

Computer Architecture

Lecture 5: Compiler Techniques for ILP & Branch Prediction (Chapter 3)

Chih-Wei Liu 劉志尉

National Chiao Tung University

cwliu@twins.ee.nctu.edu.tw

Data Dependence and Parallelism

- If 2 instructions are **parallel**
 - they can be executed simultaneously in a pipeline without causing any stalls (except the structural hazards)
 - their execution order can be swapped
- If 2 instructions are **dependent**
 - they **must be executed in order** or partially overlapped.
- To exploit parallelisms over instructions is equivalent to determine dependences over instructions

Exploit Instruction-Level Parallelism

- Two main approaches:
 - Hardware-based dynamic approaches
 - Hardware locates the parallelism in run-time
 - Used in server and desktop processors (Not used as extensively in PMP processors)
 - Superscalar processors: Pentium 4, IBM Power, AMD Opteron
 - Compiler-based static approaches
 - Software finds parallelism at compile-time
 - Used in DSP processors (Not as successful outside of scientific applications)
 - VLIW processors: Itanium 2, ITRI PAC

Why ILP?

- Multi-issue Processor: two or more instructions can be issued (or executed) in parallel
 - The goal is to maximize IPC (instruction per cycle)
 - Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls
 - To reduce the impact of data and control hazards
 - Basic block ILP
- Can we make CPI closer to 1?
 - If we have n -cycle latency, then we need $n-1$ instructions between a producing instruction and its use

Basic Block ILP

- Basic Block (BB) ILP is quite small :
 - BB: a straight-line code sequence **with no branches in** except to the entry and **no branches out** except at the exit
 - average dynamic branch frequency 15% to 25%
=> 4 to 7 instructions execute between a pair of branches
 - Plus instructions in BB likely to depend on each other
- **We must exploit ILP across multiple basic blocks**

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

Loop-level
parallelism



```
x[1] = x[1] + y[1]
x[2] = x[2] + y[2]
...
x[1000]=x[1000]+y[1000]
```

- **Loop unrolling** to exploit loop-level parallelism

Overcome the Data Dependence

- Maintaining the dependence but avoiding a hazard
 - scheduling the code in HW/SW approach
- Eliminating a dependence by transforming the code
 - primary by software
- Dependence detection
 - by **register names**: simpler
 - by **memory locations**: more complicated
 - Two addresses may refer to the same location but look quite different (e.g. 100(R4), 20(R6) may be identical)
 - The effective address of a load/store may changed from instruction to instruction (20(R4), 20(R4) may be different)

(True) Data Dependence

- Data and Control dependencies are a property of the program (application)
- Data dependence conveys:
 - Possibility of a pipeline hazard
 - Order in which results must be calculated (i.e. program behavior)
 - Upper bound on ILP
- Dependencies that flow through memory locations are difficult to detect
 - Hardware-based dynamic approach is more attractive

Name Dependence

- Two instructions use the same register or memory location (i.e. the same name) but no flow of information
 - Not a true data dependence, *but is a problem when reordering instructions or a irregularly pipelined datapath.*
 - *Anti-dependence*: instruction j writes a register or memory location that instruction i reads
 - Initial ordering (i before j) must be preserved
 - *Output-dependence*: instruction i and instruction j write the same register or memory location
 - Ordering must be preserved
- To resolve, use renaming techniques

Register Renaming and WAW/WAR

- 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

- WAW: ADD.D/MUL.D
- WAR: ADD.D/SUB.D, S.D/MUL.D
- RAW: DIV.D/ADD.D, ADD.D/S.D
SUB.D/MUL.D

- 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

Register renaming result

Control Dependence

- Control Dependence
 - Ordering of instruction i with respect to a branch instruction
 - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
 - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Control Dependence Ignored

- The method to preserve the control dependence
 - Be used in most simple pipeline CPUs
 - Simple but inefficient
- Control dependence is not the critical property that must be preserved
 - We may execute instruction that should not have been executed, thereby violating the control dependence, if we can do so without affecting the correctness of the program
- The two properties critical to program correctness are
 - The exception behavior
 - Data flow

Control Dependence Examples

- Example 1:

DADDU R1,R2,R3

BEQZ R4,L

DSUBU R1,R1,R6

L: ...

OR R7,R1,R8

- R1 in OR instruction depends on DADDU or DSUBU, relied on the branch is taken or not.

- Example 2:

DADDU R1,R2,R3

BEQZ R12,skip

DSUBU R4,R5,R6

DADDU R5,R4,R9

skip:

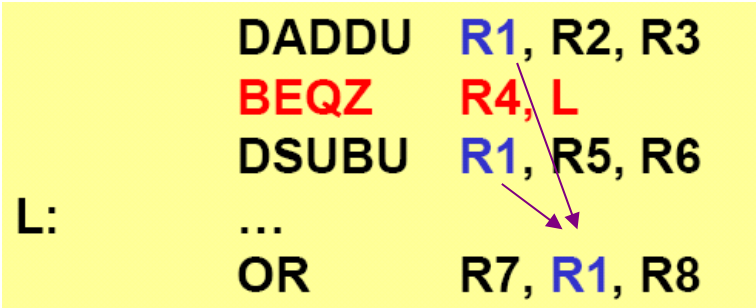
OR R7,R8,R9

- Assume R4 isn't used after skip
 - Possible to move DSUBU before the branch (this violates the control dependence but not affects the data flow)

Preserve Data Flow Behavior

- The data flow is **the actual flow of data** among instructions
- Branches make data flow dynamic
- Example

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



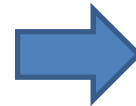
- R1 value depends on the branch is taken or not
- DSUBU cannot be moved above the branch.
- Speculation should take care this problem
 - Program order, that determines which predecessor will actually deliver a data value to the instruction, should be ensured by maintaining the control dependences

Compiler Techniques for Exposing ILP

- Pipeline scheduling
 - Separate dependent instruction from the source instruction by the pipeline latency (or instruction latency) of the source instruction

- Example:

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

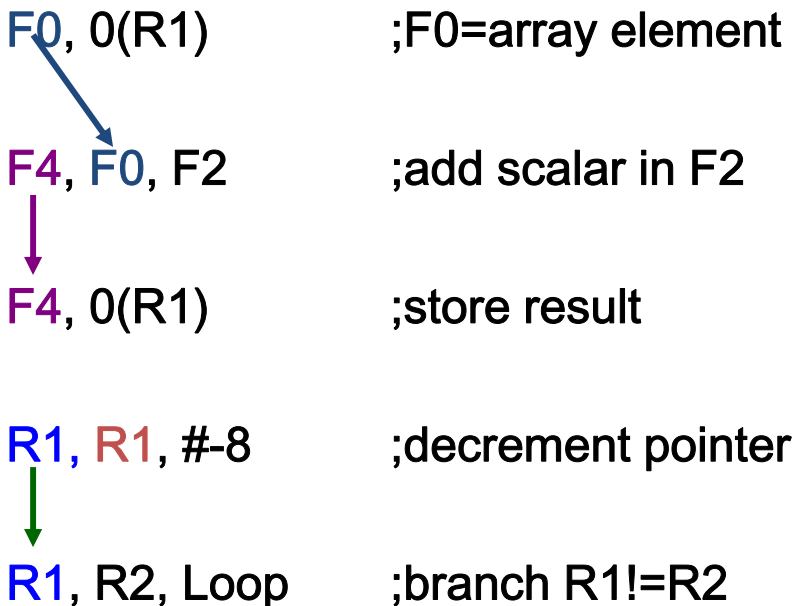


```
Loop:  L.D    F0,0(R1)
        ADD.D F4,F0,F2
        S.D    F4,0(R1)
        DADDUI R1,R1,#-8
        BNE   R1,R2,Loop
```

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

Data Dependence

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



- The arrows show the order that must be preserved for correct execution.
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped.

Step 1: Insert Pipeline Stalls

```

Loop:  L.D      F0,0(R1)
        stall
        ADD.D  F4,F0,F2
        stall
        stall
        S.D    F4,0(R1)
        DADDUI R1,R1,#-8
        stall      (assume integer load latency is 1)
        BNE   R1,R2,Loop
    
```

9 C.C/ iteration

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

Step 2: Re-Scheduling

Scheduled code:

```

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
    
```

7 C.C/ iteration

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

Step 3: Loop Unrolling

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

```

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
    
```

- note: number of live registers vs. original loop

Step 4: Re-Schedule the Unrolled loop

- Pipeline schedule the unrolled loop:

```
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)
      S.D F16,8(R1)
      BNE R1,R2,Loop
```

14 C.C/ 4 iterations
or 3.5 C.C/ iteration

ILP and Data Dependencies

- HW/SW must preserve **program order**:
order instructions would execute in if executed sequentially as determined by original source program
 - Dependences are a property of programs
- Presence of dependence indicates **potential** for a hazard, but actual hazard and length of any **stall is property of the pipeline**
- Importance of the data dependencies
 - 1) indicates the possibility of a hazard
 - 2) determines order in which results must be calculated
 - 3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order **only where it affects the outcome of the program**

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n , and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
 - 1st executes $(n \bmod k)$ times and has a body that is the original loop
 - 2nd 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

5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:
 1. Determine loop unrolling useful by finding that **loop iterations were independent** (except for maintenance code)
 2. Use different registers to avoid unnecessary constraints forced by **using same registers for different computations**
 3. **Eliminate the extra test and branch instructions** and **adjust the loop termination and iteration code**
 4. Determine that **loads and stores in unrolled loop can be interchanged** by observing that loads and stores from different iterations are independent
 - Transformation requires analyzing memory addresses and finding that they do not refer to the same address
 5. **Schedule the code**, preserving any dependences needed to yield **the same result as the original code**

3 Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
 - Amdahl's Law
2. Growth in code size
 - For larger loops, concern it increases the instruction cache miss rate
3. Register pressure (compiler limitation): potential shortfall in registers created by aggressive unrolling and scheduling
 - If not be possible to allocate all live values to registers, may lose some or all of its advantage
 - Loop unrolling reduces impact of branches on pipeline; another way is **branch prediction**

Getting CPI below 1

- $CPI \geq 1$ if issue only 1 instruction every clock cycle
- **Multiple-issue processors** come in 3 flavors:
 1. **statically**-scheduled **superscalar** processors,
 2. **dynamically**-scheduled superscalar processors, and
 3. **VLIW** (very long instruction word) processors
- 2 types of superscalar processors issue **varying numbers of instructions per clock**
 - use in-order execution if they are statically scheduled, or
 - out-of-order execution if they are dynamically scheduled
- VLIW processors, in contrast, issue **a fixed number of instructions** formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (Intel/HP Itanium)

Basic VLIW

(Very Long Instruction Word)

- A VLIW uses multiple, **independent** functional units
- A VLIW packages multiple independent operations into one very long instruction
 - The burden for choosing and packaging independent operations falls on compiler
 - HW in a superscalar makes these issue decisions unnecessary
- VLIW depends on enough parallelism for keeping FUs busy
 - Loop unrolling and then code scheduling
 - Compiler may need to do local scheduling and global scheduling
- Here we consider a VLIW processor might have instructions that contain 5 operations, including 1 integer (or branch), 2 FP, and 2 memory references
 - Depend on the available FUs and frequency of operation

VLIW: Very Large Instruction Word

- Each “instruction” has explicit coding for multiple operations
 - In IA-64, grouping called a “packet”
 - In Transmeta, grouping called a “molecule” (with “atoms” as ops)
- Tradeoff instruction space for simple decoding
 - The long instruction word has room for many operations
 - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
 - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
 - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
 - Need compiling technique that schedules across several branches

Recall: Unrolled Loop that Minimizes Stalls for Scalar

1	Loop:	L.D	F0, 0(R1)	
2		L.D	F6, -8(R1)	L.D to ADD.D: 1 Cycle
3		L.D	F10, -16(R1)	ADD.D to S.D: 2 Cycles
4		L.D	F14, -24(R1)	
5		ADD.D	F4, F0, F2	
6		ADD.D	F8, F6, F2	
7		ADD.D	F12, F10, F2	
8		ADD.D	F16, F14, F2	
9		S.D	0(R1), F4	
10		S.D	-8(R1), F8	
11		S.D	-16(R1), F12	
12		DSUBUI	R1, R1, #32	
13		BNEZ	R1, LOOP	
14		S.D	8(R1), F16	; 8-32 = -24

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW

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

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.29 clocks per iteration

23 ops in 9 clocks, average 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW

VLIW Problems

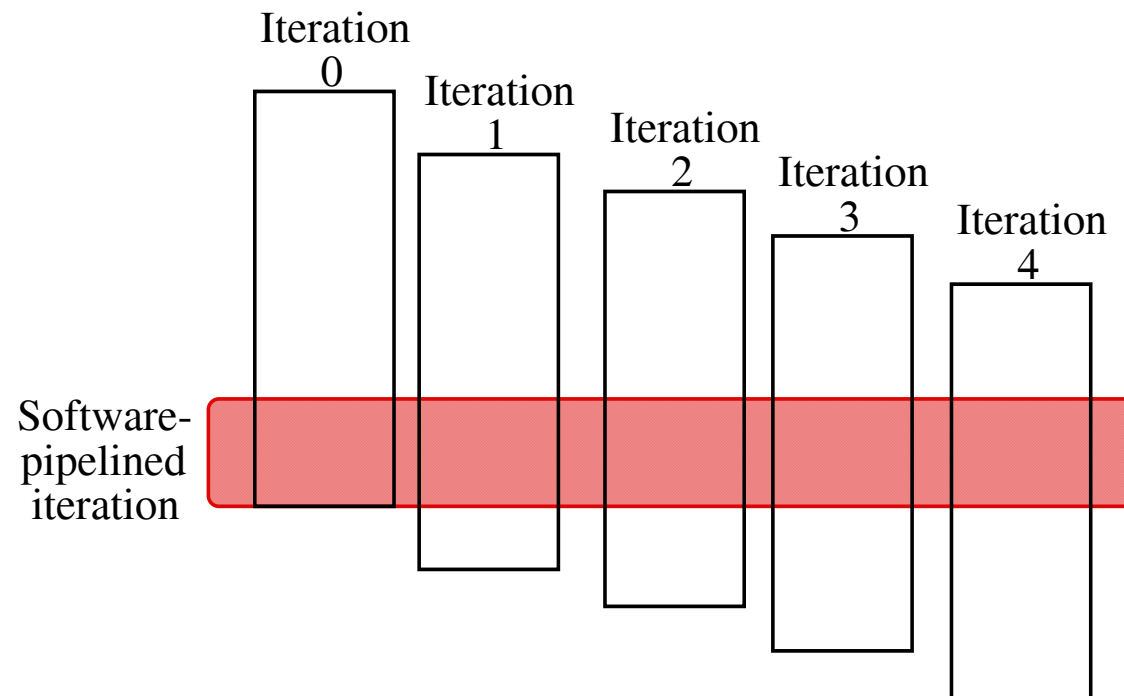
- Increase in code size
 - Ambitious loop unrolling
 - Whenever instructions are not full, the unused FUs translate to waste bits in the instruction encoding
 - An instruction may need to be left completely empty if no operation can be scheduled
 - **Clever encoding or compress/decompress**
- Binary code compatibility
 - different numbers of functional units and unit latencies require different versions of the code
 - Need re-compilation
 - Solution: Object-code translation or emulation

Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- [IA-64](#): instruction set architecture
- 128 64-bit integer regs + 128 82-bit floating point regs
 - Not separate register files per functional unit as in old VLIW
- Hardware checks dependencies (interlocks => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags)
=> 40% fewer mispredictions?
- [Itanium™](#) was first implementation (2001)
 - Highly parallel and deeply pipelined hardware at 800Mhz
 - 6-wide, 10-stage pipeline at 800Mhz on 0.18 μ process
- [Itanium 2™](#) is name of 2nd implementation (2005)
 - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 μ process
 - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3

Another Possibility: Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop



Control Hazard Avoidance

- Consider Effects of Increasing the ILP
 - Control dependencies rapidly become the limiting factor
 - They tend to not get optimized by the compiler
 - Higher branch frequencies result
 - Plus multiple issue (more than one instructions/sec) → more control instructions per sec.
 - Control stall penalties will go up as machines go faster
 - Amdahl's Law in action - again
- **Branch Prediction:** helps if can be done for reasonable cost
 - Static by compiler: appendix C
e.g. predict not taken, delay branch
 - Dynamic by HW: this section

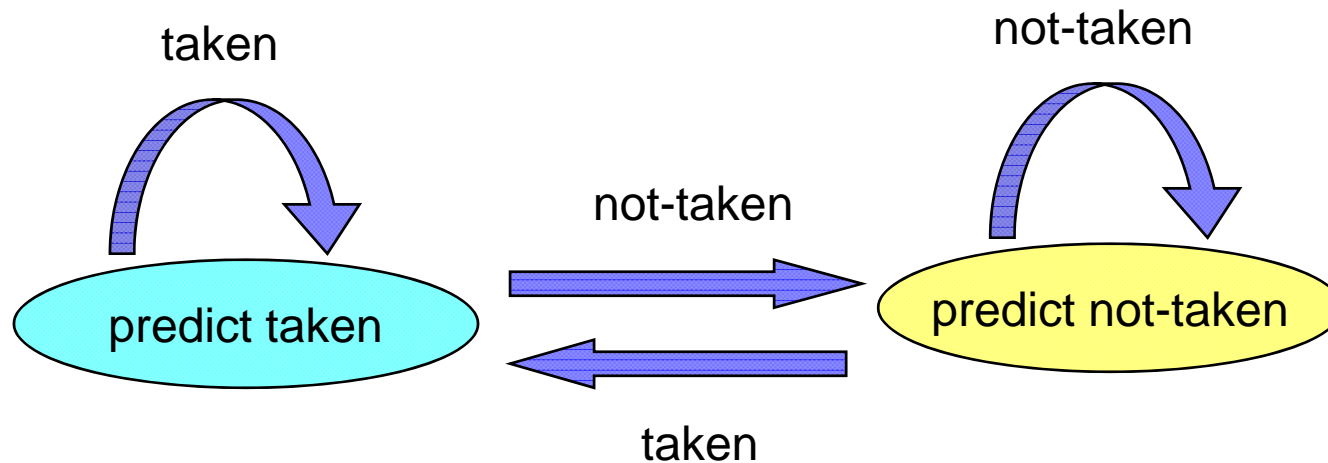
Dynamic Branch Prediction

- Why does prediction work?
 - Underlying algorithm has **regularities**
 - Data that is being operated on has **regularities**
 - Instruction sequence has **redundancies** that are artifacts of way that humans/compiler 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

Dynamic Branch Prediction

- The predictor will depend on the behavior of the branch at run time
- Goals:
 - allow the processor to resolve the outcome of a branch early, prevent control dependences from causing stalls
- Effectiveness of a branch prediction scheme depends not only on the accuracy but also on the cost of a branch
 - $BP_Performance = f(\text{accuracy}, \text{cost of misprediction})$
- Branch History Table (BHT)
 - Lower bits of PC address index table of 1-bit values
 - No “precise” address check – just match the lower bits
 - Says whether or not branch taken last time

1-bit Dynamic Hardware Prediction



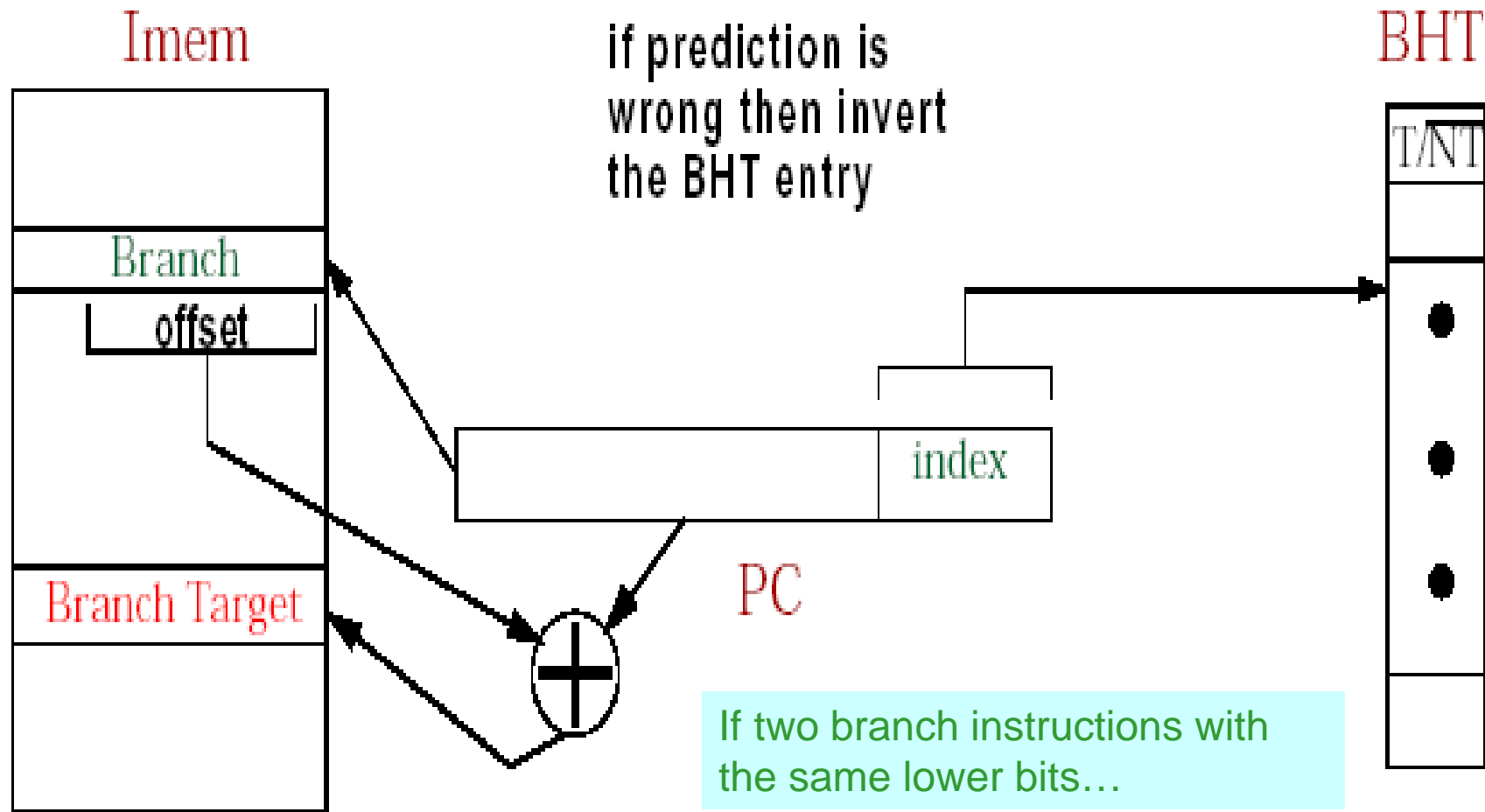
Problem: Loop case

LOOP:	LOAD	R1, 100(R2)
	MUL	R6, R6, R1
	SUBI	R2, R2, #4
	BNEZ	R2, LOOP

The steady-state prediction behavior will mispredict on the first and last loop iterations

BHT Prediction

Useful only for the target address is known before CC is decided



Problem with the Simple BHT

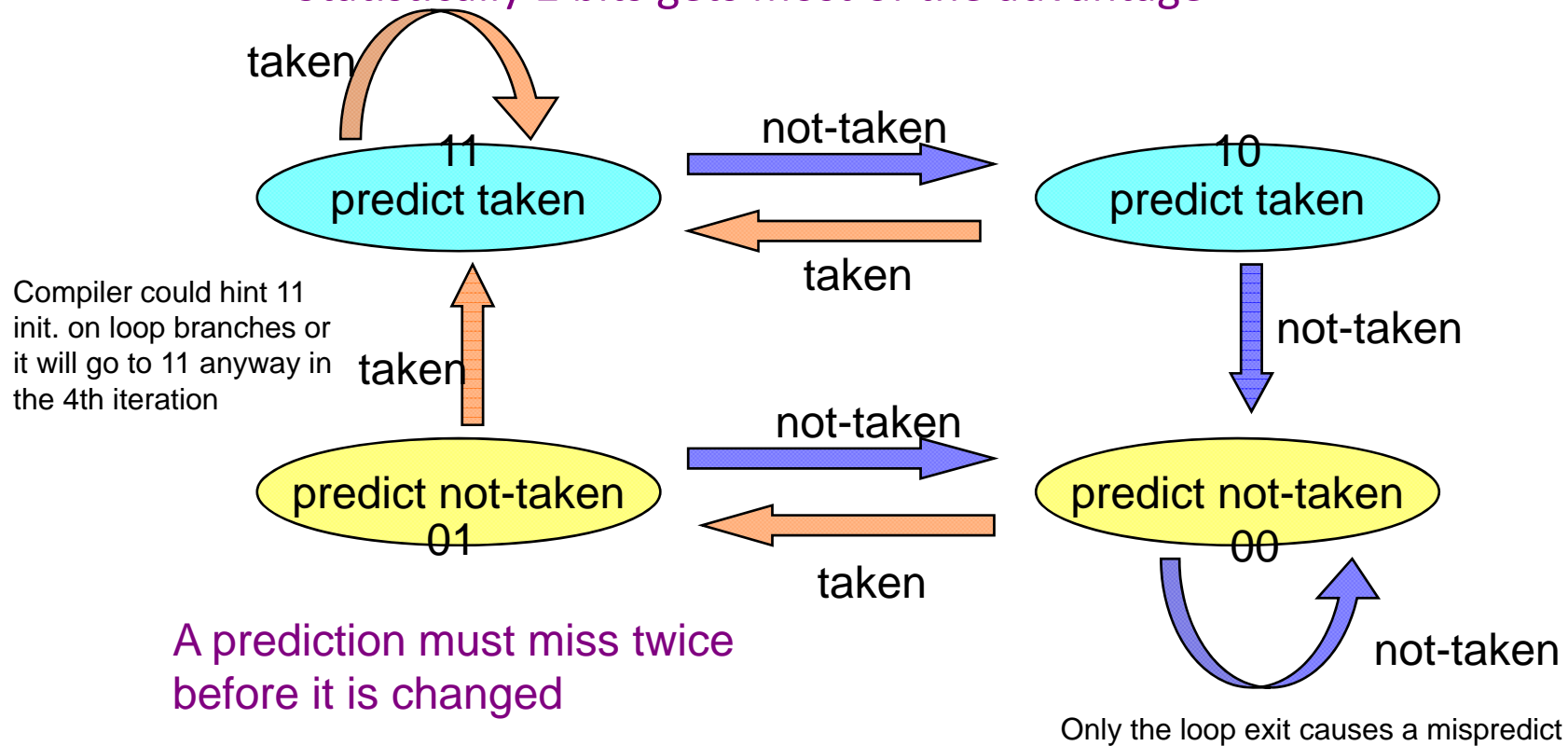
clear benefit is that it's cheap and understandable

- Aliasing
 - All branches with the same index (lower) bits reference same BHT entry
 - Hence they mutually predict each other
 - No guarantee that a prediction is right. But it may not matter anyway
 - Avoidance
 - Make the table bigger - OK since it's only a single bit-vector
 - This is a common cache improvement strategy as well
 - Other cache strategies may also apply
- Consider how this works for loops
 - Always mispredict twice for every loop
 - One is unavoidable since the exit is always a surprise
 - However previous exit will always cause a mis-prediction the first try of every new loop entry

N-bit Predictors

idea: improve on the loop entry problem

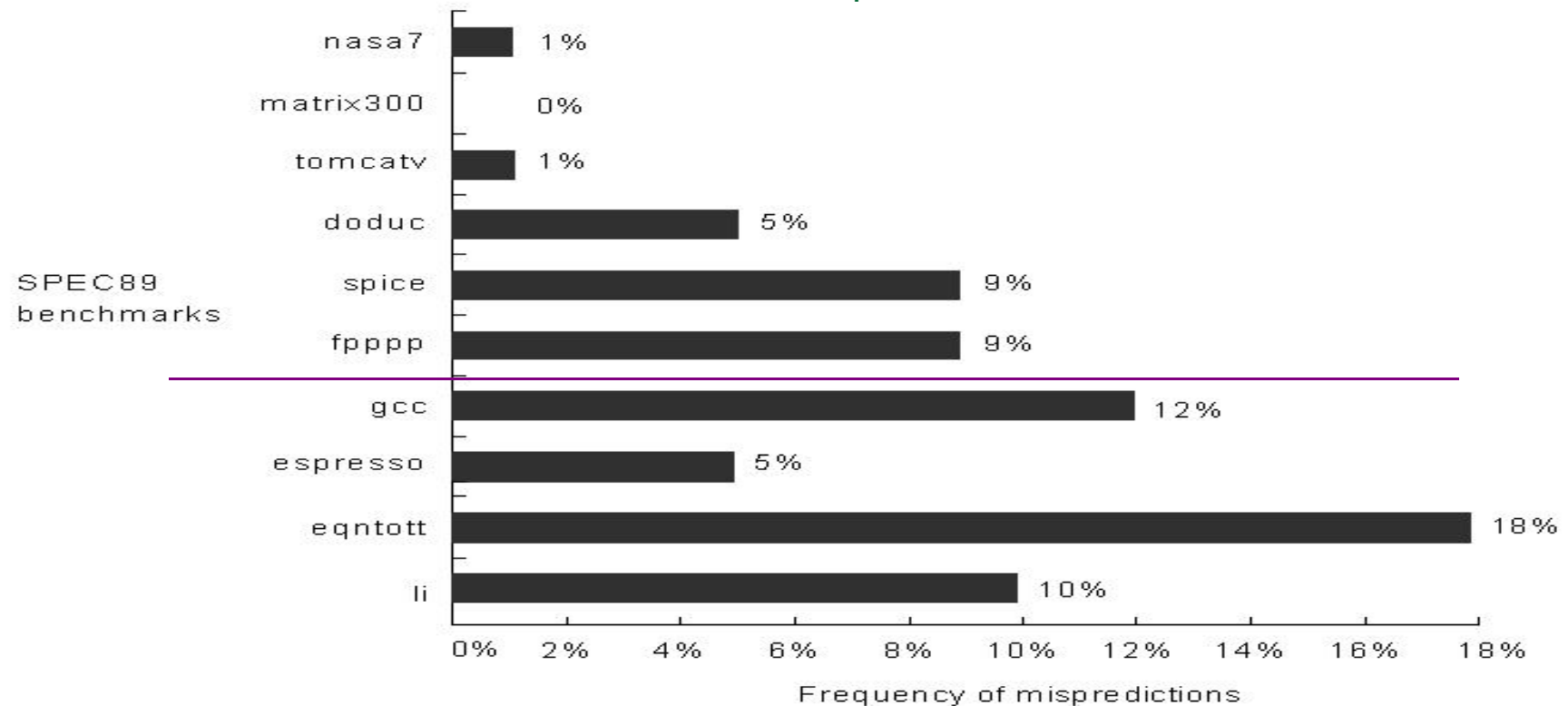
- 2-bit counter implies 4 states
 - Statistically 2 bits gets most of the advantage



BHT Accuracy

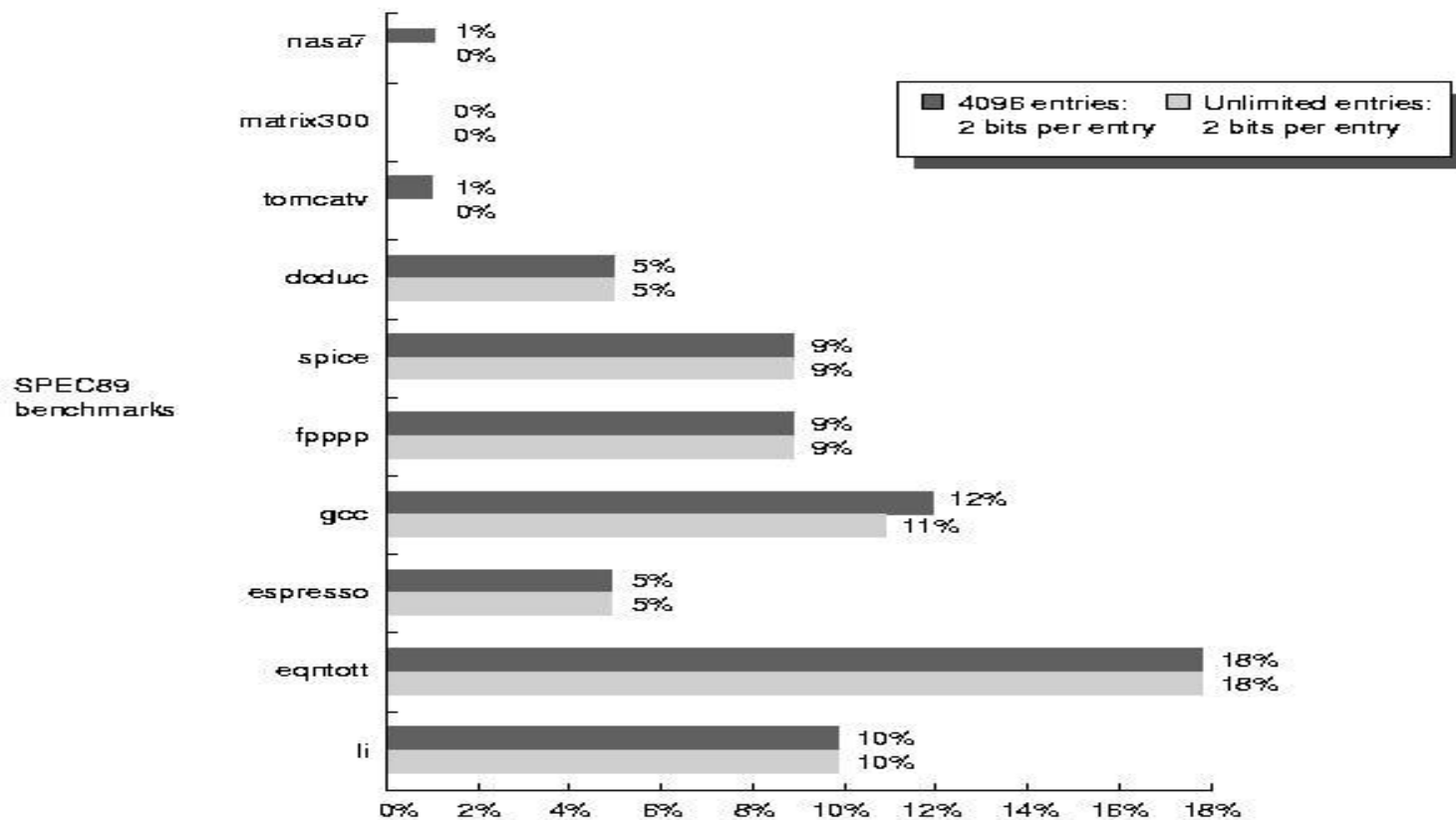
- Mispredict because either:
 - Wrong guess for that branch (accuracy)
 - Got branch history of wrong branch when index the table (size)

4K of BPB with 2-bit entries misprediction rates on SPEC89@IBM Power



To Increase the BHT Size

- 4096 about as good as infinite table
- The hit rate of the buffer is clearly not the limiting factor for an enough-large BHT size



The worst case for the 2-bit predictor

```

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

```

```

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

```

aa and bb are assigned to R1 and R2

**if the first 2 untaken then the
3rd will always be taken**

Improve Prediction Strategy By Correlating Branches

- Consider the worst case for the 2-bit predictor

```
if (aa==2) then aa=0;  
if (bb==2) then bb=0;  
if (aa != bb) then whatever
```

← if the first 2 fail then the 3rd will always be taken

– single level predictors can never get this case

- Correlating or 2-level predictors

– The predictor uses the behavior of other branch(es) to make a prediction

– Correlation = what happened on the last branch

– Predictor = which way to go

Correlating Branches

Two-level predictors

- Hypothesis: recently executed branches are correlated
- 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
 - Old 2-bit BHT is then a (0,2) predictor
- Global Branch History: m-bit shift register keeping T/NT status of last m branches
- Each entry in table has m n-bit predictors

2-Level (m,n) BHT

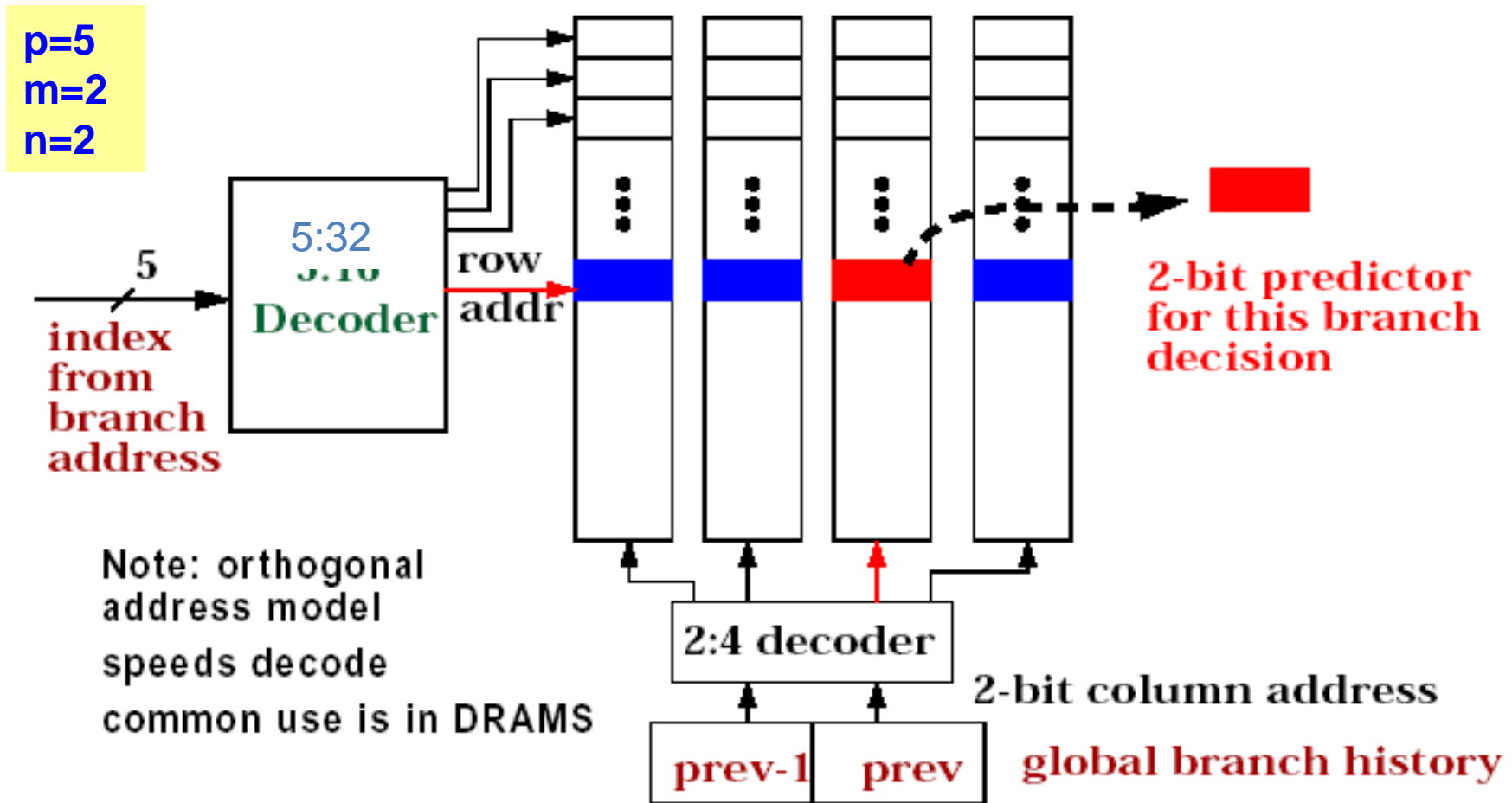
- Use the behavior of **the last m branches** to choose from 2^m branch predictors, each of which is an **n-bit predictor** for a single branch
- Total bits for the (m, n) BHT prediction buffer:

$$\text{Total_memory_bits} = 2^m \times n \times 2^p$$

- **p bits of buffer index** = 2^p bit BHT
- **2^m banks of memory** selected by the global branch history (which is just a shift register) - e.g. a column address
- Use p bits of the branch address to select row
- **Get the n predictor bits** in the entry to make the decision

(2,2) Predictor Implementation

4 banks = each with 32 2-bit predictor entries



Example of Correlating Branch Predictors

d is assigned to R1

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

Example of Correlating Branch Predictors (Cont.)

initial value of d	d==0?	b1	value of d before b2	d==1?	b2
0	YES	not taken	1	YES	not taken
1	NO	taken	1	YES	not taken
2	NO	taken	2	NO	taken

1-bit predictor initialized to NT

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

All the branches are mispredicted !!!

Example of Correlating Branch Predictors (Cont.)

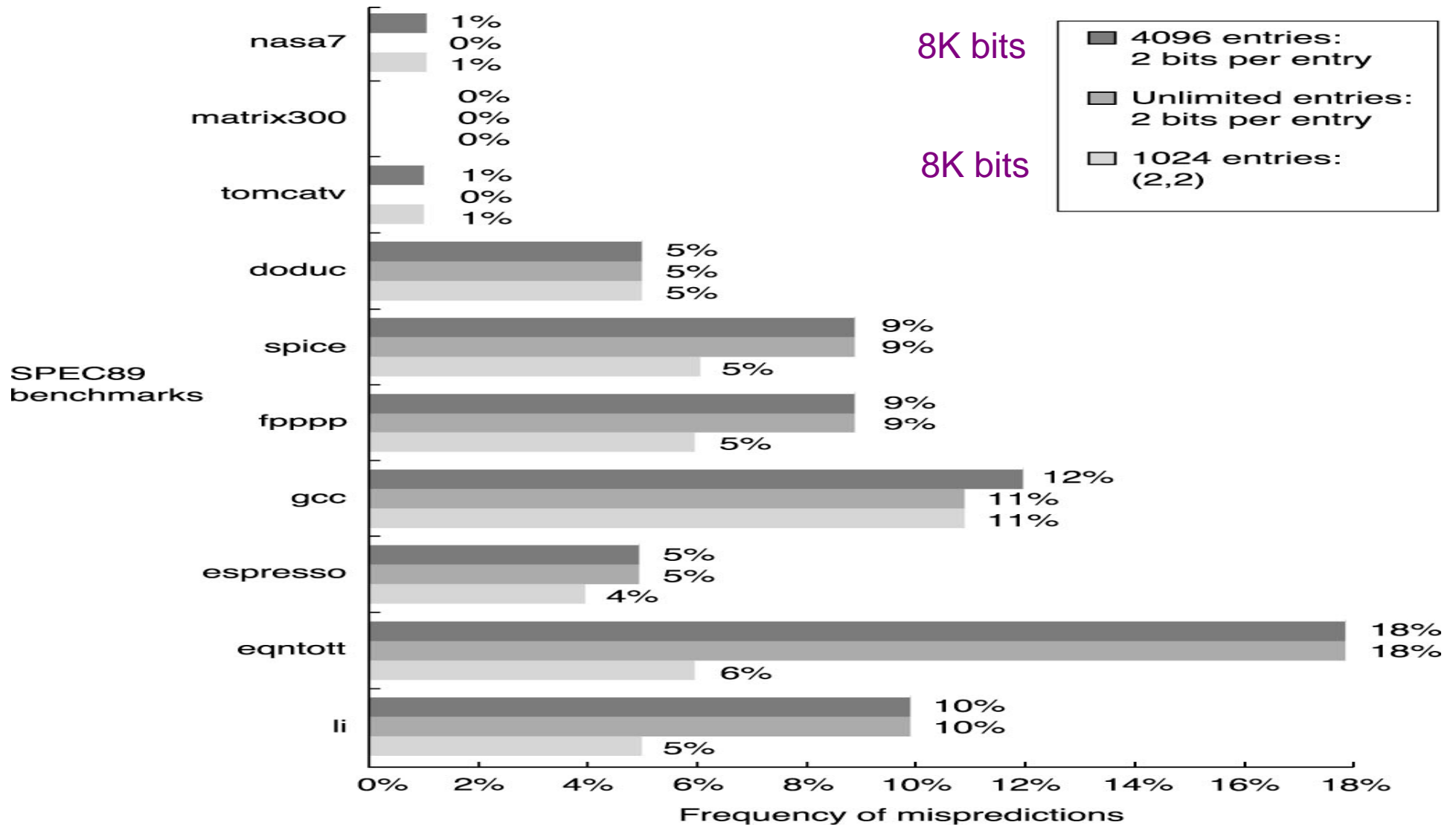
Prediction bits	Prediction if last branch not taken	Prediction if last branch taken
NT/NT	NT	NT
NT/T	NT	T
T/NT	T	NT
T/T	T	T

(1,1) predictor

Use 1-bit correlation + 1-bit prediction with initialized to NT/NT

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

Comparison: Accuracy of Different 2-bit Predictors

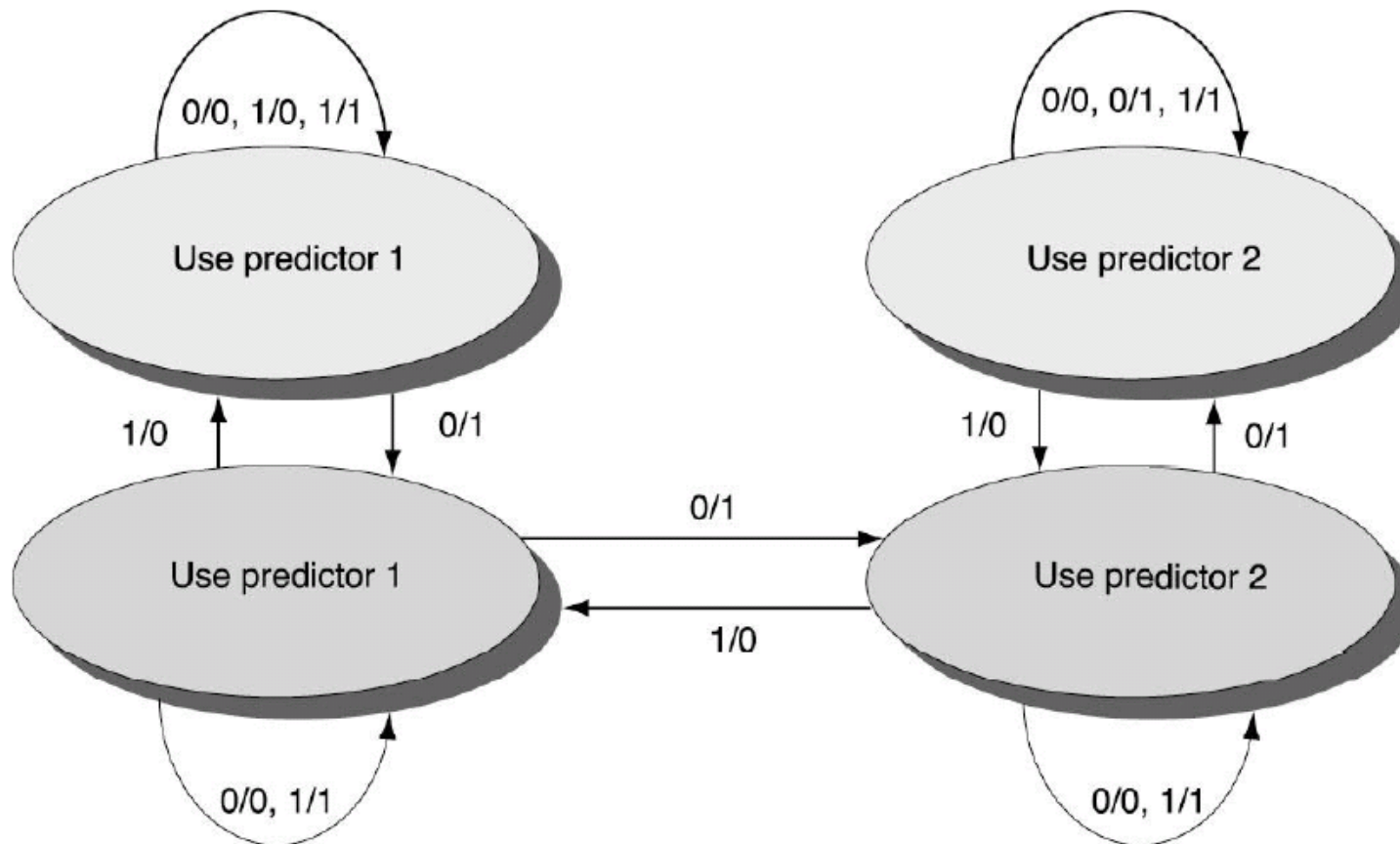


Tournament Predictors

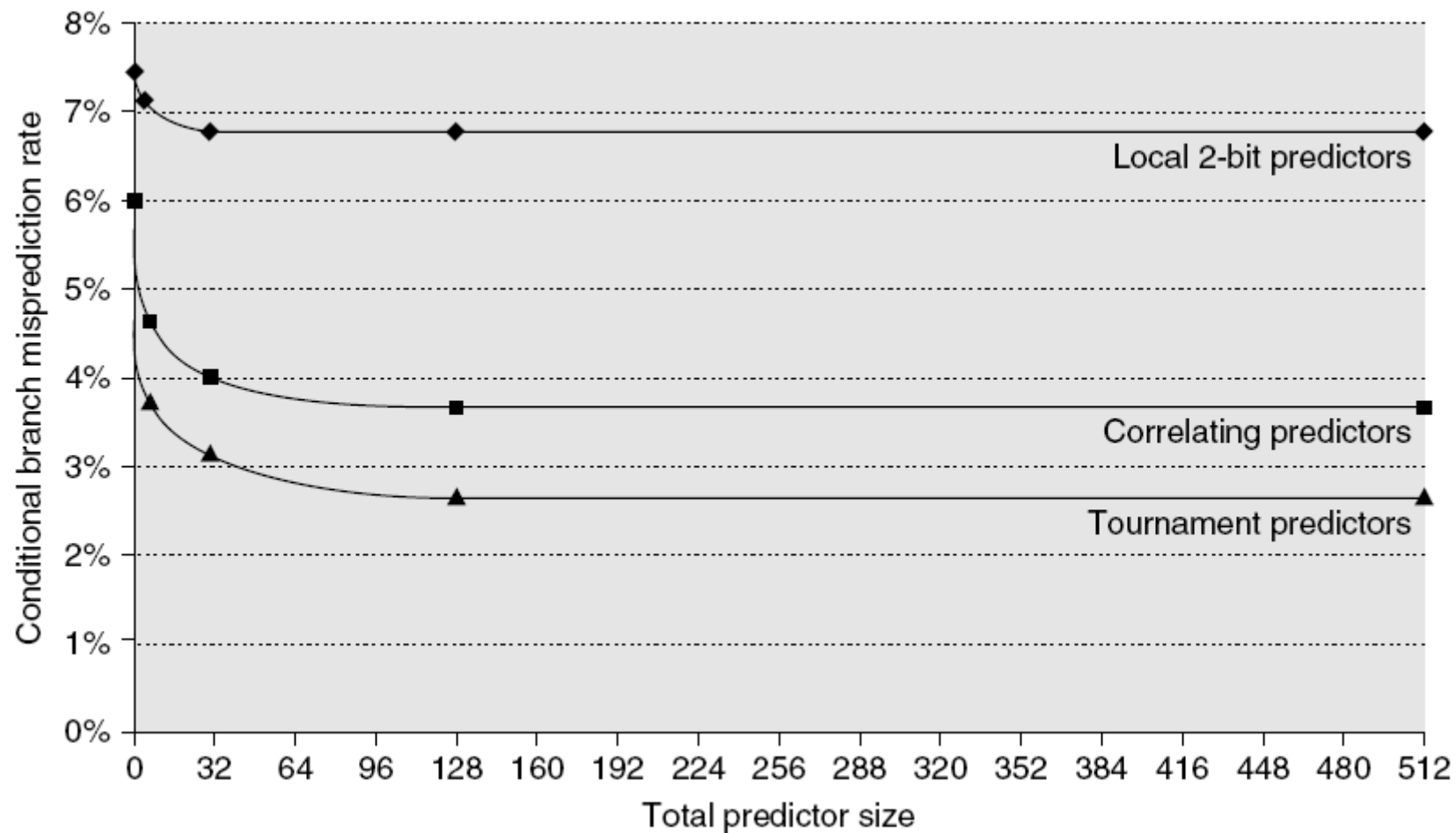
The most popular one

- Recall that the correlator is just a local predictor
- Adaptively combine **local and global** predictors
 - **Multiple predictors**
 - One based on global information: Results of recently executed m branches
 - One based on local information: Results of past executions of the current branch instruction
 - **Selector** to choose which predictors to use
 - E.g.: 2-bit saturating counter, incremented whenever the “predicted” predictor is correct and the other predictor is incorrect, and it is decremented in the reverse situation
- Advantage
 - Ability to select the right predictor for the right branch

State Transition Diagram for A Tournament Predictor



Branch Prediction Performance



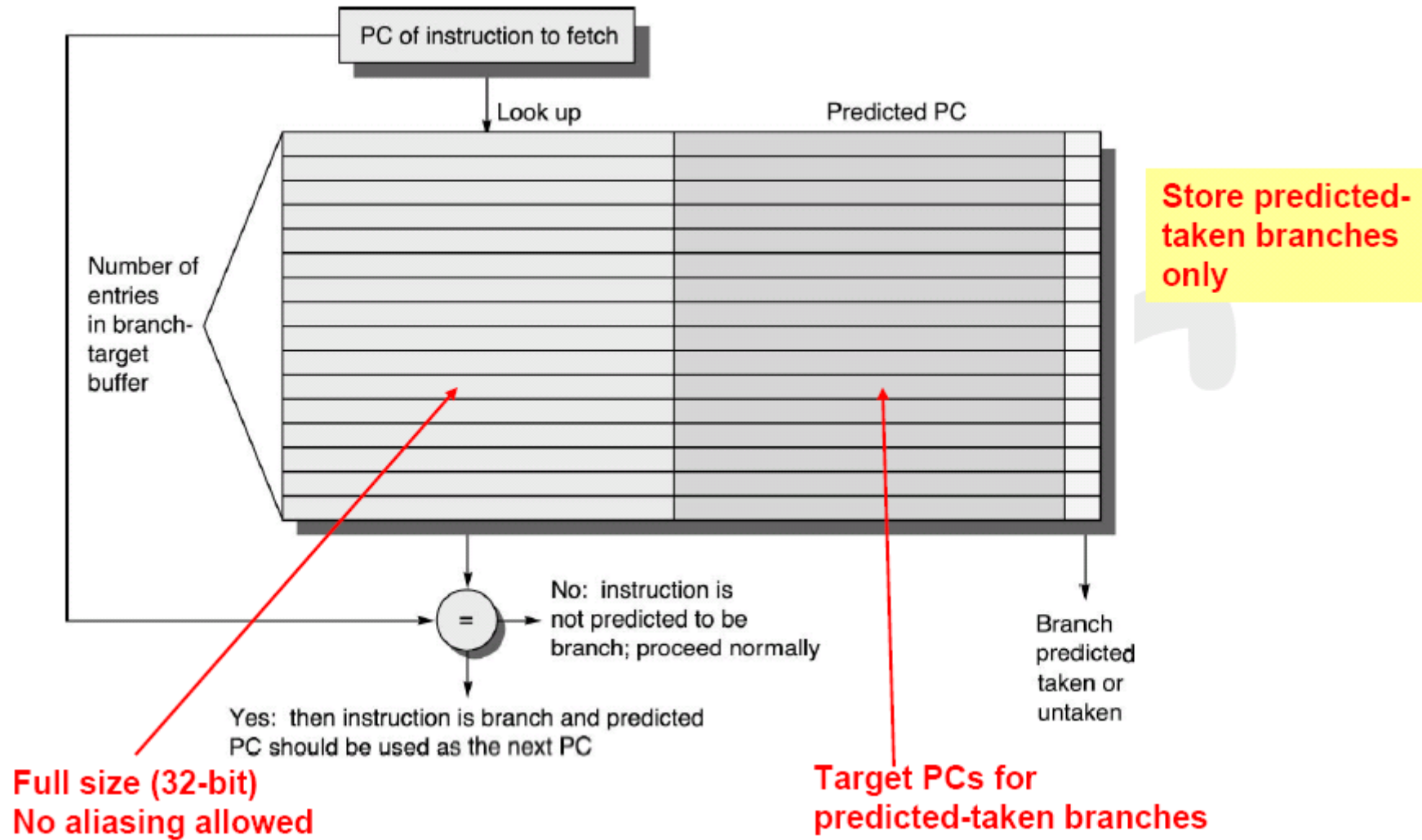
High-Performance Instruction Delivery

- For a multiple issue processor, predicting branches well is not enough
- Deliver a **high-bandwidth instruction stream** is necessary (e.g., 4~8 instructions/cycle)
 - Branch target buffer
 - Integrated instruction fetch unit
 - Indirect branch by predicting return address

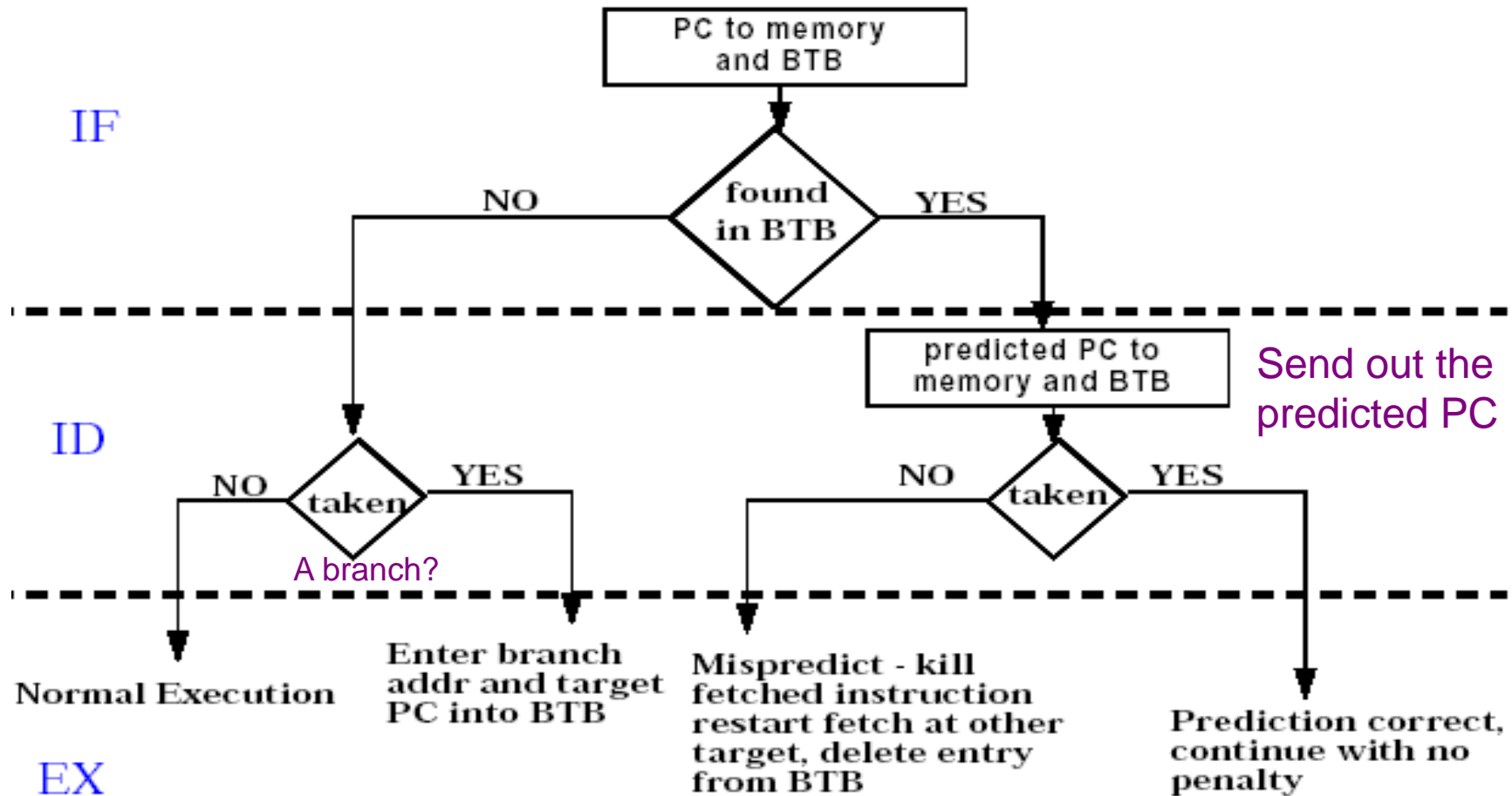
Branch Target Buffer/Cache

- To reduce the branch penalty from 1 cycle to 0
 - Need to know what the address is at the end of IF
 - But the instruction is not even decoded yet
 - So use the instruction address rather than wait for decode
 - If prediction works then penalty goes to 0!
- BTB Idea -- Cache to store taken branches (no need to store untaken)
 - Access the BTB during IF stage
 - Match tag is instruction address → compare with current PC
 - Data field is the predicted PC
- May want to add predictor field
 - To avoid the mispredict twice on every loop phenomenon
 - Adds complexity since we now have to track untaken branches as well

BTB -- Illustration



Flowchart for BTB



Penalties Using this Approach for 5-Stage MIPS

Instruction in buffer	Prediction	Actual Branch	Penalty Cycles
Yes	Taken	Taken	0
Yes	Taken	Not Taken	2
No		Taken	2
No		Not Taken	0

Note:

- Predict_wrong = 1 CC to update BTB + 1 CC to restart fetching
- Not found and taken = 2CC to update BTB

Note:

- For complex pipeline design, the penalties may be higher

Example

- Given prediction accuracy (for inst. in buffer): 90%
- Given hit rate in buffer (for branches predicted token): 90%
- Assume 60% of the branches are taken
- Determine the total branch penalty=?

Solution

- Probability (branch in buffer, but actually not taken) = percent buffer hit rate \times percent incorrect prediction = $90\% \times 10\% = 0.09$
- Probability (branch not in buffer, but actually taken) = 10%
- Hence, we have 2 cycles $\times (0.09+0.1) = 0.38$ cycles

Comparing the delay branch with the penalty = 0.5 cycles/branch

Integrated Instruction Fetch Units

- Consider the fetch unit as a separate autonomous unit, not a pipeline stage
- Functions for the integrated instruction fetch unit
 - Branch prediction
 - Prefetch
 - To deliver multiple instructions per cycle
 - Instruction memory access and buffering
 - may require accessing multiple cache lines
 - prefetch may hide the latency for memory access
 - buffering may be necessary

Return Address Predictor

- **Indirect jump** – jumps whose destination address varies at run time
 - indirect procedure call, select or case, procedure return
 - SPEC89 benchmarks: **85% of indirect jumps are procedure returns**
- **Accuracy of BTB for procedure returns are low**
 - if procedure is called from many places, and the calls from one place are not clustered in time
- **Use a small buffer of return addresses operating as a stack**
 - Cache the most recent return addresses
 - Push a return address at a call, and pop one off at a return
 - If the cache is sufficient large (max call depth) → perfect

Dynamic Branch Prediction Summary

- Branch prediction scheme are limited by
 - Prediction accuracy
 - Mis-prediction penalty
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch
- Tournament predictors take insight to next level, by using multiple predictors
 - usually one based on global information and one based on local information, and combining them with a selector
 - In 2006, tournament predictors using $\approx 30K$ bits are in processors like the Power5 and Pentium 4
- Branch Target Buffer: include branch address & prediction
- Reduce penalty further by fetching instructions from both the predicted and unpredicted direction
 - Require dual-ported memory, interleaved cache \rightarrow HW cost
 - Caching addresses or instructions from multiple path in BTB