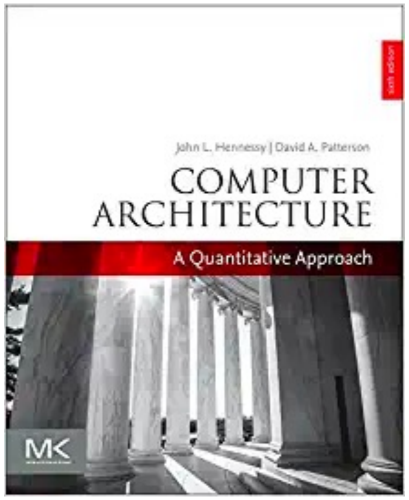


Chapter 3

Instruction-Level Parallelism and Its Exploitation

Part 2: Pipelining Scheduling and Compiler Optimizations

"Who's first?"
"America."
"Who's second?"
"Sir, there is no second."

-Dialog between two observers of the sailing race later named "The America's Cup" and run every few years -- the inspiration for John Cocke's naming of the IBM research processor as "America." This processor was the precursor to the RS/6000 series and the first superscalar microprocessor.

Acknowledgements

- Thanks to many sources for slide material
  - © 1990 Morgan Kaufmann Publishers, © 2001-present Elsevier
    Computer Architecture: A Quantitative Approach by J. Hennessy & D. Patterson
  - © 1994 Morgan Kaufmann Publishers, © 2001-present Elsevier
    Computer Organization and Design by D. Patterson & J. Hennessy
  - © 2002 K. Asinovic & Arvind, MIT
  - © 2002 J. Kubiatowicz, University of California at Berkeley
  - © 2006, © 2010 No Starch Press for Inside the Machine by J. Stokes
  - © 2007 W.-M. Hwu & D. Kirk, University of Illinois & NVIDIA
  - © 2007-2010 J. Owens, University of California at Davis
  - © 2010 CRC Press for Introduction to Concurrency in Programming Languages by M. Sottile, T. Mattson, and C. Rasmussen
  - © 2017, IBM POWER9 Processor Architecture by Sadasivam et al., IBM
  - © 2016, © 2019 POWER9 Processor User's Manual, IBM
  - © The OpenPOWER Foundation
  - © 2022, W. Feng, Virginia Tech

Compiler Techniques for Exposing ILP

- Pipeline Scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Separation 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

Pipeline Stalls

```
Loop:   L.D     F0,0(R1)        ; F0=array element
        stall
        ADD.D   F4,F0,F2        ; add scalar in F2
        stall
        stall
        S.D     F4,0(R1)        ; store result
        DADDUI  R1,R1,#-8       ; decrement pointer
        stall (assume integer load latency is 1)
        BNE     R1,R2,Loop
```

Instruction producing result	Instruction using result	Separation 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

Pipeline Scheduling

```
                                      Cycle Issued?
Loop:   L.D     F0,0(R1)              1
        stall                         2
        ADD.D   F4,F0,F2              3
        stall                         4
        stall                         5
        S.D     F4,0(R1)              6
        DADDUI  R1,R1,#-8             7
        stall   (assume integer load latency is 1)   8
        BNE     R1,R2,Loop            9
```

<u>Scheduled Code</u>

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

How long for 4 iterations? 8? 12? 16?

Instruction producing result	Instruction using result	Separation 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

Loop Unrolling (w/o Pipeline Scheduling)

- Loop Unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

Note: # of live registers vs. original loop

	Cycle Issued?	
Loop: L.D F0,0(R1)	1	
ADD.D F4,F0,F2	3	
S.D F4,0(R1)	6	;drop DADDUI & BNE
L.D F6,-8(R1)	7	
ADD.D F8,F6,F2	9	
S.D F8,-8(R1)	12	;drop DADDUI & BNE
L.D F10,-16(R1)	13	
ADD.D F12,F10,F2	15	
S.D F12,-16(R1)	18	;drop DADDUI & BNE
L.D F14,-24(R1)	19	
ADD.D F16,F14,F2	21	
S.D F16,-24(R1)	24	
DADDUI R1,R1,#-32	25	
BNE R1,R2,Loop	27	

		Separation
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

How long for 4 iterations? 8? 12? 16?

Loop Unrolling *WITH* Pipeline Scheduling

- Pipeline schedule the unrolled loop

Instruction	Cycle Issued?
Loop: L.D F0,0(R1)	1
L.D F6,-8(R1)	2
L.D F10,-16(R1)	3
L.D F14,-24(R1)	4
ADD.D F4,F0,F2	5
ADD.D F8,F6,F2	6
ADD.D F12,F10,F2	7
ADD.D F16,F14,F2	8
S.D F4,0(R1)	9
S.D F8,-8(R1)	10
DADDUI R1,R1,#-32	11
S.D F12,16(R1)	12
S.D F16,8(R1)	13
BNE R1,R2,Loop	14

Can you identify the *name dependences* and *data dependences?*

Instruction producing result	Instruction using result	Separation 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

How long for 4 iterations? 8? 12? 16?

Loop Unrolling with Unknown # of Iterations

- Unknown number of loop iterations?
  - Assume *n* = number of iterations
  - Goal: Loop unroll and make *k* copies of the loop body
  - Approach: Generate a pair of consecutive loops (instead of a single unrolled loop)
    - First executes *n mod k* times
    - Second executes *n* / *k* times

Algorithmic Summary

- Loop Unrolling and Pipeline Scheduling

  Key Requirement: Must understand how one instruction depends on another and how the instructions can be changed or reordered given dependences.

  – Determine that loop unrolling useful by identifying loop iterations as independent (except for loop maintenance code)
  – Use different registers to avoid unnecessary constraints that are forced by using same registers for different computations (e.g., name dependences)
  – Eliminate extra test and branch instructions and adjust loop termination and iteration code
  – Determine loads and stores in unrolled loop that can be interchanged by observing that loads and stores from different iterations are independent
  – Schedule the code, preserving any dependences needed to yield the same result as the original code.

Limitations of Loop Unrolling

- Less overhead "amortizable" with each unroll
  - Example: Generated sufficient parallelism among instructions that loop could be scheduled with *no* stall cycles.
- Code size limitations
  - Memory is cheap, so why is this a problem?
- Compiler limitations
  - What happens to hardware resource usage via aggressive unrolling and pipeline scheduling?

Branching Hurts Performance

- Due to the need to enforce control dependences through hazards and stalls (or “bubbles”).

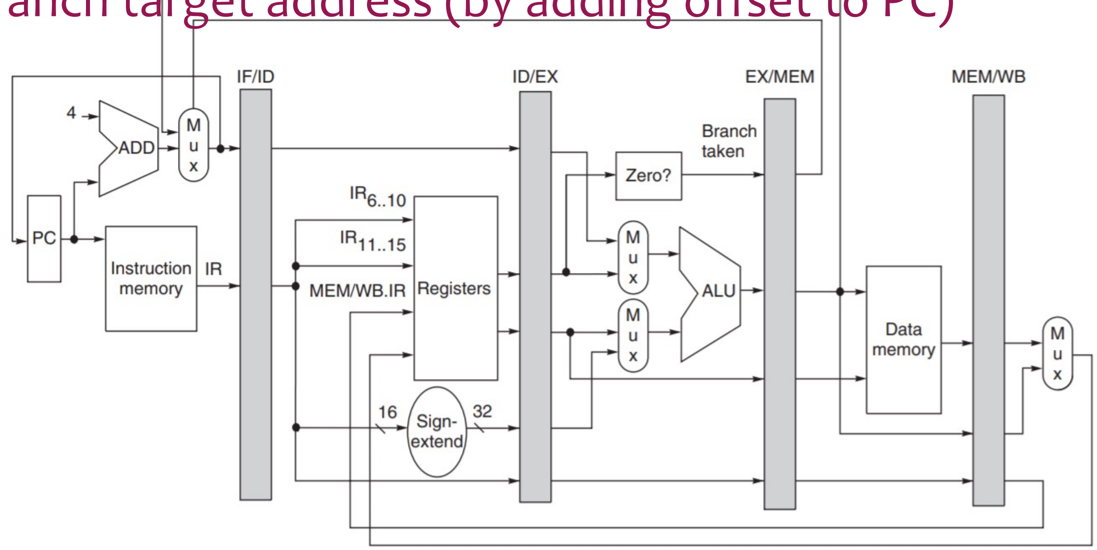
Methods to reduce performance loss due to branches:

1. Loop unrolling is one way to reduce # of branch hazards. See previous slides.
2. Predict how branches will behave.
   - As # of instructions in flight has increased, the importance of more accurate branch prediction grows.

Recall:

Five-Stage Pipelined Processor

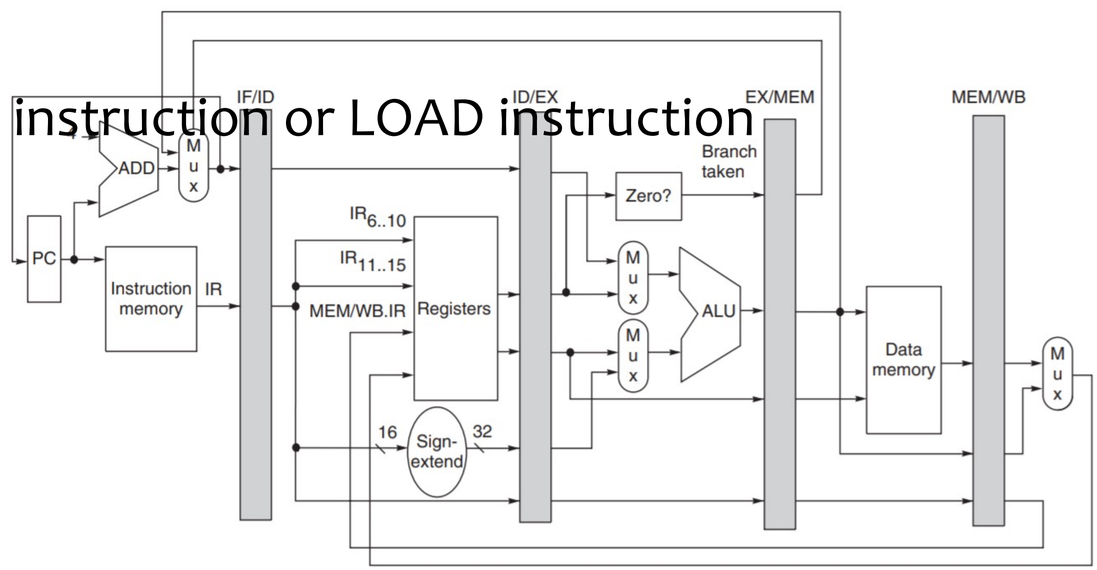
1. Instruction Fetch (IF)
   - Send PC to memory & fetch current instruction from memory
   - Update PC to next sequential PC (e.g., 4 for 32-bit architecture)
2. Instruction Decode (ID) / Register Fetch
   - Decode instruction *and* read source registers in parallel
     - Why is this possible? "Fixed-field decoding"
   - Do *EQUALITY* test on registers during read for possible branch
     - Sign-extend the offset field of instruction, if needed
     - Compute possible branch target address (by adding offset to PC)



Recall:

Five-Stage Pipelined Processor

3. Execute (EXE) / Effective Address
   - EXE on operands from previous cycle
     - Memory Reference
     - Register-Register ALU instruction
     - Register-Immediate ALU instruction
4. Memory Access (MEM)
   - LOAD: Memory read using effective address calculated
   - STORE: Memory write data from register read to effective addr
5. Write Back (WB)
   - Register-Register ALU instruction or LOAD instruction



Reducing the Impact of Branches

1. Baseline: *Freeze* or *flush* the pipeline
   - Hold or delete any instruction after the branch until branch destination known.

Untaken branch instr	IF	ID	EXE	MEM	WB				
Instr *i+1*		IF	ID	EXE	MEM	WB			
Instr *i+2*			IF	ID	EXE	MEM	WB		
Instr *i+3*				IF	ID	EXE	MEM	WB	
Instr *i+4*					IF	ID	EXE	MEM	WB

Taken branch instr	IF	ID	EXE	MEM	WB				
Instr *i+1*		IF	idle	idle	idle	idle			
Branch target			IF	ID	EXE	MEM	WB		
Branch target + 1				IF	ID	EXE	MEM	WB	
Branch target + 2					IF	ID	EXE	MEM	WB

2. Slightly better but more complex … treat every branch as *not taken.* Trick: Do *not* change processor state until branch outcome definitively known.

Reducing the Impact of Branches

3. Delayed Branch

Job of the compiler is to make "sequential successor" valid and useful

```
branch instr
"sequential successor" → branch delay slot
branch target if taken
```

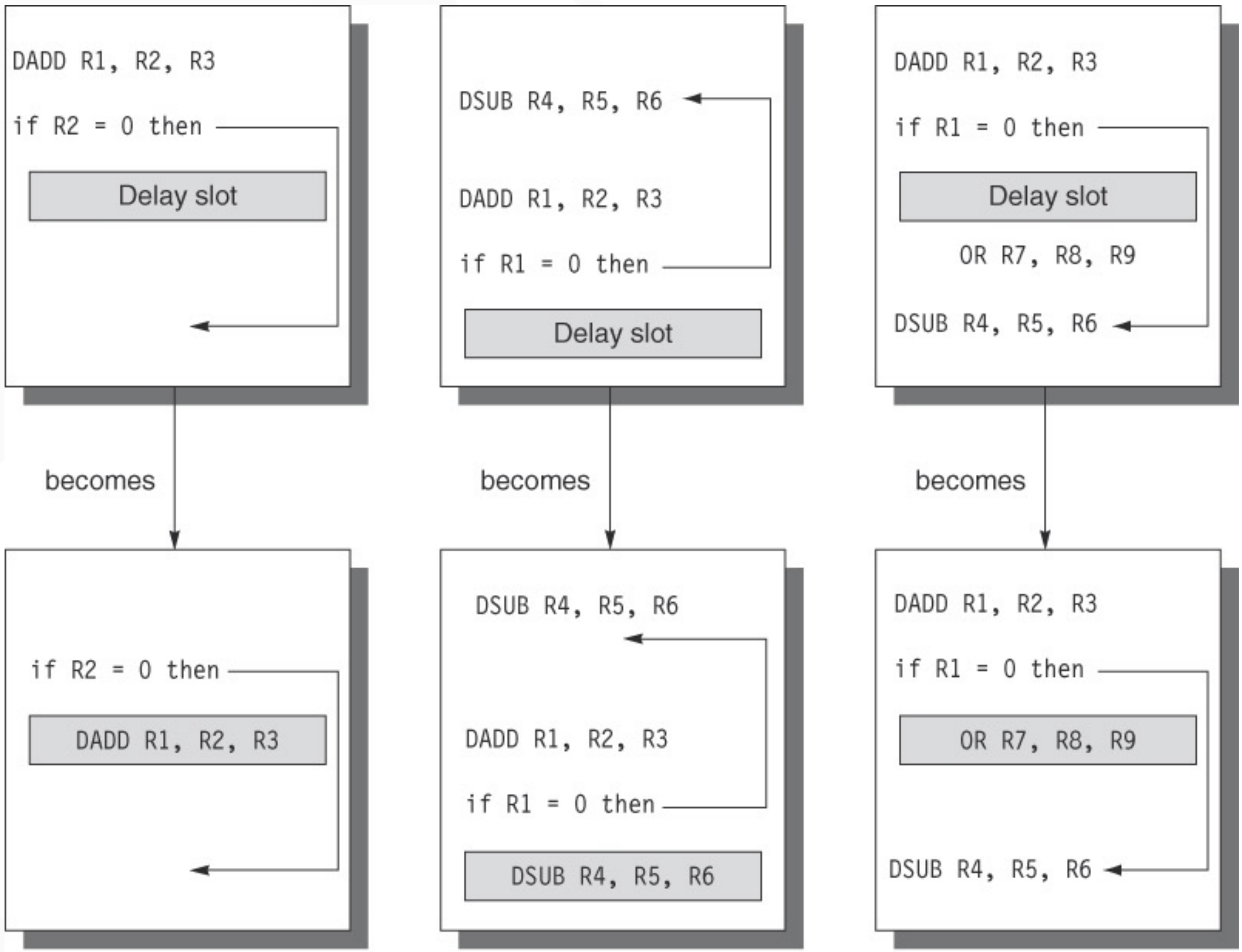
where "sequential successor" executes whether or not branch is taken

Untaken branch instr	IF	ID	EXE	MEM	WB				
Branch delay instr (i+1)		IF	ID	EXE	MEM	WB			
Instr *i+2*			IF	ID	EXE	MEM	WB		
Instr *i+3*				IF	ID	EXE	MEM	WB	
Instr *i+4*					IF	ID	EXE	MEM	WB

Taken branch instr	IF	ID	EXE	MEM	WB				
Branch delay instr (i+1)		IF	ID	EXE	MEM	WB			
Branch target			IF	ID	EXE	MEM	WB		
Branch target + 1				IF	ID	EXE	MEM	WB	
Branch target + 2					IF	ID	EXE	MEM	WB

The behavior of a delayed branch is the same whether branch is taken or not!

Scheduling Branch Delay Slot



(a) From before (b) From target (c) From fall-through

Performance of Branch Schemes

- Assuming ideal CPI of 1, then the effective pipeline speed up with branch penalties

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Pipeline stall cycles from branches

= Branch frequency x Branch penalty

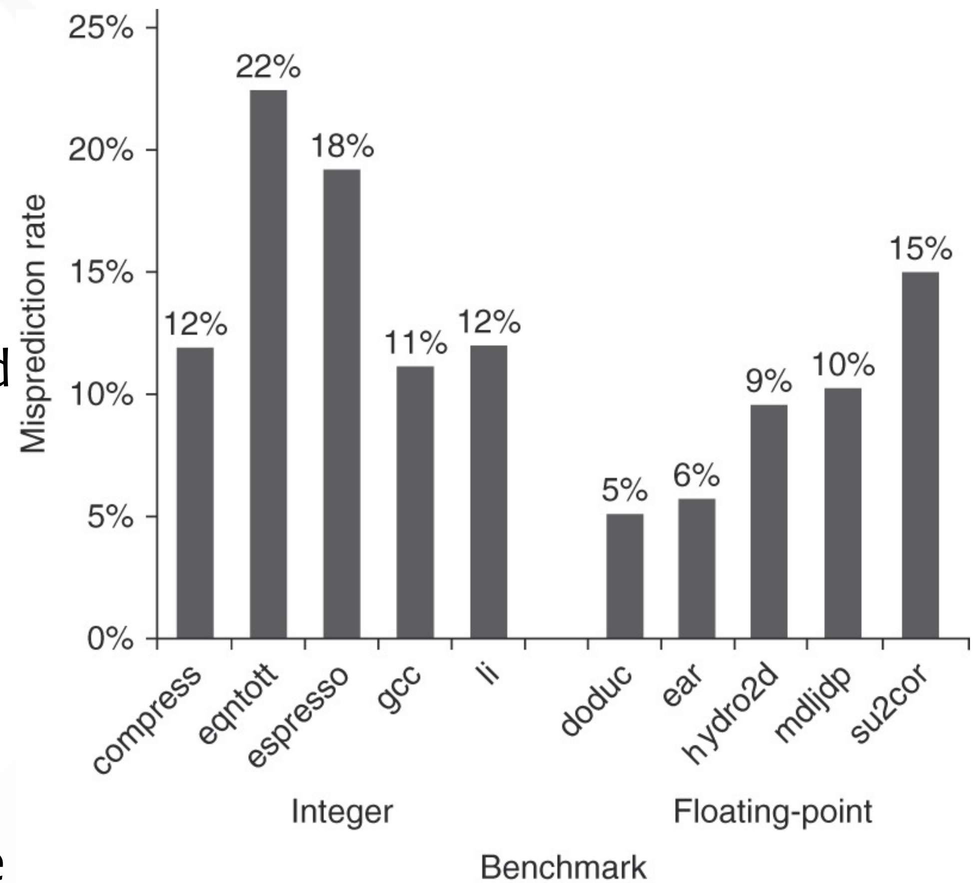
Advanced Techniques for Reducing the Impact of Branches

1. Static Branch Prediction
   - Observation: The behavior of branches is often biomodally distributed, i.e., an individual branch is often highly biased toward taken or untaken.
   - See next slide.

2. Dynamic Branch Prediction
   - Basic 2-bit Predictor
   - Correlating Predictor
   - Local Predictor
   - Tournament Predictor

Static Branch Prediction

- Profile the code
- Observation
  - Branch behavior often bimodally distributed, i.e., individual branch highly biased to taken or untaken
- Experimental Setup
  - Same input data used for runs and for collecting profile
- Rigged?
  - Changing the input so that the profile is for a different run leads to only a small change in accuracy



Success of Branch Prediction Using ***Static*** Branch Prediction

Dynamic Branch Prediction (1-bit)

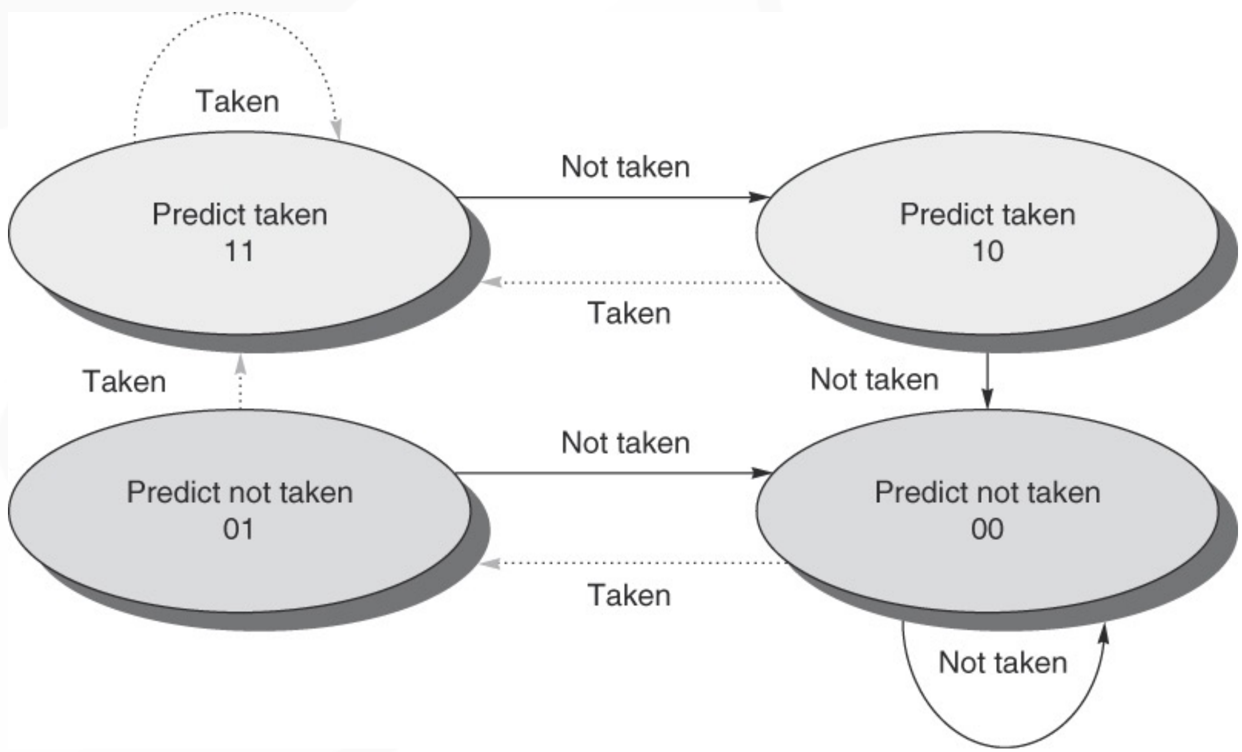
- *Branch-Prediction Buffer* or *Branch-History Table*
  - Small memory indexed by the lower portion of the address of the branch instruction.
    - Contains a bit for whether a branch was recently taken or not
    - Useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs
- Potential Issue?
  - *Yes:* Don't know if prediction is correct as it may have been put there by another branch that has the same low-order address bits.
  - *No:* Prediction is a hint that is assumed to be correct, and fetching begins in predicted direction. If hint is wrong, prediction bit is inverted and stored back.

Branch Prediction

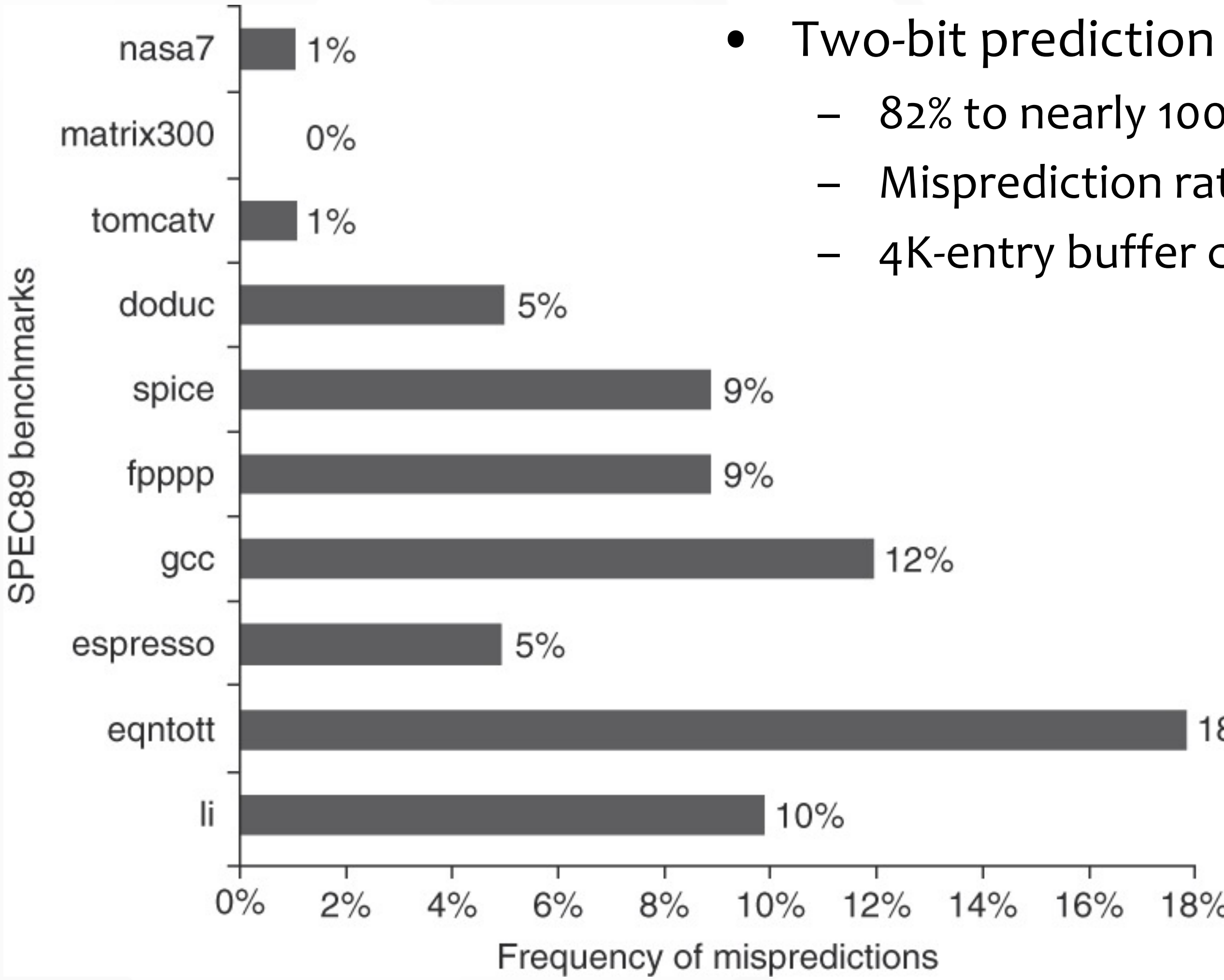
- Basic 2-bit Predictor
  - For *each* branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating Predictor
  - Multiple 2-bit predictors for *each* branch
  - One for each possible combination of outcomes of preceding *n* branches
- Local Predictor
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the *last n* occurrences of *this* branch
- Tournament Predictor
  - Combine correlating predictor with local predictor

Branch Prediction

- Basic 2-bit Predictor
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction



Prediction Accuracy



- Two-bit prediction buffer w/ 4096 entries
  - 82% to nearly 100% accuracy
  - Misprediction rate of ~ 0% to 18%
  - 4K-entry buffer considered small (2005)

Example: Branch Prediction

<u>Example</u>

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

<u>Example in MIPS</u>

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

- Code Fragment: SPEC eqntott
  - Key Observation
    - Behavior of branch b3 is correlated with behavior of branches b1 and b2.

Branch Prediction

- Basic 2-bit Predictor
  - For *each* branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating Predictor
  - Multiple 2-bit predictors for *each* branch
  - One for each possible combination of outcomes of preceding *n* branches
- Local Predictor
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the *last n* occurrences of *this* branch
- Tournament Predictor
  - Combine correlating predictor with local predictor

Correlating Predictor

- Description:
  - (m,n): Uses behavior of last m branches to choose from 2^m branch predictors, each of which is an n-bit predictor for a single branch.
- Hardware required?
  - # of bits in an (m,n) predictor?
    - 2^m x n x # prediction entries selected by branch address.
  - Global history of the most recent m branches?
    - Use m-bit shift register to record.
- Example
  - (2,2) buffer with 64 total entries: 4 low-order address bits of branch and 2 global bits representing behavior of two most recently executed branches form a 6-bit index that can be used to index the 64 entries.

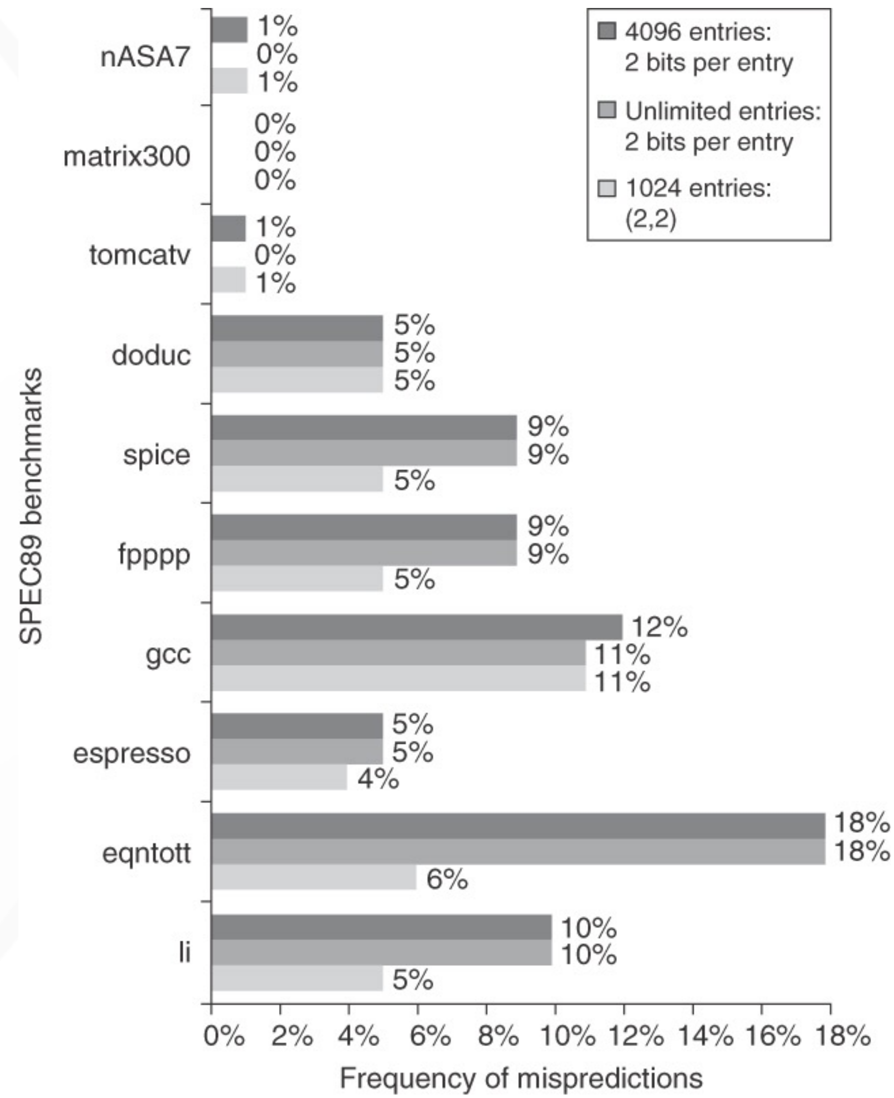
Example: Correlating Predictor

- How many bits are in the (0,2) branch predictor with 4k entries?

  2^0 *x 2 x 4k = 8k bits*

- How many entries are in the (2,2) predictor with the same number of bits?

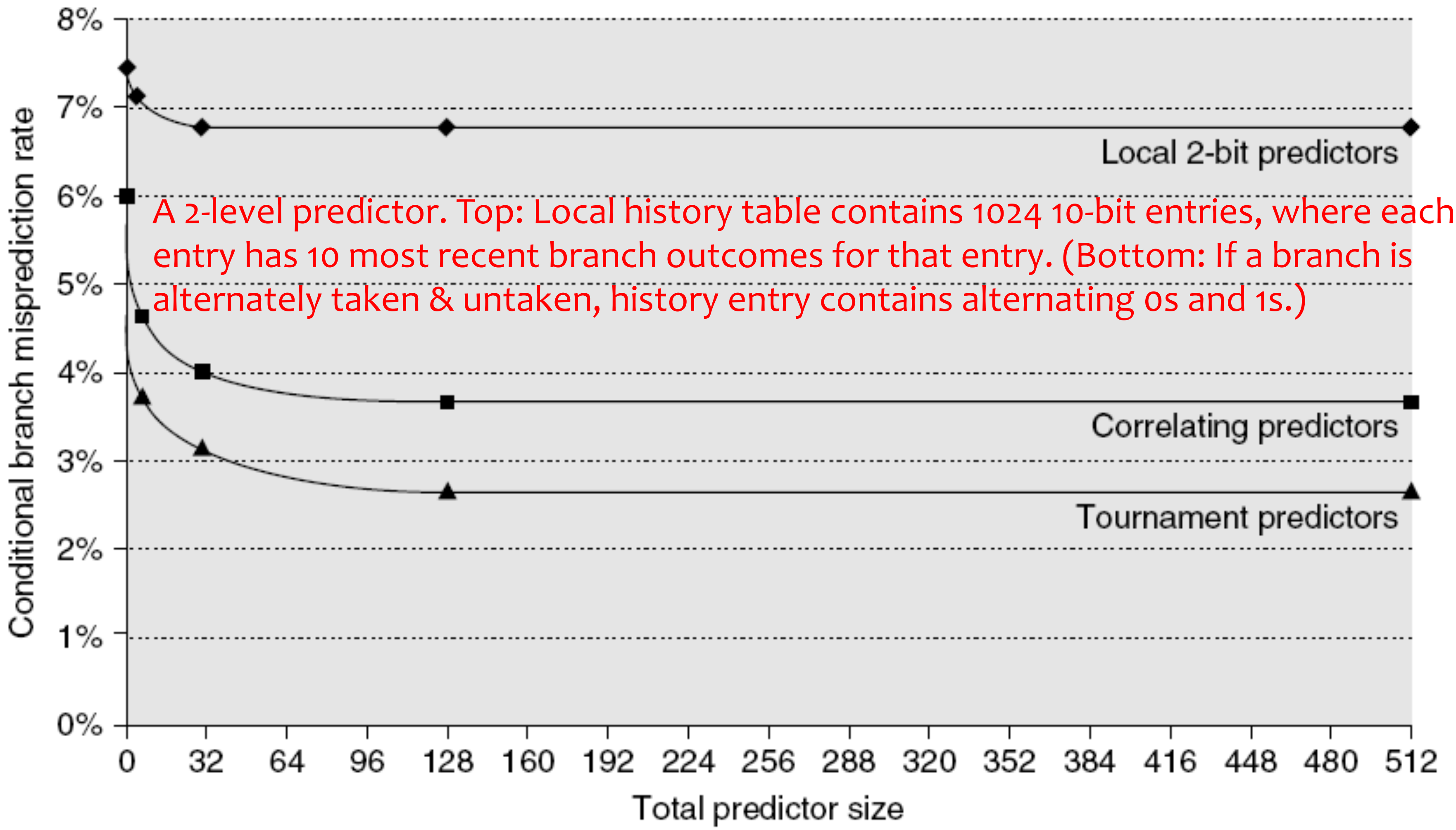
  2^2 *x 2 x # entries = 8k bits*
  # entries = 1k



Branch Prediction

- Basic 2-bit Predictor
  - For *each* branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating Predictor
  - Multiple 2-bit predictors for *each* branch
  - One for each possible combination of outcomes of preceding *n* branches
- Local Predictor
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the *last n* occurrences of *this* branch
- Tournament Predictor
  - Combine correlating predictor with local predictor

Branch Prediction Performance



Branch predictor performance on SPEC89

Misprediction Rates for Intel Core i7 Branch Predictor

- Slightly higher on average for the integer benchmarks than for the FP (4% versus 3%)