Branch Prediction

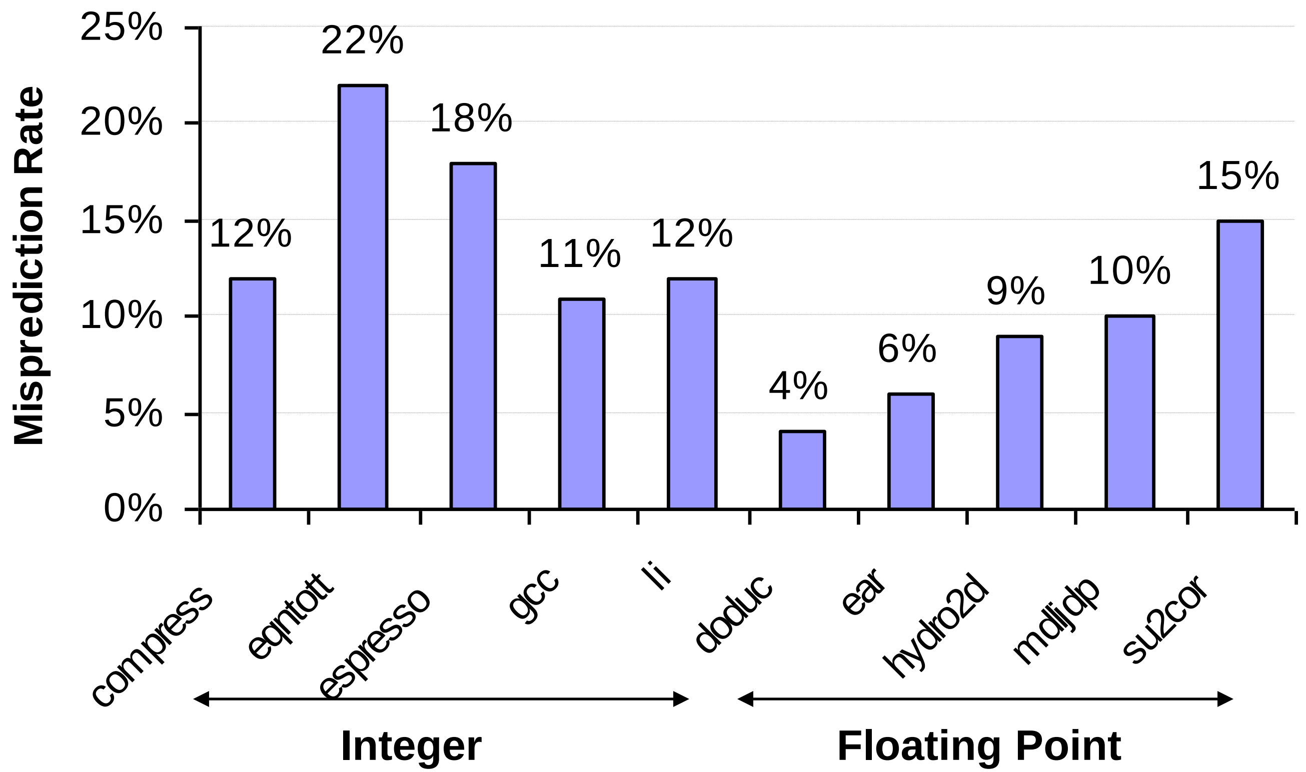
Branch Prediction

- As we try to exploit more ILP, the accuracy of branch prediction becomes critical.

Branch Prediction

- We have learned scheduling code in delayed slots
- To reorder code around branches, we need to predict branch statically at compile time
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% for SPEC

• **More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run.**



Dynamic Branch Prediction

- Performance = *ƒ*(accuracy, cost of misprediction, freq of branch)

- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies that are artifacts of the way that humans/compilers think about problems

- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - There are a small number of important branches in programs which have dynamic behavior

Branch History Table (Branch-Prediction Buffer)

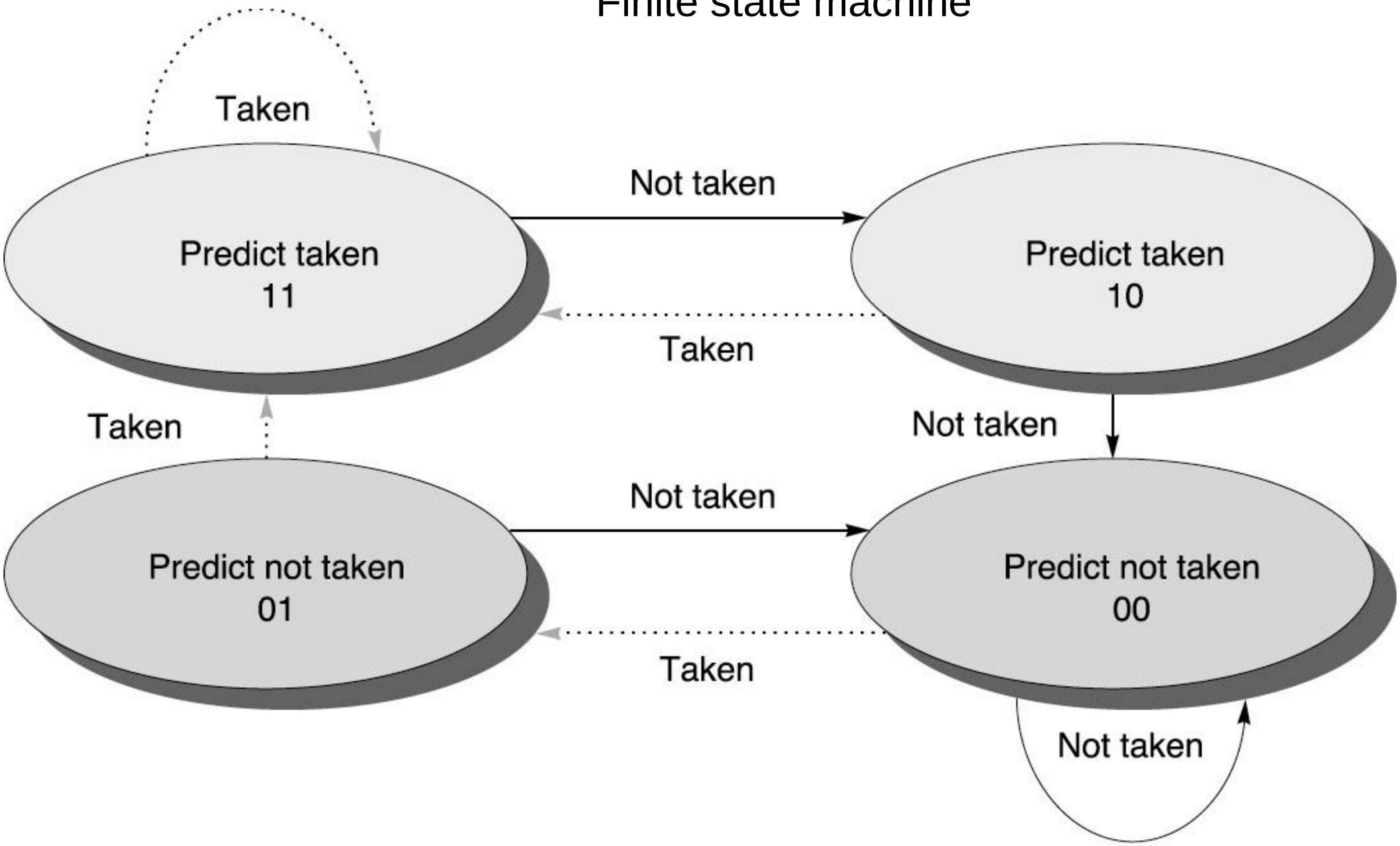
- A memory indexed by the lower bits of PC address
- The memory contains one bit: One-bit prediction scheme
- Says whether or not the branch taken last time
- No tag and no address check: It saves hardware, but we don’t know if it is the right branch. (It may have been put there by another branch that has the same low-order address bits.

Adding more prediction bits

- In a loop, 1-bit branch prediction will cause 2 mis-predictions
  - Enter the loop
  - End of loop
- To remedy this weakness, 2-bit prediction schemes are often used.
- A prediction must miss twice before it is changed.

2-bit Dynamic Branch Prediction

Finite state machine



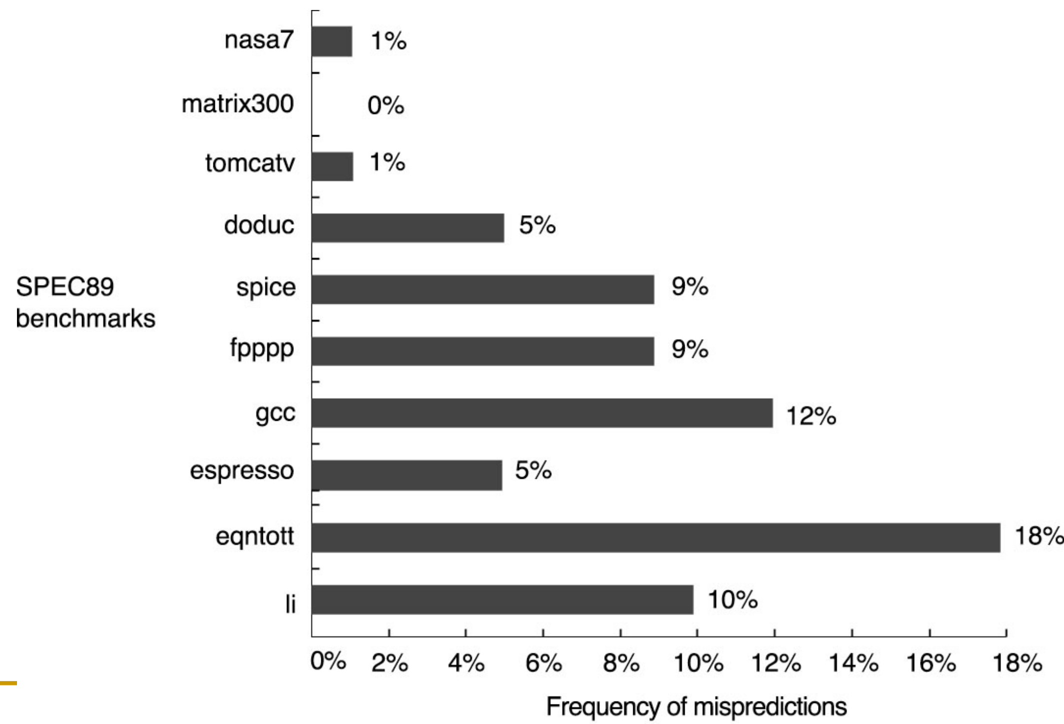
Implementing a Branch-Prediction Buffer

- Alternative 1: a small, special “cache” accessed with the instruction address during the IF stage

- Alternative 2: a pair of bits attached to each block in the instruction cache and fetched with the instruction

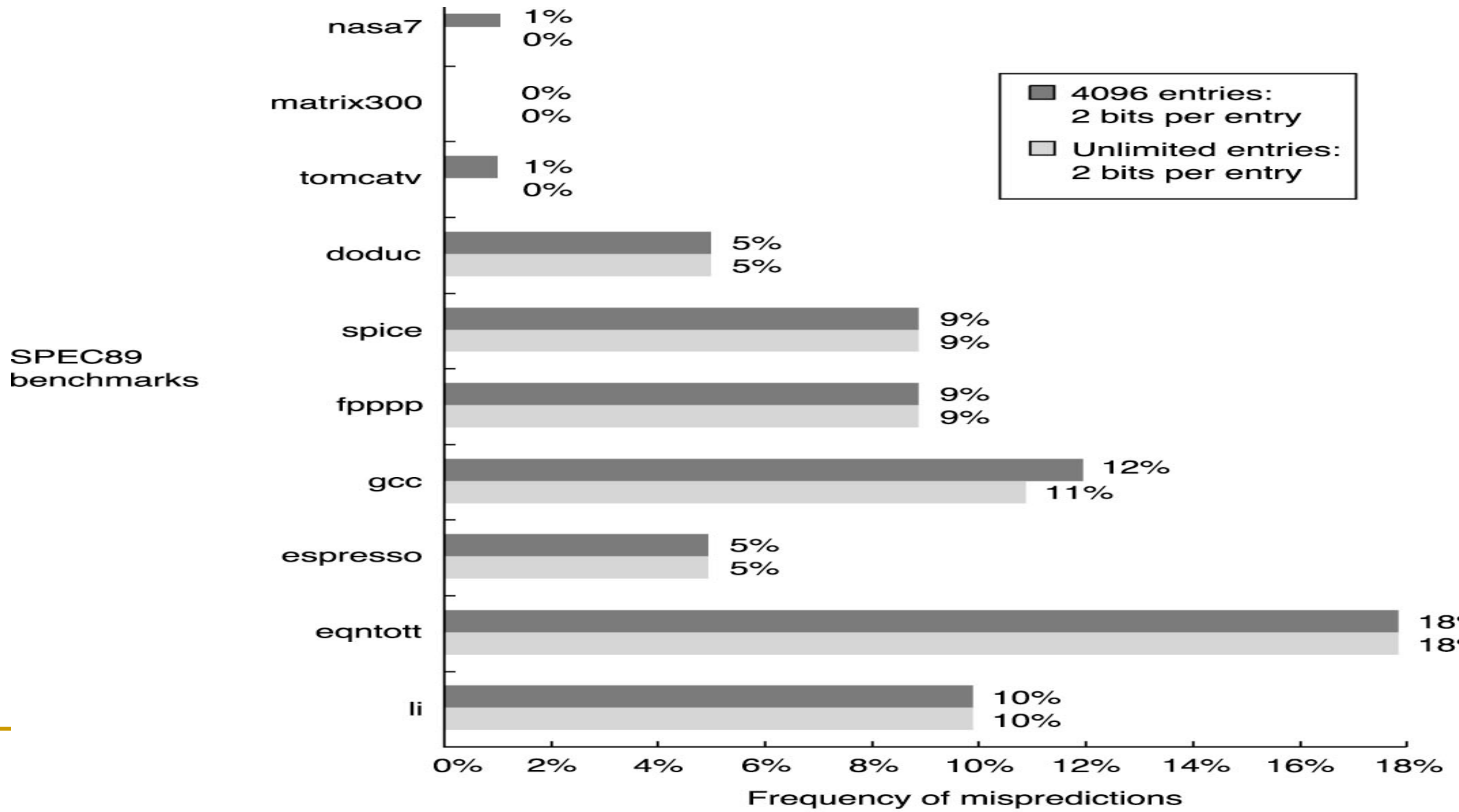
Branch History Table Accuracy

- Mis-predict because
  - Wrong guess for the branch
  - Got branch history of wrong branch when indexing the table



Branch History Table Accuracy

- 4096 entry table programs vary from 1% mis-prediction (nasa7, tomcatv) to 18%(eqntott), with Spice 9% and gcc 12%
- 4096 about as good as infinite table for older version of a subset of SPEC benchmark. For newer version, about 8K entries would be needed to match infinite 2-bit predictor.



How Wrong Can Branch Predication Be?

```
if (d==0)
    d=1;
if (d==1)
```

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

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

Correlating Predictors

With one bit predictor with one bit of correlation

Prediction bits	Prediction if last branch not taken	Prediction if last branch taken
NT/NT	Not taken	Not taken
NT/T	Not taken	Taken
T/NT	Taken	Not taken
T/T	Taken	Taken

Correlating Predictors

```
if (d==0)
    d=1;
if (d==1)
```

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

With one bit predictor with one bit of correlation

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

- *Except for first iteration, all branches are correctly predicted*

Correlating Predictors

- 2-bit prediction uses a small amount of (hopefully) local information to predict behaviour
- Sometimes the behaviour is correlated, and we can do better by keeping track of direction of related branches, for example consider the following code:

```
if (d==0)
    d = 1;
if (d==1) {
```

- If the first branch is not taken, neither is the second.
- Predictors that use the behaviour of other branches to make a prediction are called *correlating predictors* or *two-level predictors*

Correlating Branches

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

```
     DSUBUI   R3, R1, #2
     BNEZ     R3, L1        ; branch b1 (aa!=2)
     ANDI     R1, R1, #0    ; aa=0
L1:  SUBUI    R3, R2, #2
     BNEZ     R3, L2        ; branch b2 (bb!=2)
     ANDI     R2, R2, #0    ; bb=0
L2:  SUBU     R3, R1, R2    ; R3=aa-bb
     BEQZ     R3, L3        ; branch b3 (aa==bb)
```

- The behavior of branch b3 is correlated with the behavior of b1 and b2
- Clearly of both branches b1 and b2 are untaken, then b3 will be taken
- A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior

Correlating Branches

- Hypothesis: recent branches are correlated; i.e. behavior of recently executed branches affects prediction of current branch

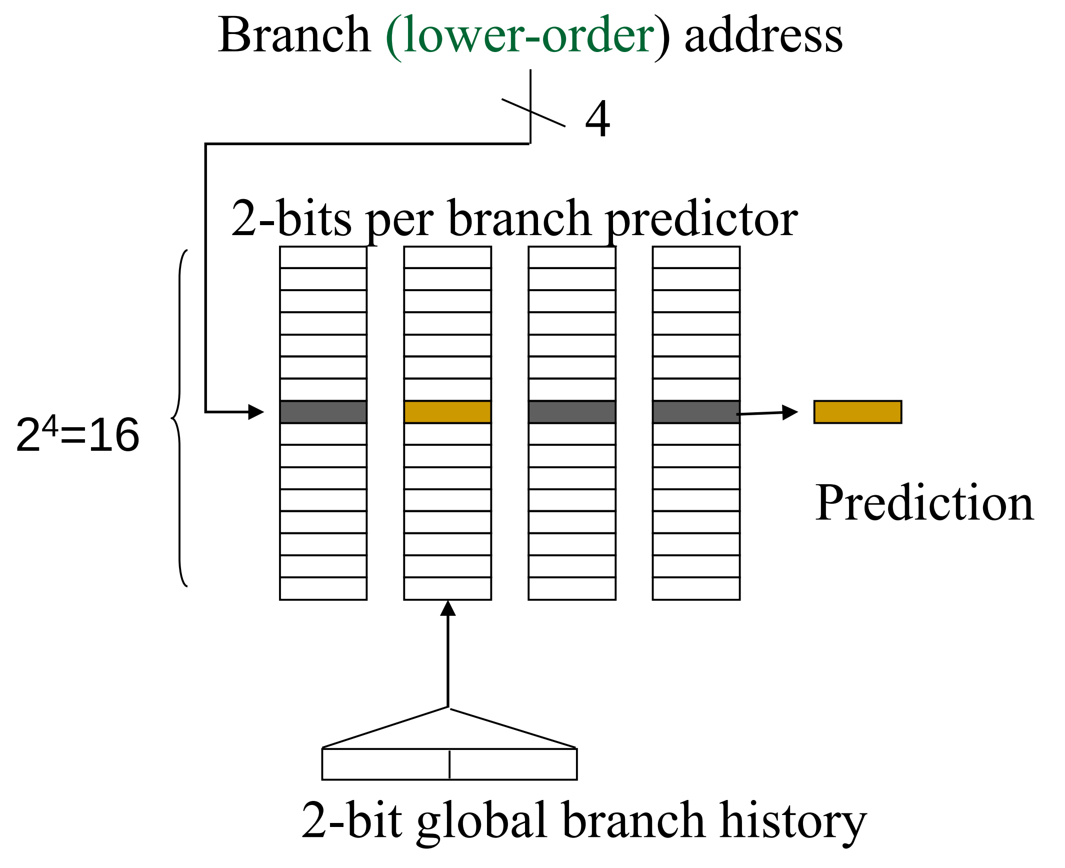
- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table

- In general, (m,n) predictor means record last m branches to select between 2^m history tables each with n-bit counters
  - 2-bit BHT is a (0,2) predictor

Correlating Branches

(2,2) predictor

- Behavior of recent branches selects between four predictions of next branch, updating just that prediction



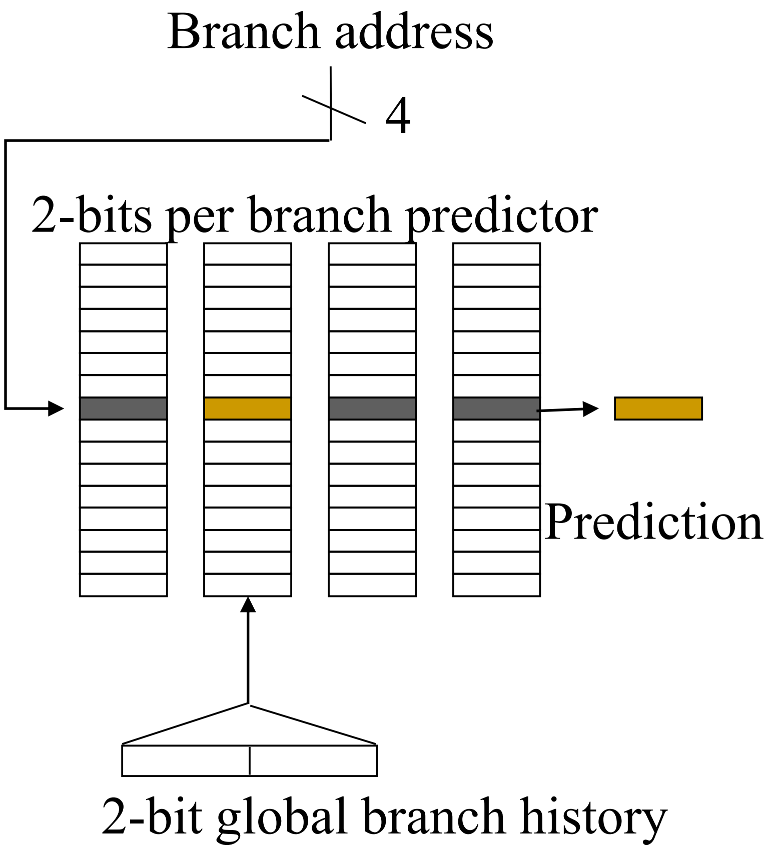
Total 64 entries, each entry has 2 bit

The number of bits in an (m,n) predictor

- Total bits = 2^m x n x Number of prediction entries selected by the branch address
- There are total

  $2^2 \times 2 \times 2^4 = 128$ bits

  in the predictor in this figure.



The number of bits in an (m,n) predictor

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

- Ans:

  $2^0 x 2 x 4K = 8K$ bits

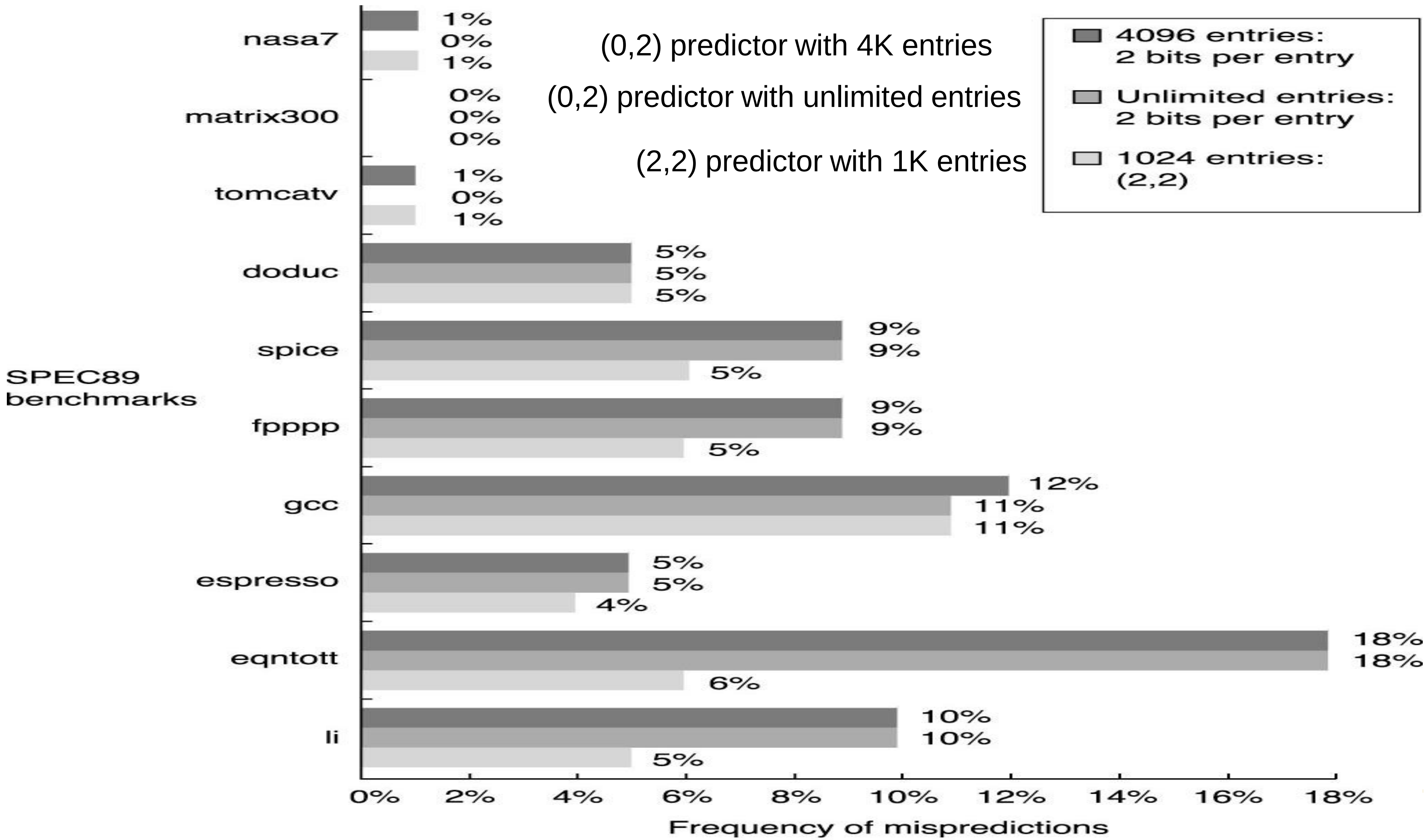
The number of bits in an (m,n) predictor

- How many branch-selected entries are in a (2,2) predictor with a total of 8K bits in the branch prediction buffer?
- How many lower order address bits used to index

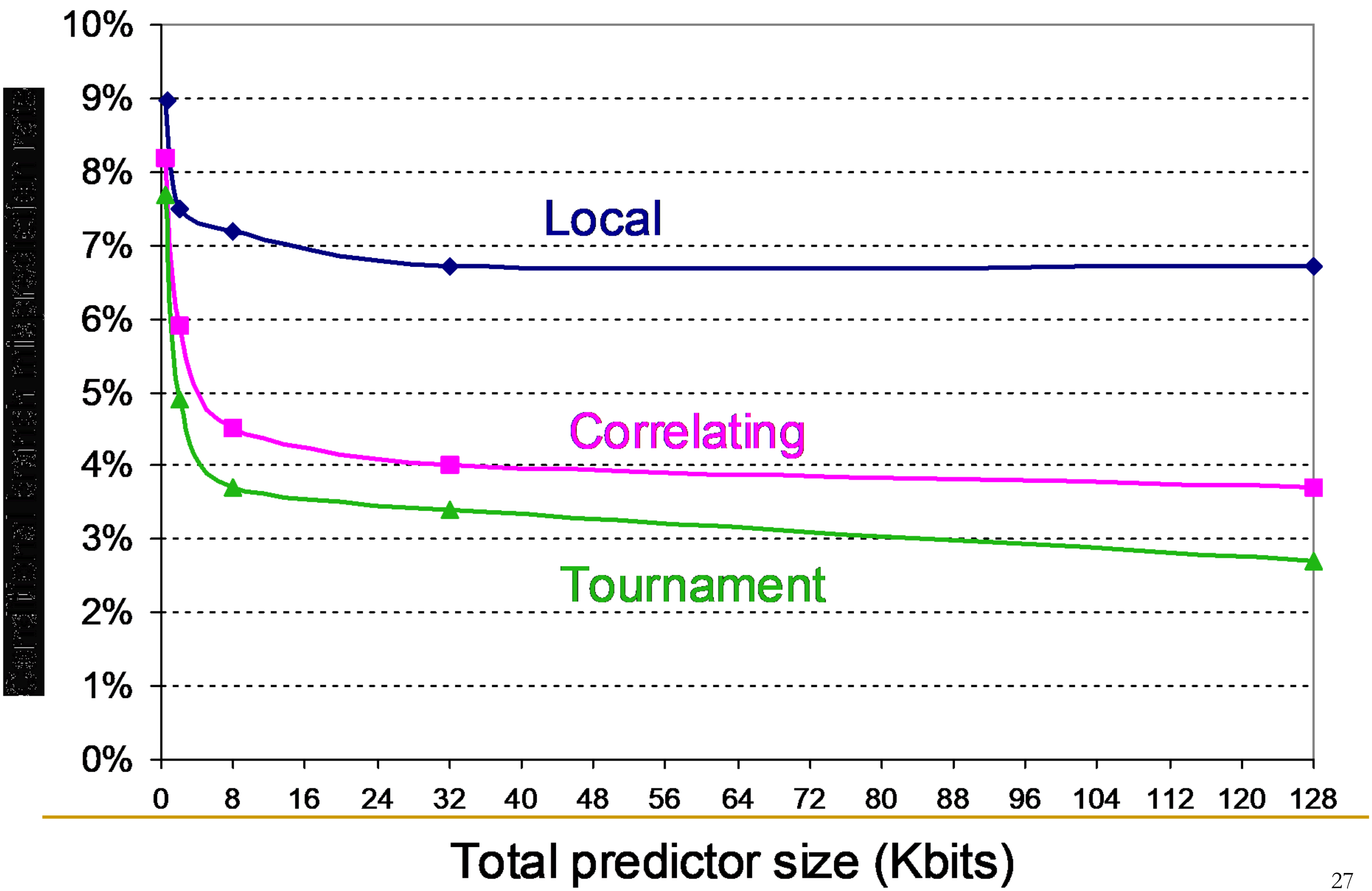
Ans:

- 8K = 2^2 x 2 x Number of prediction entries selected by the branch address
- Number of prediction entries selected by the branch address = 1K
- 10 lower order address bits used to index

Accuracy of Different Schemes

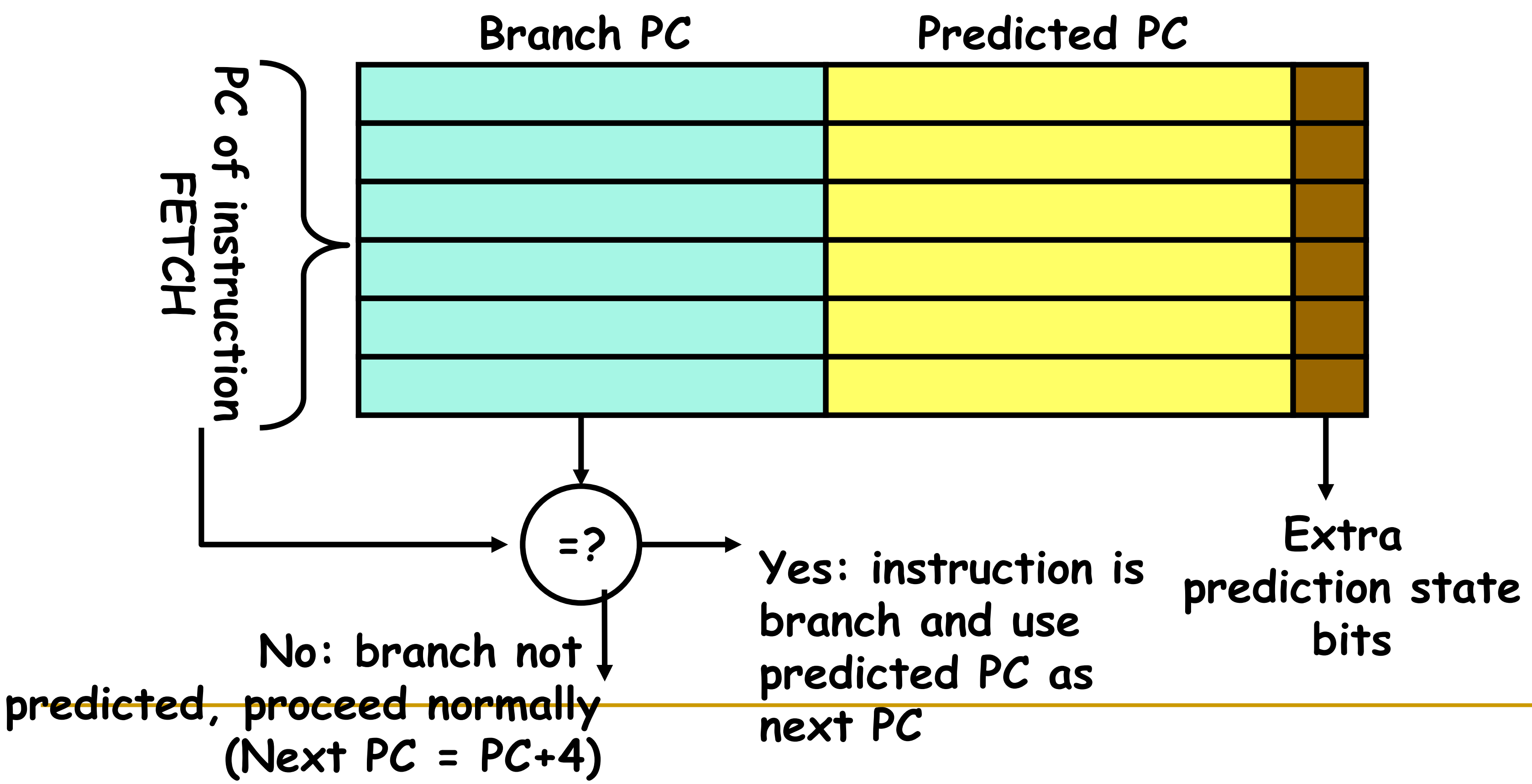


Accuracy v. Size (SPEC89)

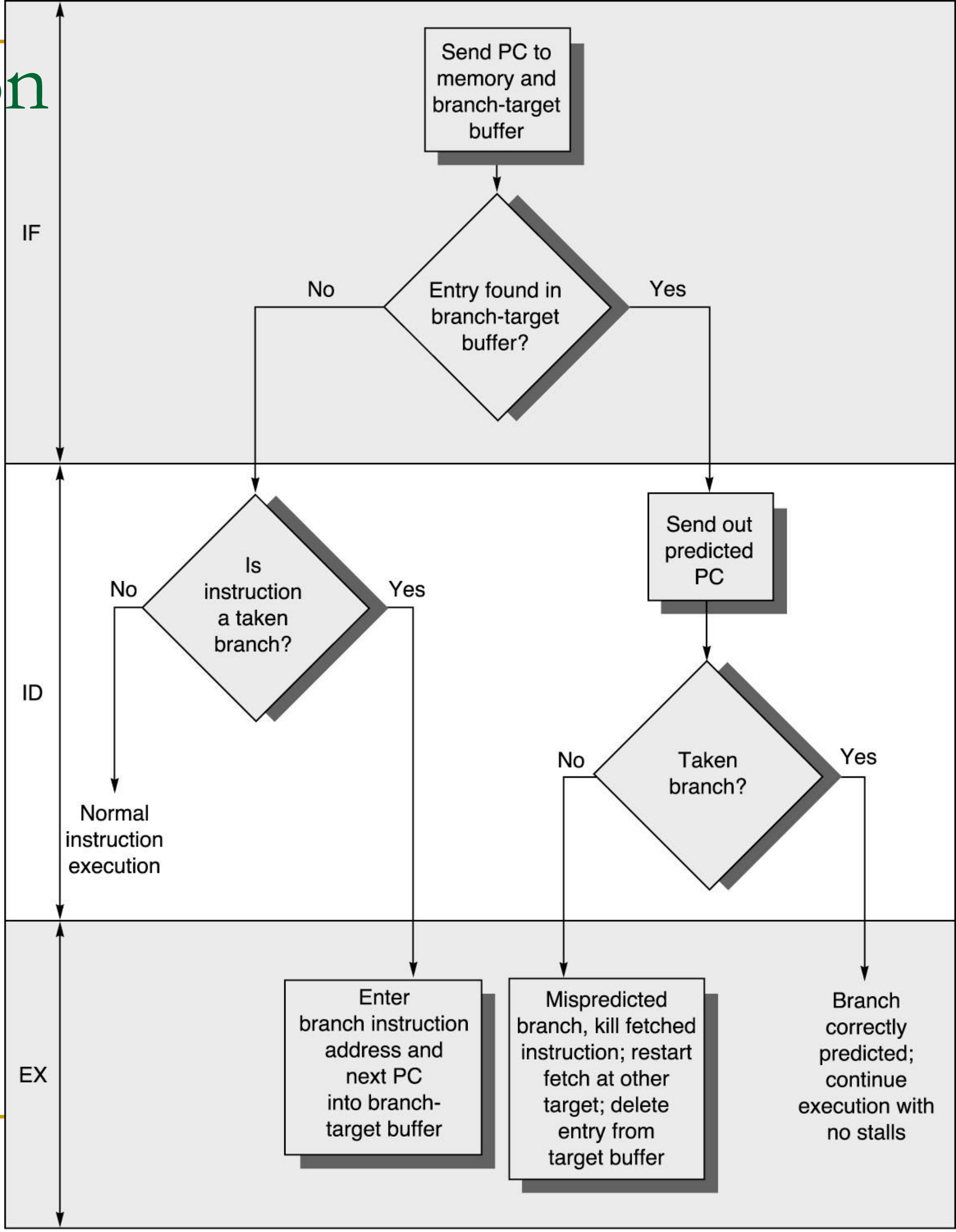


Need Address at the Same Time of Prediction

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since we can't use the wrong branch address



Branch Predication w/ BTB



Dynamic Branch Prediction Summary

- Prediction is important
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch.
  - Either different branches or different executions of the same branches
- Tournament Predictor: more resources for competitive solutions and pick between them
- Branch Target Buffer: include branch address & prediction

Exceptions and Interrupts

Example: Device Interrupt

(Say, arrival of network message)

External Interrupt

```
        …
add     r1,r2,r3
subi    r4,r1,#4
slli    r4,r4,#2
```

Hiccup(!)

```
lw      r2,0(r4)
lw      r3,4(r4)
add     r2,r2,r3
sw      8(r4),r2
        …
```

PC saved
Disable All Ints
Supervisor Mode

Restore PC
User Mode

"Interrupt Handler"

Raise priority
Reenable All Ints
Save registers

```
        …
lw      r1,20(r0)
lw      r2,0(r1)
addi    r3,r0,#5
sw      0(r1),r3
        …
```

Restore registers
Clear current Int
Disable All Ints
Restore priority
RTE

Alternative: Polling

(again, for arrival of network message)

External Interrupt

```
                 Disable Network Intr
                       ...
                 subi  r4,r1,#4
                 slli  r4,r4,#2
                 lw    r2,0(r4)
                 lw    r3,4(r4)
                 add   r2,r2,r3
                 sw    8(r4),r2
                 lw    r1,12(r0)
                 beq   r1,no_mess
                 lw    r1,20(r0)
                 lw    r2,0(r1)
                 addi  r3,r0,#5
                 sw    0(r1),r3
                 Clear Network Intr
no_mess:               ...
```

Polling Point
(check device register)

"Handler"

Polling is faster/slower than Interrupts.

- Polling is faster than interrupts because
  - Compiler knows which registers in use at polling point. Hence, do not need to save and restore registers (or not as many).
  - Other interrupt overhead avoided (pipeline flush, trap priorities, etc).
- Polling is slower than interrupts because
  - Overhead of polling instructions is incurred regardless of whether or not handler is run. This could add to inner-loop delay.
  - Device may have to wait for service for a long time.
- When to use one or the other?
  - Frequent/regular events good for polling, *as long as device can be controlled at user level.*
  - Interrupts good for infrequent/irregular events

Exceptions and Interrupts

- Arithmetic trap (overflow, divided by zero…)
- Using an undefined instruction
- Hardware malfunction
- Invoking an operating system service from a user program
- I/O device request

Exception/Interrupt classifications

- *Exceptions:* relevant to the current process
  - Faults, arithmetic traps
  - Invoke software on behalf of the currently executing process

- *Interrupts:* caused by asynchronous, outside events
  - I/O devices requiring service (DISK, network)

A related classification: Synchronous vs. Asynchronous

- *Synchronous:* means related to the instruction stream, i.e. during the execution of an instruction
  - Must stop an instruction that is currently executing
  - Page fault on load or store instruction
  - Arithmetic exception
  - Software Trap Instructions
- *Asynchronous:* means unrelated to the instruction stream, i.e. caused by an outside event.
  - Does not have to disrupt instructions that are already executing
  - Interrupts are asynchronous

Precise Interrupts/Exceptions

- An interrupt or exception is considered *precise* if there is a single instruction (or interrupt point) for which:
  - All instructions before that have committed their state
  - No following instructions (including the interrupting instruction) have modified any state.
- This means, that you can restart execution at the interrupt point and "get the right answer"
  - Implicit in our previous example of a device interrupt:
    - Interrupt point is at first lw instruction