Microprocessor Microarchitecture
Branch Prediction

Lynn Choi

School of Electrical Engineering

Branch

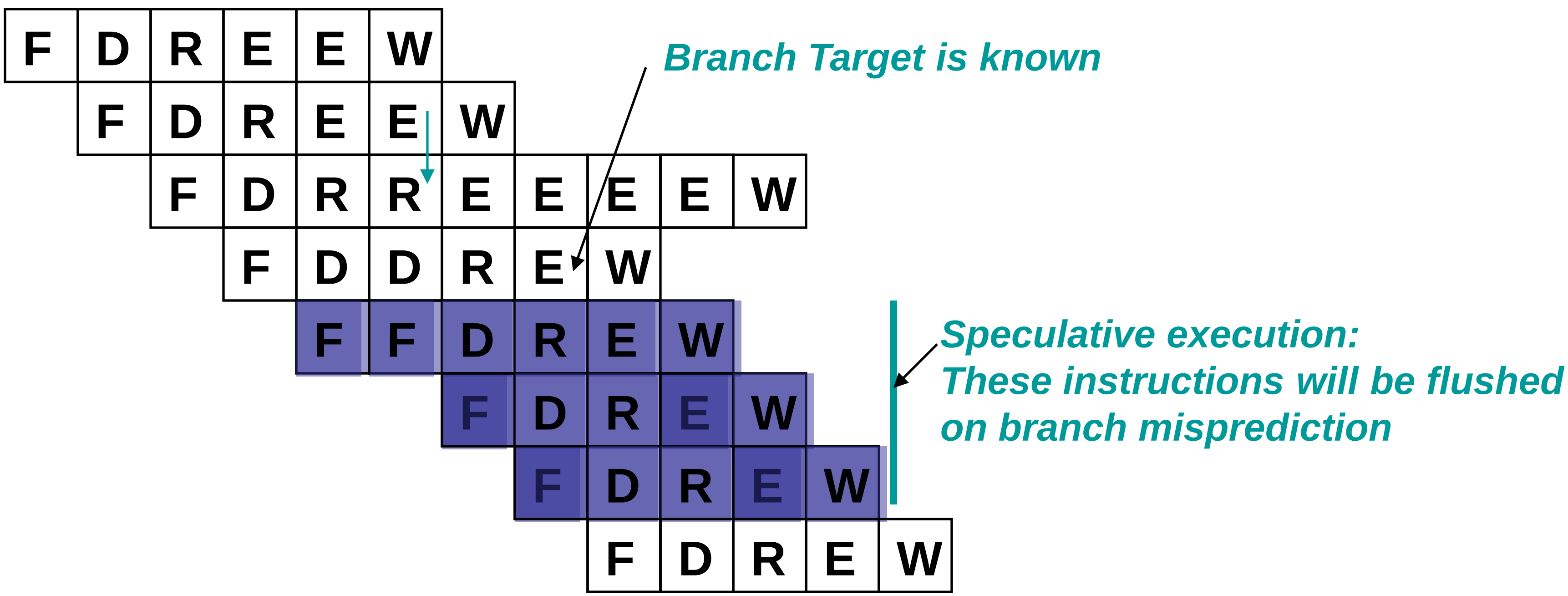
- ***Branch Instruction distribution (% of dynamic instruction count)***
  - 24% of integer SPEC benchmarks
  - 5% of FP SPEC benchmarks
  - Among branch instructions
    - 80% conditional branches
- ***Issues***
  - In early pipelined architecture,
    - Before fetching next instruction,
      - Branch target address has to be calculated
      - Branch condition need to be resolved for conditional branches
    - *Instruction fetch & issue* ***stalls*** until the target address is determined, resulting in ***pipeline bubbles***

Solution

- ***Resolve the branch as early as possible***
- ***Branch Prediction***
  - Predict branch condition & branch target
  - A simple solution
    - PC <- PC + 4: implicitly prefetch the next sequential instruction assuming branch is not taken
  - On a misprediction, the pipeline has to be *flushed*,
    - Example
      - With 10% misprediction rate, 4-issue 5-stage pipeline will waste ~23% of issue slots!
      - With 5% misprediction rate, 13% of issue slots will be wasted.
  - Speculative execution
    - Before branch is resolved, the instructions from the predicted path are fetched and executed
  - We need a more accurate prediction to reduce the misprediction penalty
    - As pipelines become deeper and wider, the importance of branch misprediction will increase substantially!

Branch Misprediction Flush Example

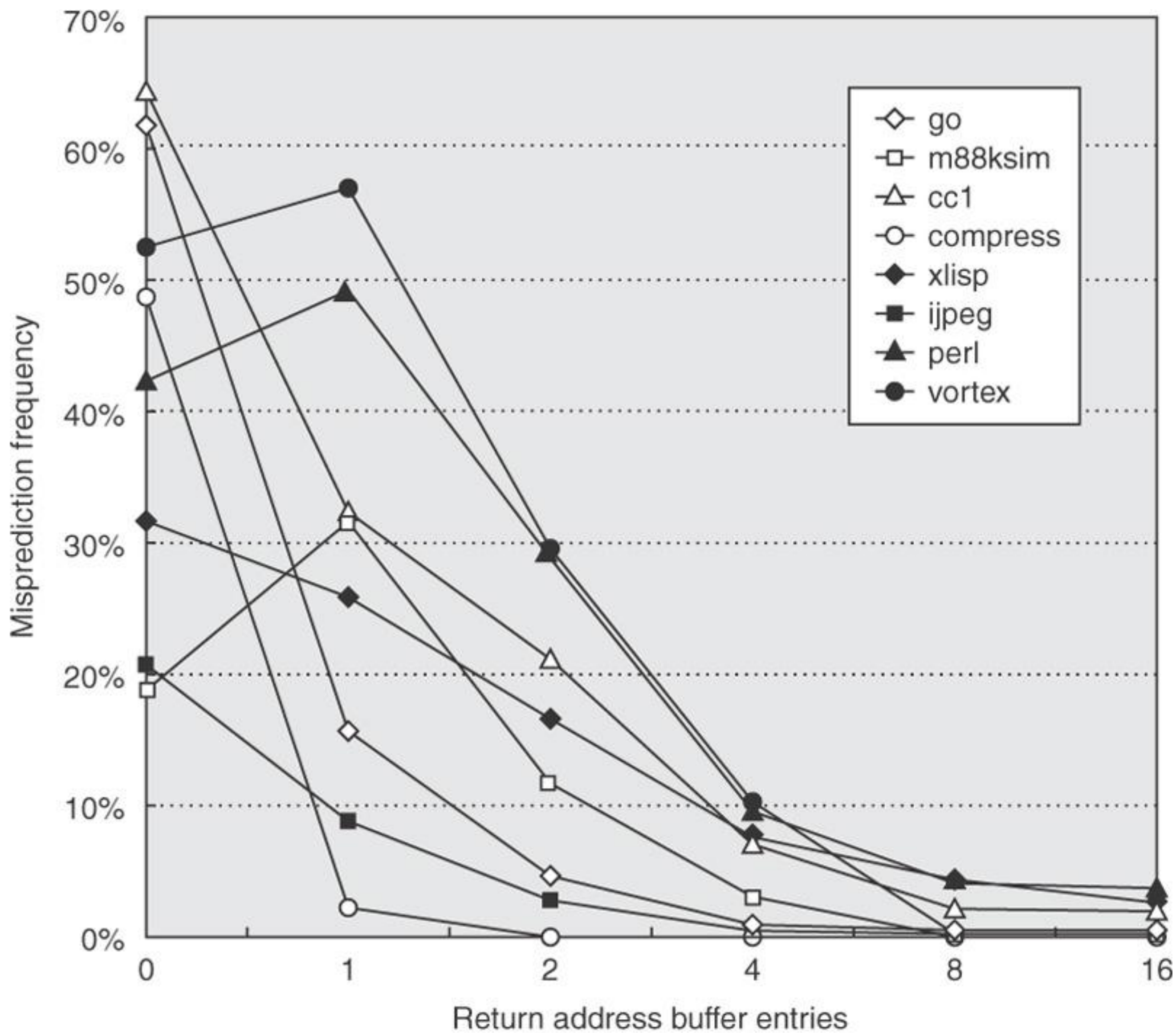
1 LD R1 <- A
2 LD R2 <- B
3 MULT R3, R1, R2
4 BEQ R1, R2, TARGET
5 SUB R3, R1, R4
6 ST A <- R3
7 TARGET: ADD R4, R1, R2



Branch Prediction

- ***Branch condition prediction***
  - For conditional branches
  - Branch Predictor - cache of execution history
  - Predictions are made even before the branch is decoded
- ***Branch target prediction***
  - Branch Target Buffer (BTB)
    - Store target address for each branch
    - Fall-through address is PC +4 for most branches
    - Combined with branch condition prediction (2-bit saturating counter)
  - Target Address Cache
    - Stores target address for only taken branches
    - Separate branch prediction tables
  - Return stack buffer (RSB)
    - Stores return address for procedure call
    - Also called return address buffers (RAB)

RSB Misprediction Rates versus Size



Branch Target Buffer

- ***For BTB to make a correct prediction, we need***
  - BTB hit: the branch instruction should be in the BTB
  - Prediction hit: the prediction should be correct
  - Target match: the target address must not be changed from the last time
- ***Example:*** BTB hit ratio of 96%, 97% prediction hit, 1.2% of target change,

  The overall prediction accuracy = 0.96 * 0.97 *0.988 = 92%
- ***Implementation:*** Accessed with VA and need to be flushed on context switch

Branch Instruction Address	Branch Prediction Statistics	Branch Target Address
. . .	. . .	. . .

Branch Target Buffer

- ***Should we store target address for both taken and not-taken branches?***
- ***How about storing instructions rather than target addresses?***
- ***Branch folding***
  - Store one or more target instructions instead of, or in addition to the predicted target address
  - Advantages
    - On a BTB hit and if the branch is unconditional, the pipeline can substitute the instruction from the BTB in place of the instruction from the cache
    - For highly predictable conditional branches, you can do the same
    - This allows 0-cycle unconditional branches and sometimes 0-cycle conditional branches
    - Or, it allows BTB access to take longer than the time between successive instruction fetches, allowing a larger BTB

Static Branch Prediction

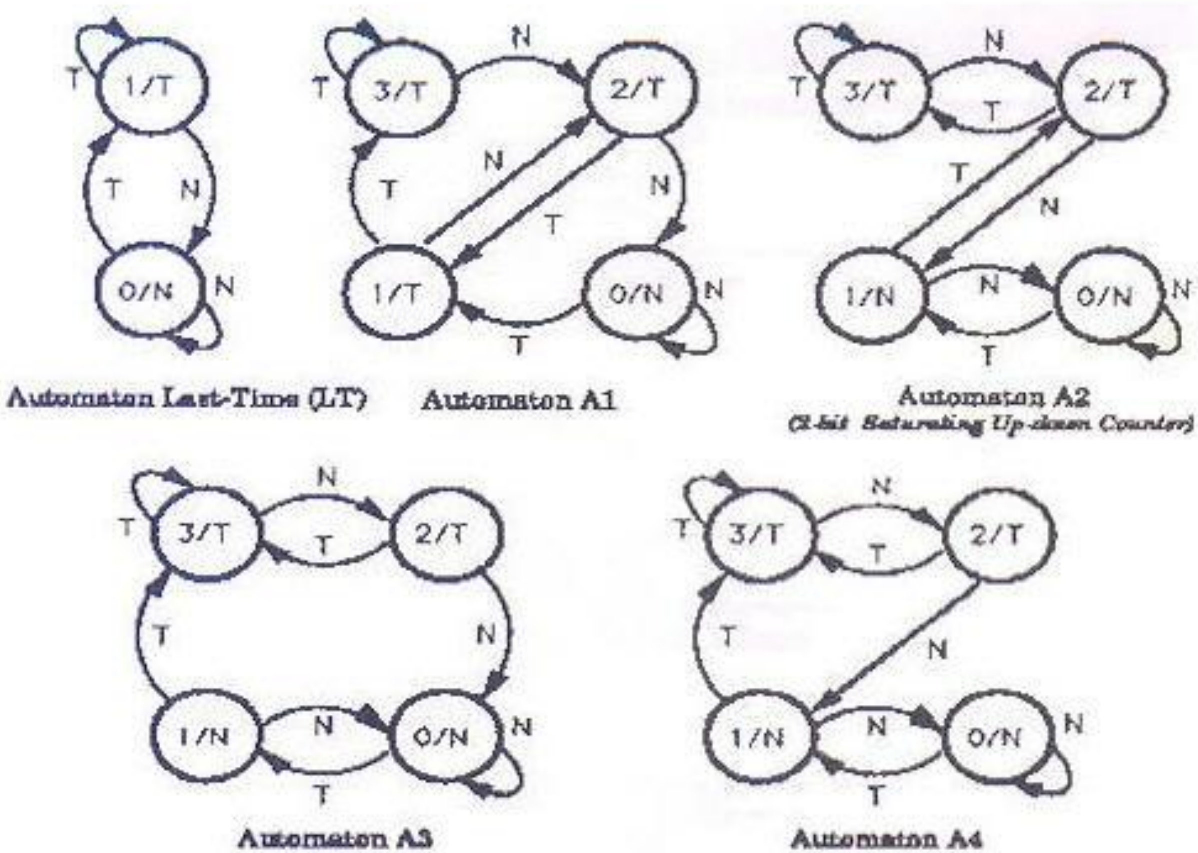
- ***Assume all branches are taken***
  - 60% of conditional branches are taken
- ***Opcode information***
  - Backward Taken and Forward Not-taken scheme
    - Quite effective for loop-bound programs
    - Miss once for all iterations of a loop
    - Does not work for irregular branches
    - 69% prediction hit rate
- ***Profiling***
  - Measure the tendencies of the branches and preset a static prediction bit in the opcode
  - Sample data sets may have different branch tendencies than the actual data sets
  - 92.5% hit rate
- ***Static predictions are used as safety nets when the dynamic prediction structures need to be warmed up***

Dynamic Branch Prediction

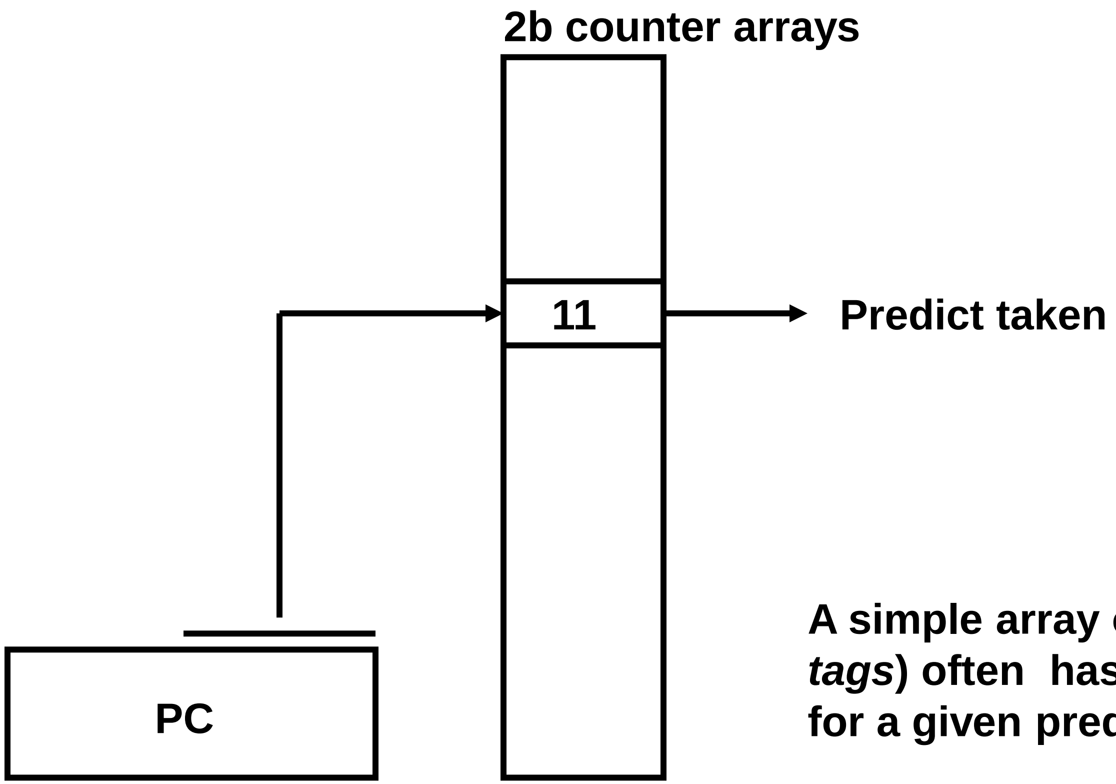
- ***Dynamic schemes- use runtime execution history***
  - LT (last-time) prediction - 1 bit, 89%
  - Bimodal predictors - 2 bit
    - 2-bit saturating up-down counters (Jim Smith), 93%
    - Several different state transition implementations
    - Branch Target Buffer(BTB)
  - Static training scheme (A. J. Smith), 92 ~ 96%
    - Use both profiling and runtime execution history
      - Statistics collected from a pre-run of the program
      - A history pattern consisting of the last n runtime execution results of the branch
  - Two-level adaptive training (Yeh & Patt), 97%
    - First level, *branch history register* (BHR)
    - Second level, *pattern history table* (PHT)

Bimodal Predictor

S(I): State at time I
G(S(I)) -> T/F: Prediction decision function
E(S(I), T/N) -> S(I+1): State transition function
Performance: A2 (usually best), A3, A4 followed by A1 followed by LT



Bimodal Predictor Structure

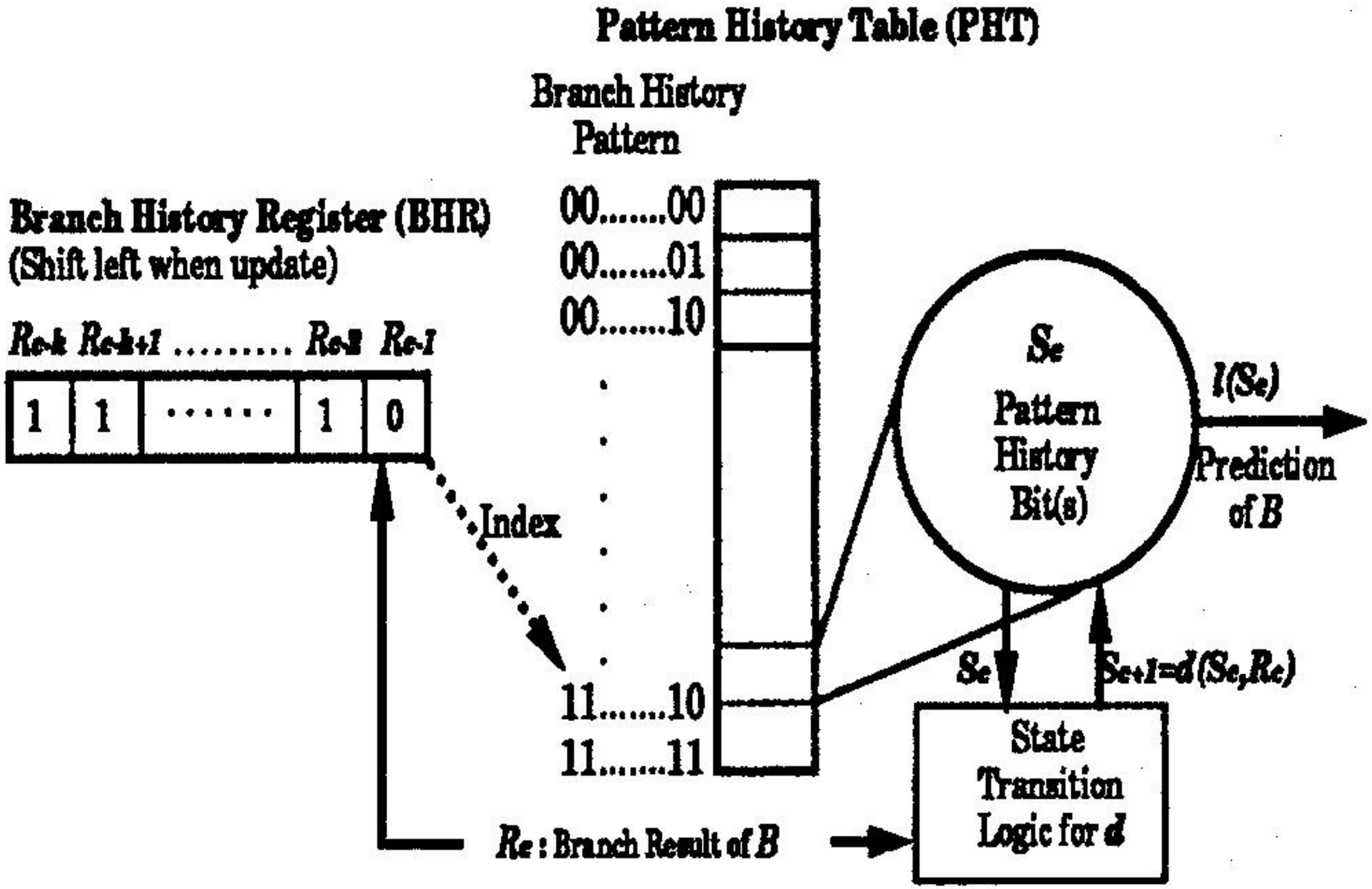


A simple array of counters (*without tags*) often has better performance for a given predictor size

Two-level adaptive predictor

- ***Motivated by***
  - Two-bit saturating up-down counter of BTB (J. Smith)
  - Static training scheme (A. Smith)
    - Profiling + history pattern of last k occurences of a branch
- ***Organization***
  - Branch history register (BHR) table
    - Indexed by instruction address (B_i)
    - Branch history of last k branches
      - Local predictor: The last k occurrences of the same branch ($R_{i,c-k}R_{i,c-k+1}\ldots R_{i,c-1}$)
      - Global predictor: The last k branches encountered
    - Implemented by k-bit shift register
  - Pattern history table (PT)
    - Indexed by a history pattern of last k branches
    - Prediction function $z = \lambda(S_c)$
      - Prediction is based on the branch behavior for the last s occurrences of the pattern
    - State transition function $S_{c+1} = \delta(S_c, R_{i,c})$
      - 2b saturating up-down counter

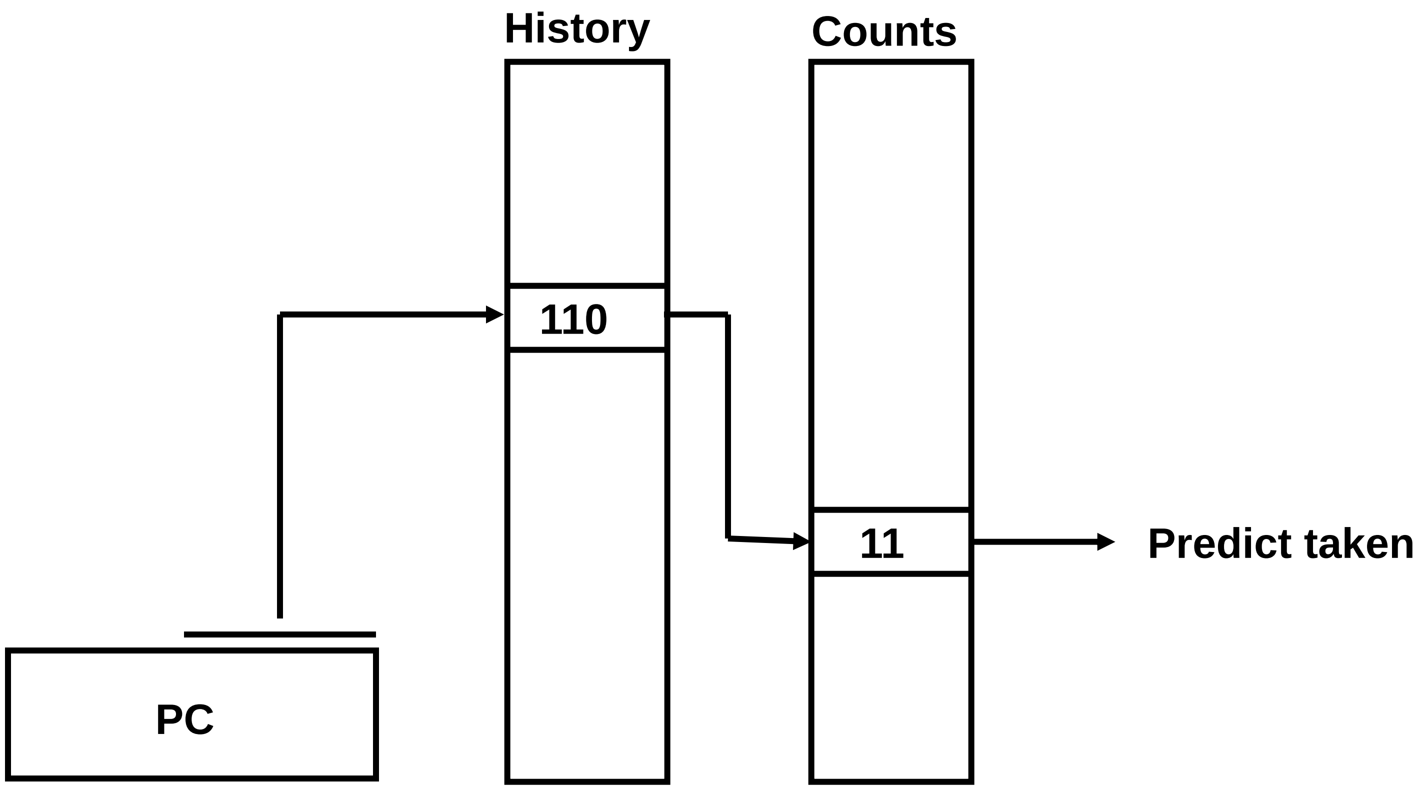
Structure of 2-level adaptive predictor



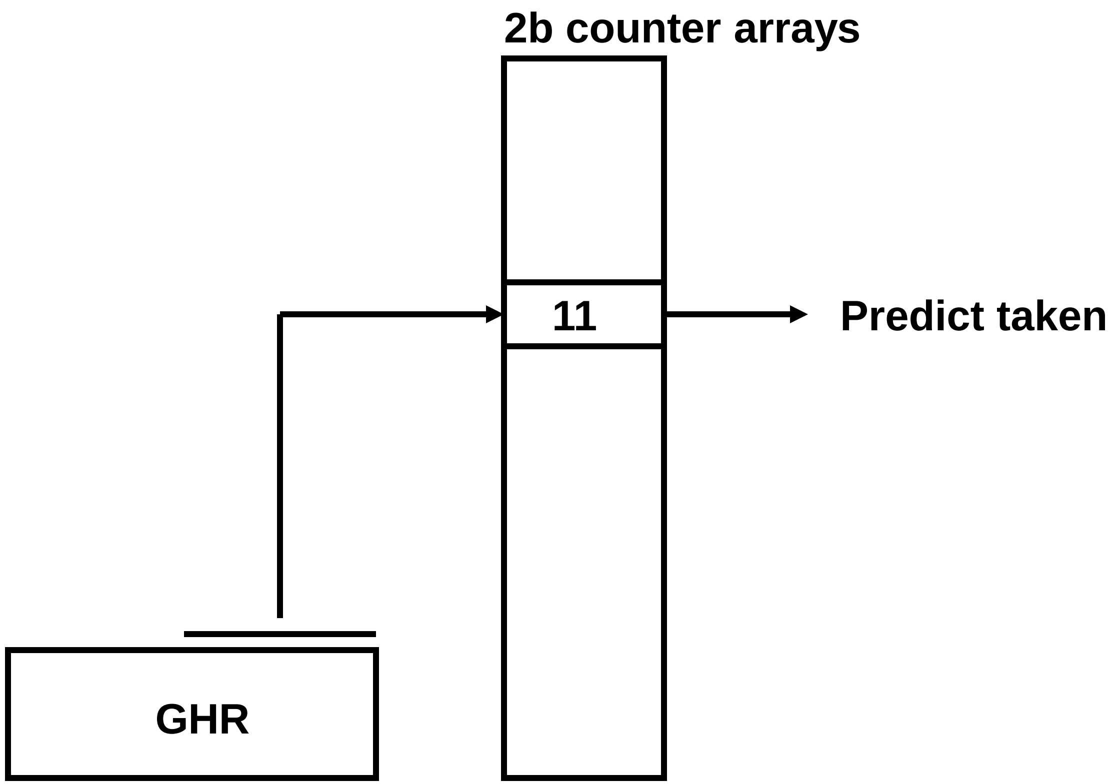
Global vs. Local History

- *Global history schemes*
  - The last k conditional branches encountered
  - Works well when the direction taken by sequentially executed branches is highly correlated
    - EX) if (x >1) then .. If (x<=1) then ..
  - These are also called *correlating predictors*
- *Local history schemes*
  - The last k occurrences of the same branch
  - Works well for branches with simple repetitive patterns
  - Two types of contention
    - Branch history may reflect a mix of histories of all the branches that map to the same history entry
    - With 3 bits of history, cannot distinguish patterns of 0110 and 1110
      - However, if the first pattern is executed many times then followed by the second pattern many times, the counters can dynamically adjust

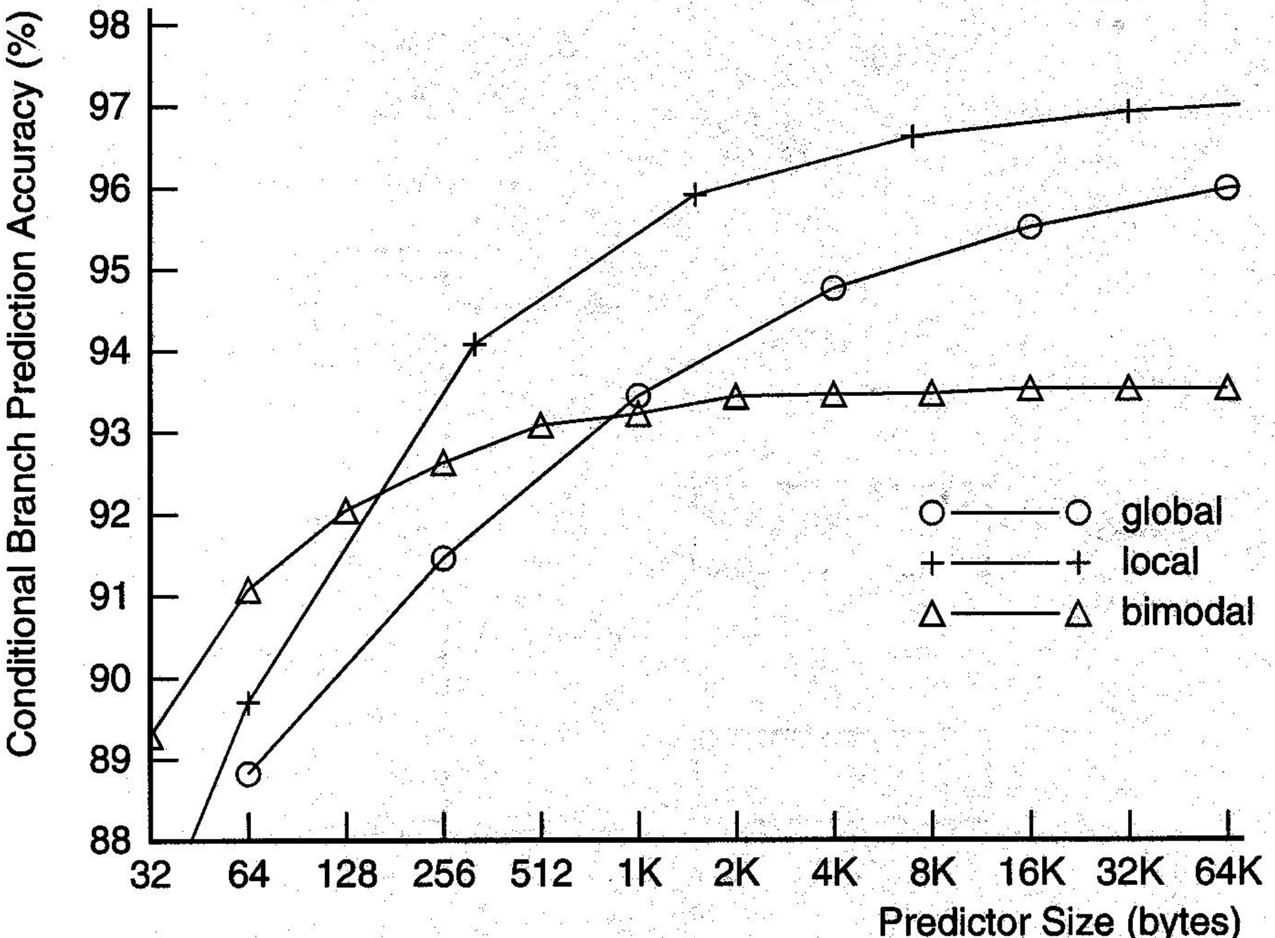
Local History Structure



Global History Structure



Global/Local/Bimodal Performance



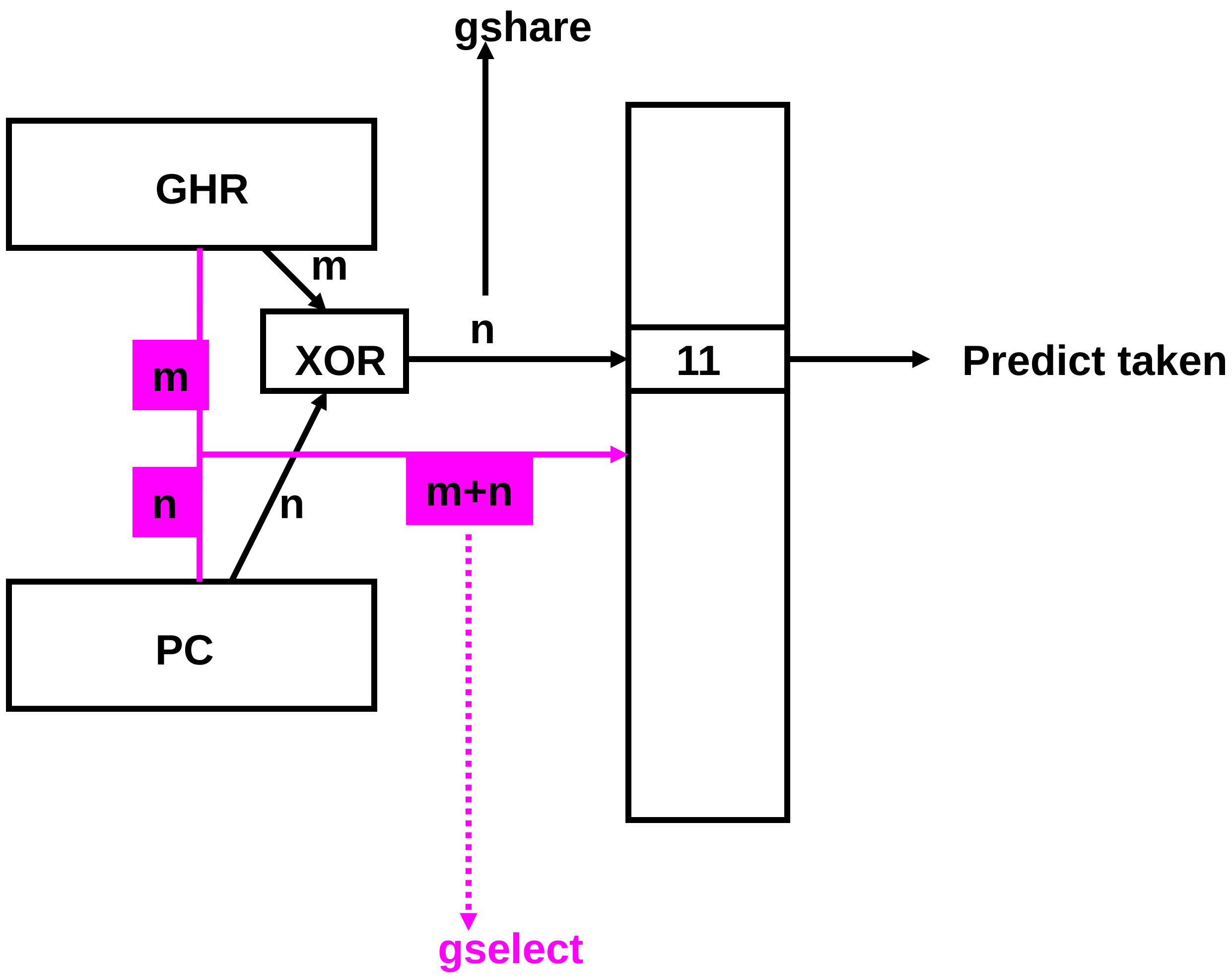
Global Predictors with Index Sharing

- ***Global predictor with index selection (gselect)***
  - Counter array is indexed with a concatenation of global history and branch address bits
  - For small sizes, gselect parallels bimodal prediction
  - Once there are enough address bits to identify most branches, more global history bits can be used, resulting in much better performance than global predictor
- ***Global predictor with index sharing (gshare)***
  - Counter array is indexed with a hashing (XOR) of the branch address and global history
    - Eliminate redundancy in the counter index used by gselect

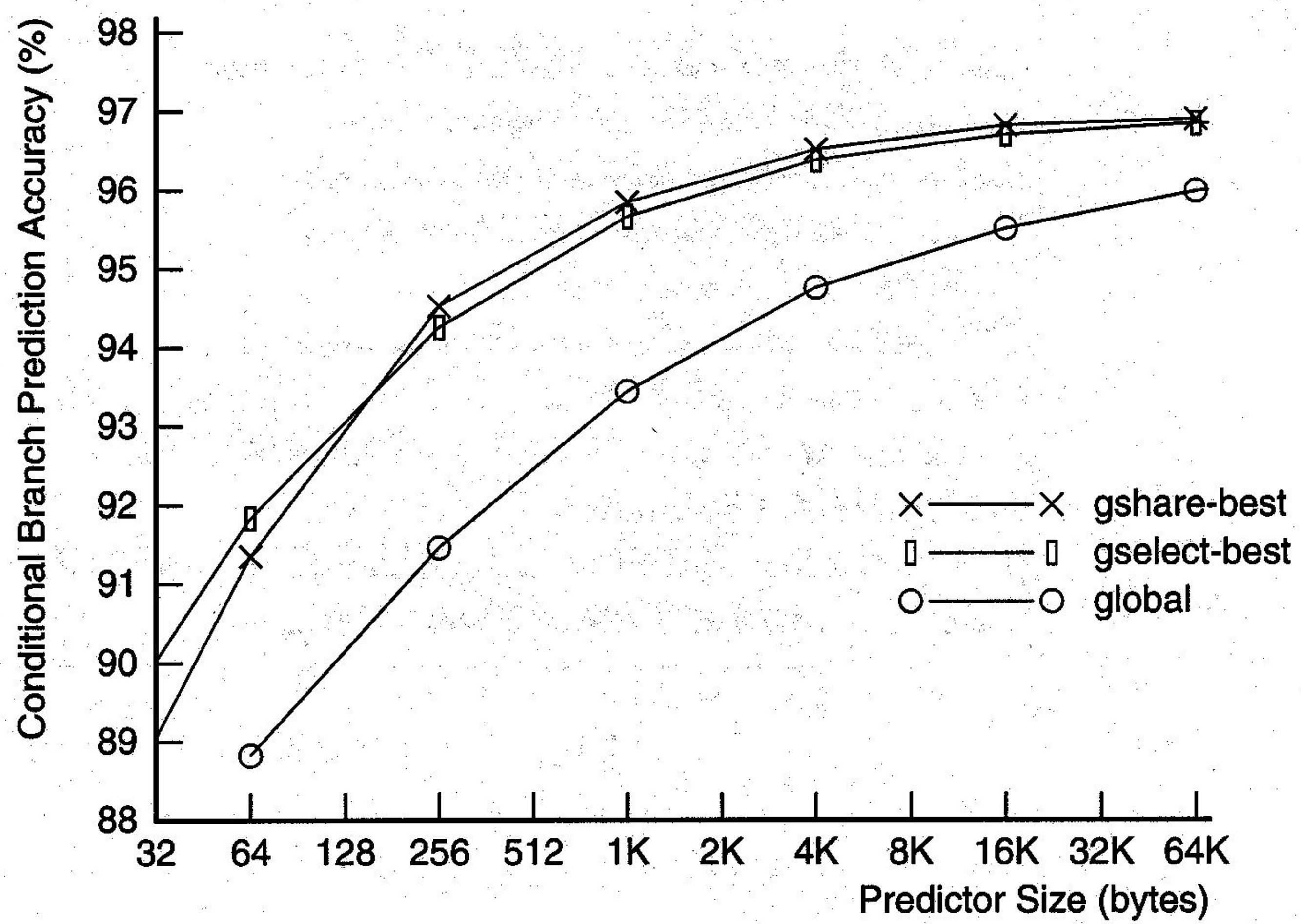
Gshare vs. Gselect

Branch Address	Global History	Gselect 4/4	Gshare 8/8
00000000	00000001	00000001	00000001
00000000	00000000	00000000	00000000
11111111	00000000	11110000	11111111
11111111	10000000	11110000	01111111

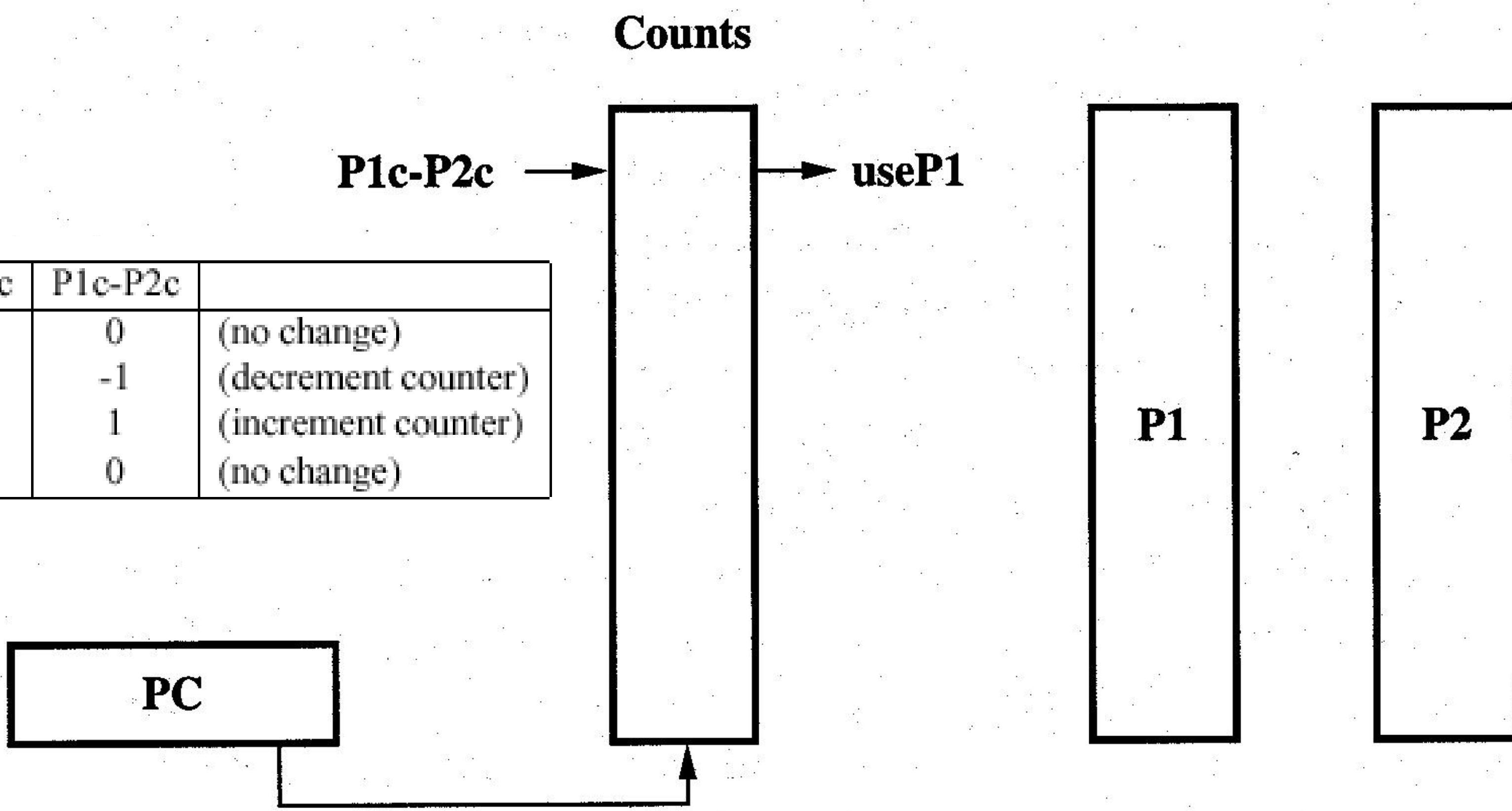
Gshare/Gselect Structure



Global History with Index Sharing Performance



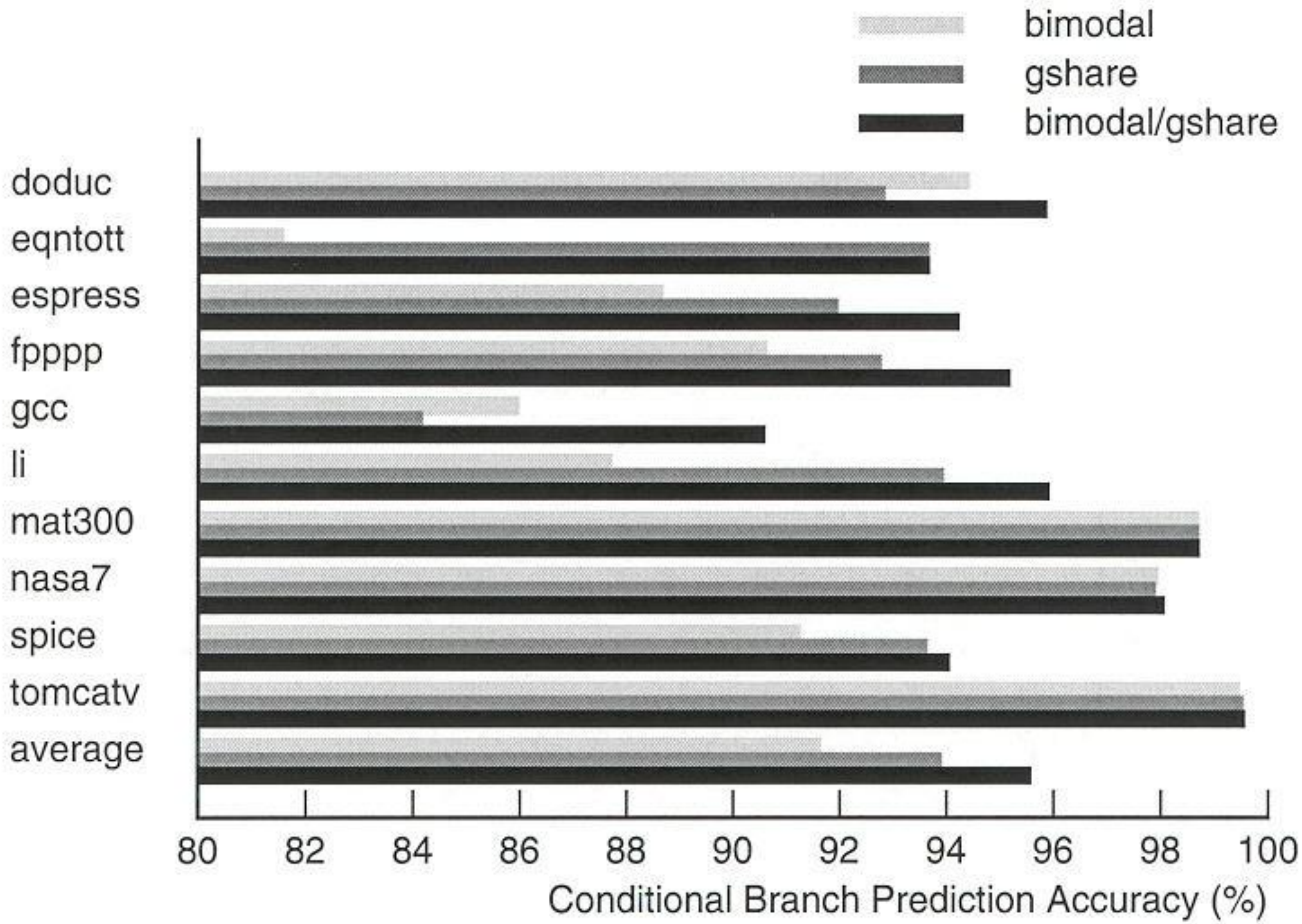
Combined Predictor Structure



P1c	P2c	P1c-P2c	
0	0	0	(no change)
0	1	-1	(decrement counter)
1	0	1	(increment counter)
1	1	0	(no change)

- ***These are also called tournament predictors***
  - Adaptively combine global and local predictors

Combined Predictor Performance



Exercises and Discussion

- ***Intel's Xscale processor uses bimodal predictor? What state would you initialize?***
- ***Y/N Questions. Explain why.***
  - Branch prediction is more important for FP applications. (Y/N) Why or Why not?
  - Branch prediction is more difficult for conditional branches than indirect branches. (Y/N) Why or Why not?
  - To predict branch targets, an instruction must be decoded first. (Y/N) Why or Why not?
  - RSB stores target address of call instructions. (Y/N) Why or Why not?
  - At the beginning of program execution, static branch prediction is more effective than dynamic branch prediction (Y/N) Why or Why not?