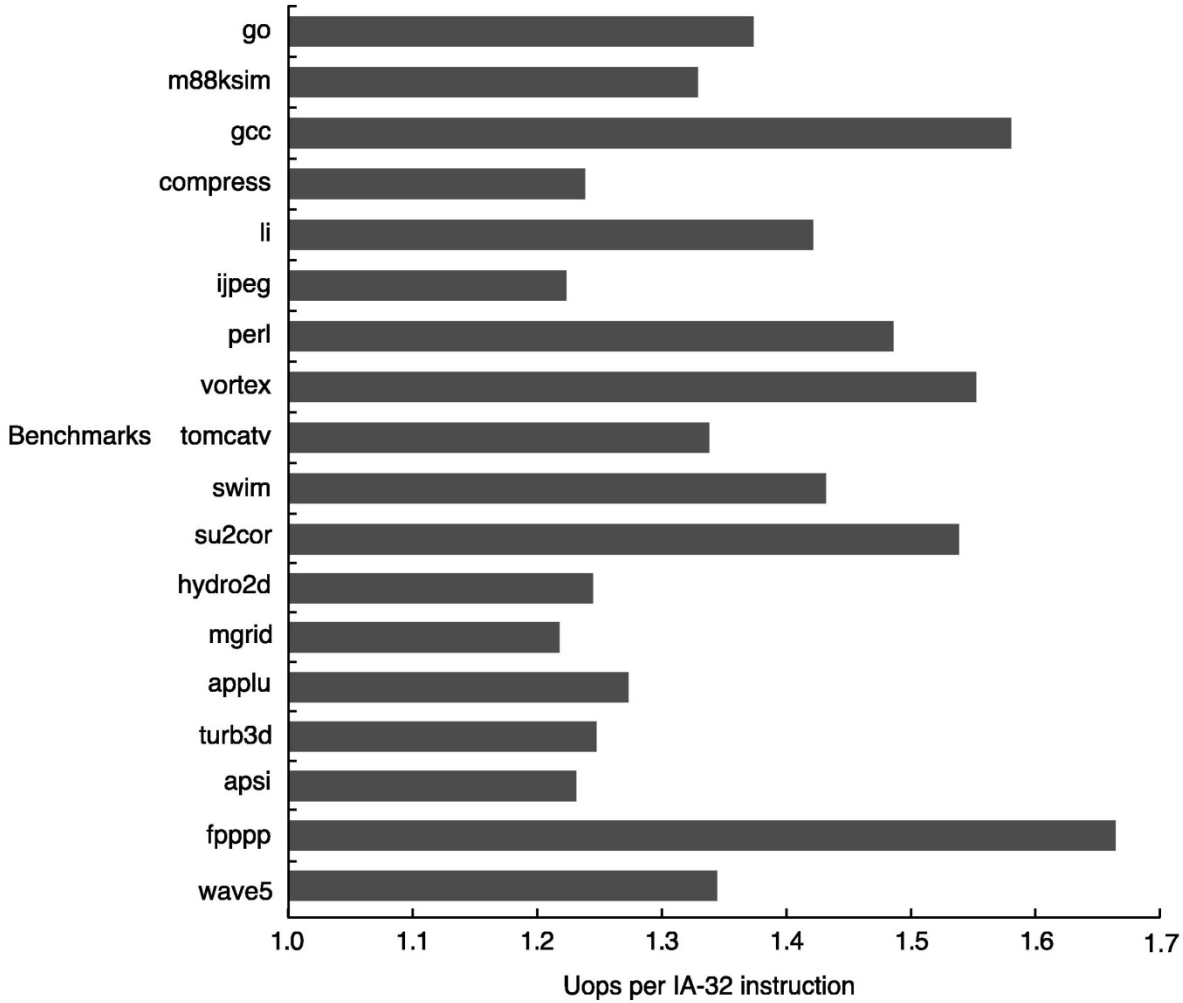
Evaluation of P6 Performance



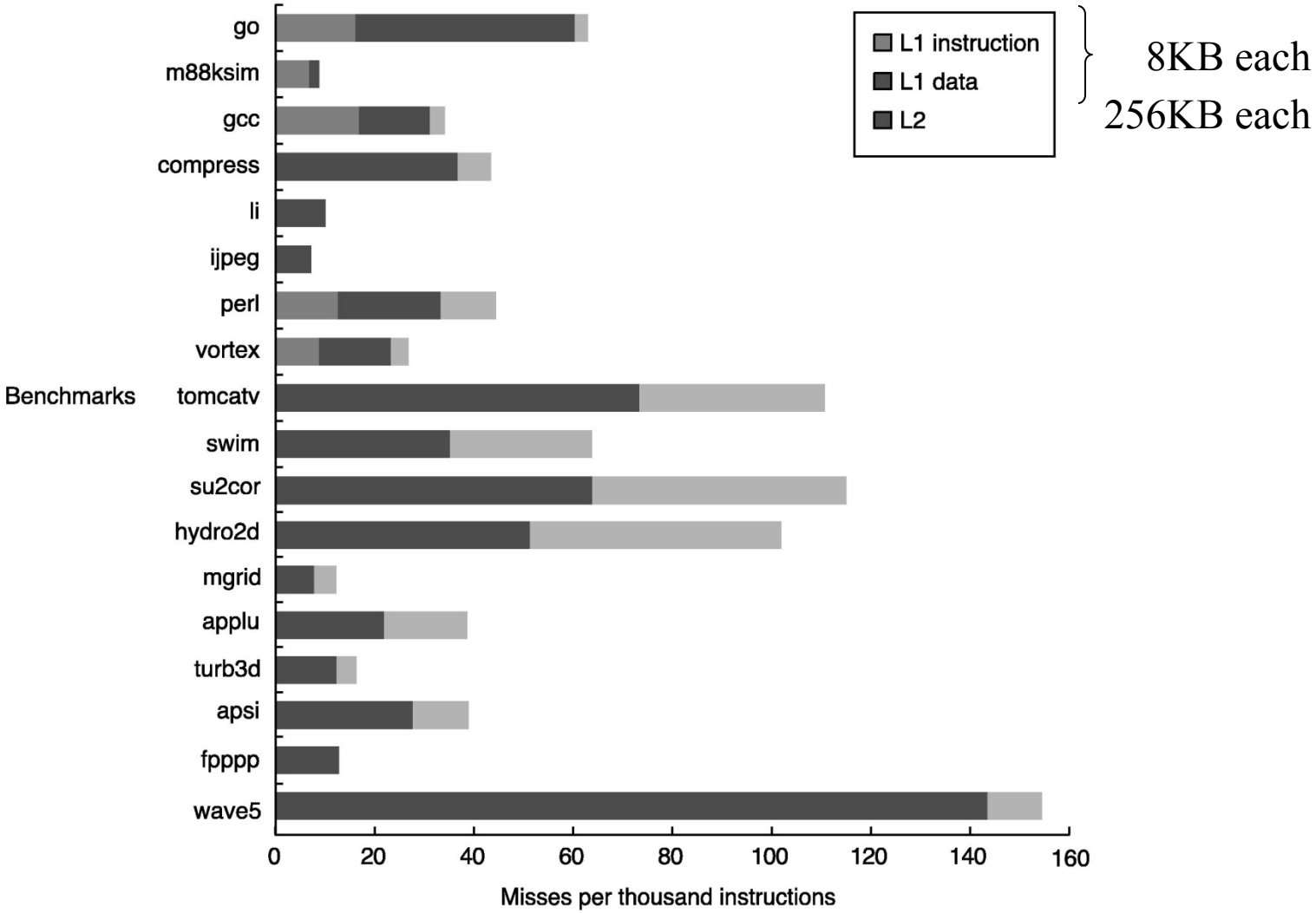
Instruction cache misses

Shortage of reservation stations or ROB entries

Evaluation of P6 Performance



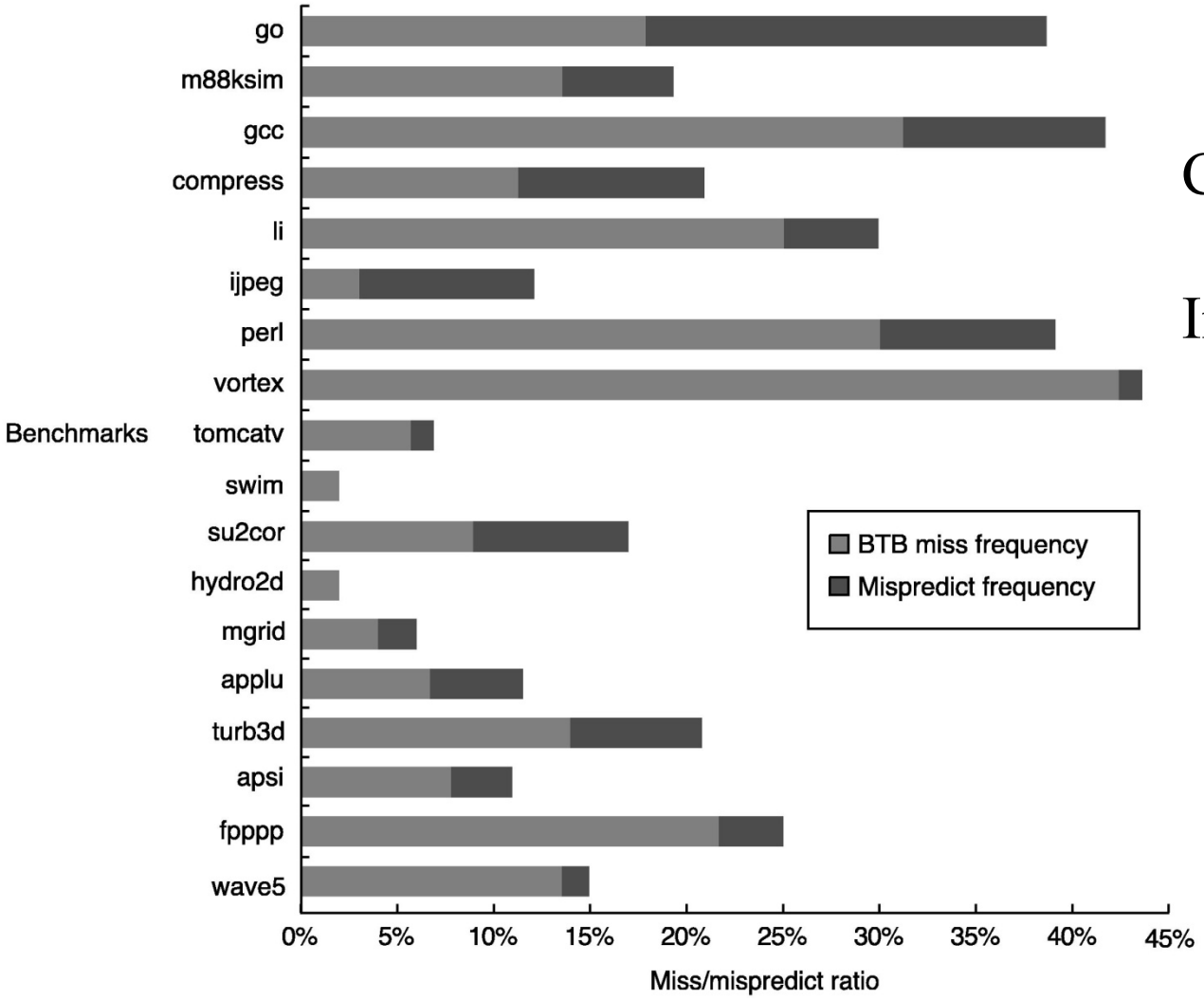
Evaluation of P6 Performance



Evaluation of P6 Performance

- Uses 512-entry branch target buffer (BTB)
- If BTB misses then uses static prediction
  - Backward branches
    - Assumed taken (why?)
    - One cycle penalty (if correctly predicted)
  - Forward branches
    - Assumed not-taken
    - No penalty (if correctly predicted) (why?)
- Mispredictions
  - Direct penalty 10-15 cycles (fetch out of order)
  - Indirect penalty (overhead of speculated instructions unnecessarily executed)

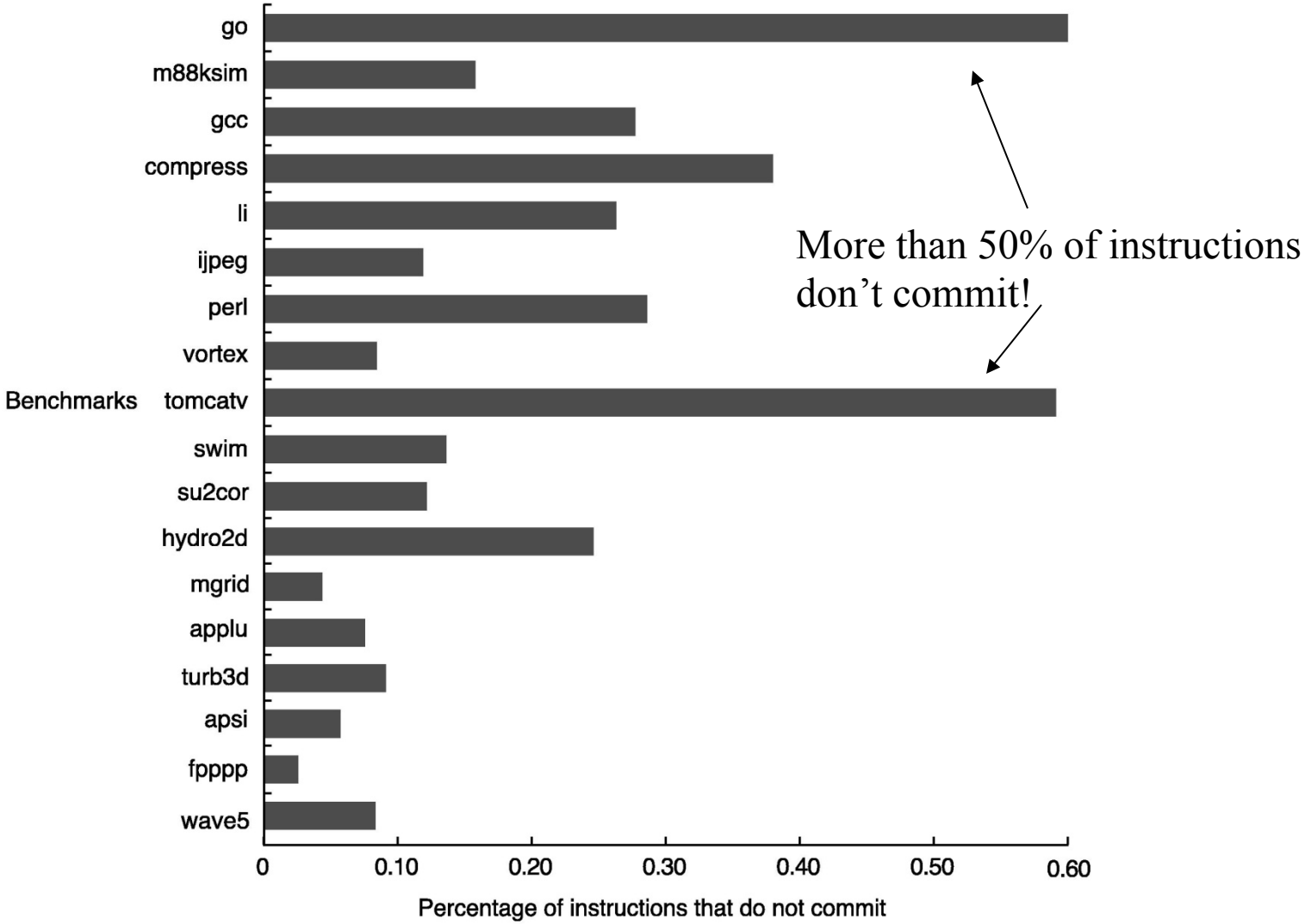
Evaluation of P6 Performance



Conclusion?

Increase size of BTB?

Evaluation of P6 Performance



Pentium III vs. Pentium 4

- NetBurst and P6 Similarities
  - Up to 3 IA-32 instructions fetched per cycle
  - Decoded to μops
  - Out-of-order execution with up to 3 μops per cycle
- NetBurst Differences
  - Deeper pipeline
    - P6: ~10 cycles for integer add from fetch to results available
    - NetBurst: ~20 cycles (2 for driving results across chip)
  - Explicit register renaming
    - P6: Reorder Buffer (ROB)
    - NetBurst: Register Address Translation table (RAT)
  - Additional integer FUs
    - P6: 5
    - NetBurst: 7 (additional integer ALU, additional address calculation unit)

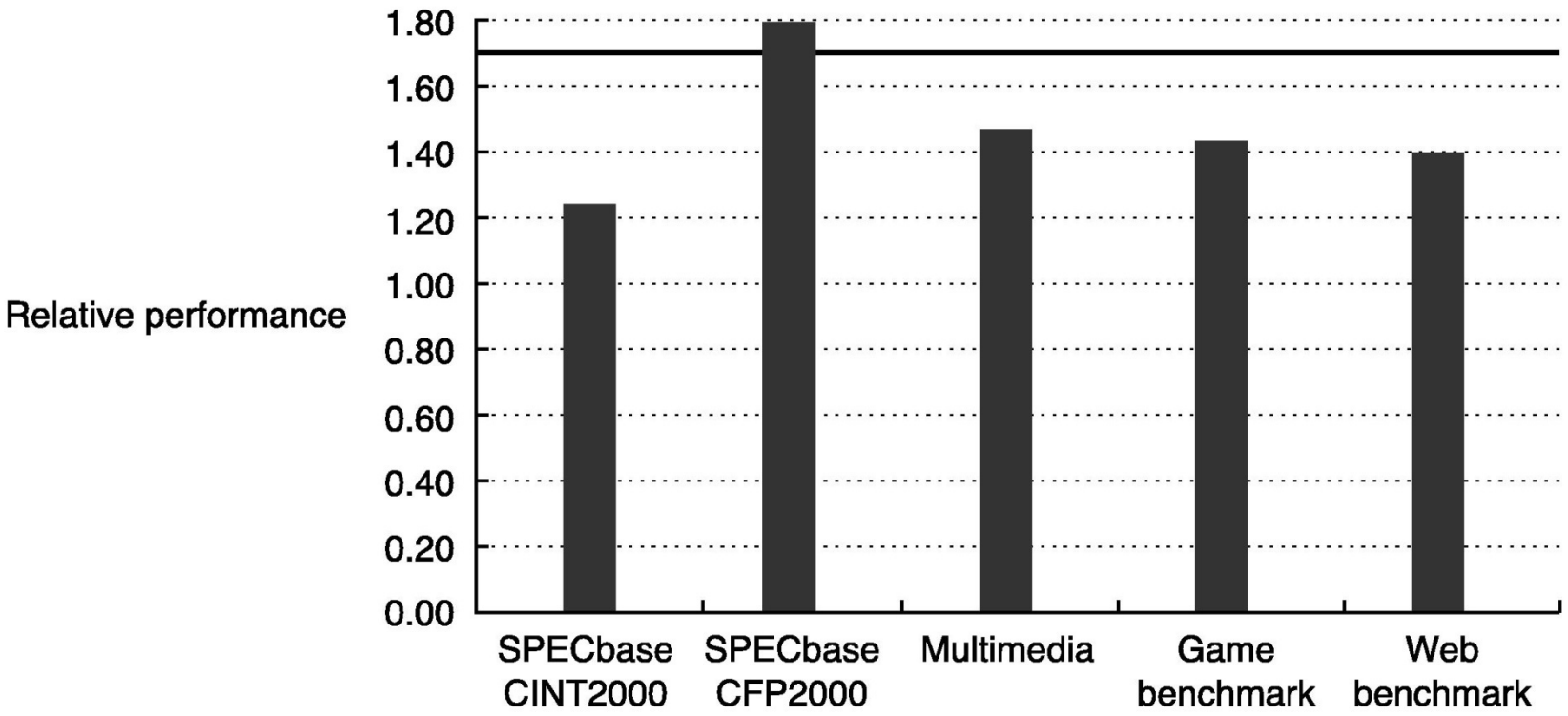
Pentium III vs. Pentium 4

- NetBurst Differences
  - Aggressive ALU and data cache
    - Reduce stalls in deep pipeline
  - Trace cache
    - P6: prefetch buffer and instruction cache
    - NetBurst: trace cache (holds decoded fixed length μops instead of instructions)
  - Larger BTB (8x)
  - Cache size
    - P6: 8K/16K L1 data
    - NetBurst: 8K L1 data, but higher bandwidth L2
  - SSE2 (“Streaming SIMD Extensions 2”) floating point
    - Two FP operations/instruction (SLIW – sorta long instruction word)
    - 128-bit SIMD (MMX)

Performance Comparison

- Recall Chapter 1…

Performance of 1.7GHz Pentium 4 relative to 1GHz Pentium III



Performance Comparison

- Integer benchmark performance improvement less than increase in clock speed (1.7x)
- Floating point performance improvement greater than increase in clock speed (SSE2)