15-740/18-740
Computer Architecture
Lecture 24: Control Flow

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Midterm II
  - November 22

- Project Poster Session
  - December 10 (tentative)

Last 3 Lectures: Superscalar Processing

- Fetch (supply N instructions)
- Decode (generate control signals for N instructions)
- Rename (detect dependencies between N instructions)
- Dispatch (determine readiness and select N instructions to execute in-order or out-of-order)
- Execute (have enough functional units to execute N instructions + forwarding paths to forward results of N instructions)
- Write into Register File (have enough ports to write results of N instructions)
- Retire (N instructions)

Last Lecture

- Dependency check logic
- Renaming
- Wakeup, selection, data forwarding (bypass)
- Retirement and resource deallocation

- Reducing complexity
  - Block structured ISA
  - Clustering

Readings

- Required:
  - McFarling, "Combining Branch Predictors," DEC WRL TR, 1993.
  - Carmean and Sprangle, "Increasing Processor Performance by Implementing Deeper Pipelines," ISCA 2002.

- Recommended:
  - Evers et al., "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work," ISCA 1998.
  - Yeh and Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," ISCA 1992.
  - Jouppi and Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," ASPLOS 1989.
  - Kim et al., "Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths," MICRO 2006.
  - Jimenez and Lin, "Dynamic Branch Prediction with Perceptrons," HPCA 2001.

The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions

- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
  - N cycles: (minimum) branch resolution latency
  - Stalling on a branch wastes instruction processing bandwidth (i.e. reduces IPC)
    - N x IW instruction slots are wasted

- How do we keep the pipeline full after a branch?
- Problem: Need to determine the **next fetch address** when the branch is fetched (to avoid a pipeline bubble)

The Branch Problem

- Assume a 5-wide superscalar pipeline with 20-cycle branch resolution latency

- How long does it take to fetch 500 instructions?
  - Assume no fetch breaks and 1 out of 5 instructions is a branch
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy
    - 100 (correct path) + 20 (wrong path) = 120 cycles
    - 20% extra instructions fetched
  - 98% accuracy
    - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
    - 40% extra instructions fetched
  - 95% accuracy
    - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
    - 100% extra instructions fetched

Branch Types

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Different branch types can be handled differently

Approaches to Conditional Branch Handling

- Branch prediction
  - Static
  - Dynamic

- Eliminating branches

  I. Predicated execution
    - Static
    - Dynamic
    - HW/SW Cooperative

  II. Predicate combining (and condition registers)

- Multi-path execution
- Delayed branching (branch delay slot)
- Fine-grained multithreading

Predicate Combining

- Complex predicates are converted into multiple branches
  - if ((a == b) && (c < d) && (a > 5000)) { … }
    - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction
  - Predicates stored and operated on using condition registers
  - A single branch checks the value of the combined predicate

+ Fewer branches in code → fewer mipredictions/stalls

-- Possibly unnecessary work

-- If the first predicate is false, no need to compute other predicates

- Condition registers exist in IBM RS6000 and the POWER architecture

Delayed Branching (I)

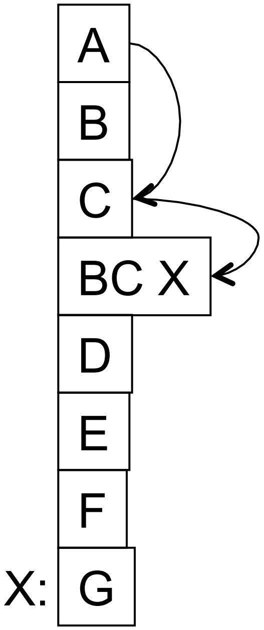
- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles
- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.

- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions

- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot

Delayed Branching (II)

Normal code:

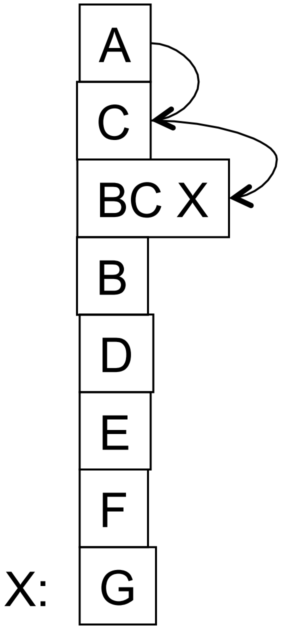


Timeline:

F	E
A	
B	A
C	B
BC	C
--	BC
G	--

6 cycles

Delayed branch code:



Timeline:

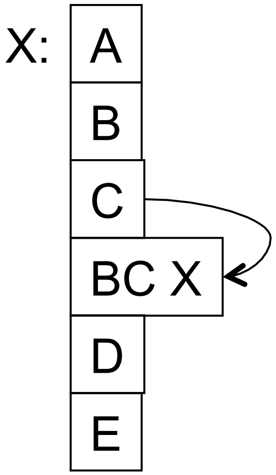
F	E
A	
C	A
BC	C
B	BC
G	B

5 cycles

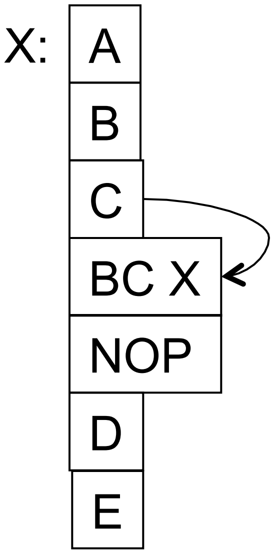
Fancy Delayed Branching (III)

- Delayed branch with squashing
  - In SPARC
  - If the branch falls through (not taken), the delay slot instruction is not executed
  - Why could this help?

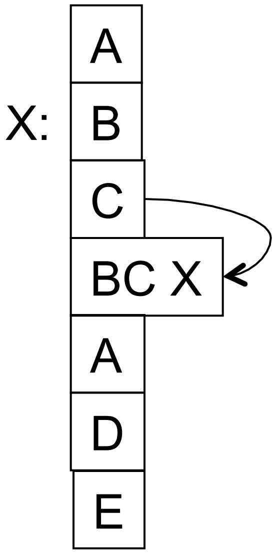
Normal code:



Delayed branch code:



Delayed branch w/ squashing:



Delayed Branching (IV)

- Advantages:

+ Keeps the pipeline full with useful instructions assuming

1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves

2. All delay slots can be filled with useful instructions

- Disadvantages:

-- Not easy to fill the delay slots (even with a 2-stage pipeline)

1. Number of delay slots increases with pipeline depth, issue width, instruction window size.

2. Number of delay slots should be variable with OoO execution. Why?

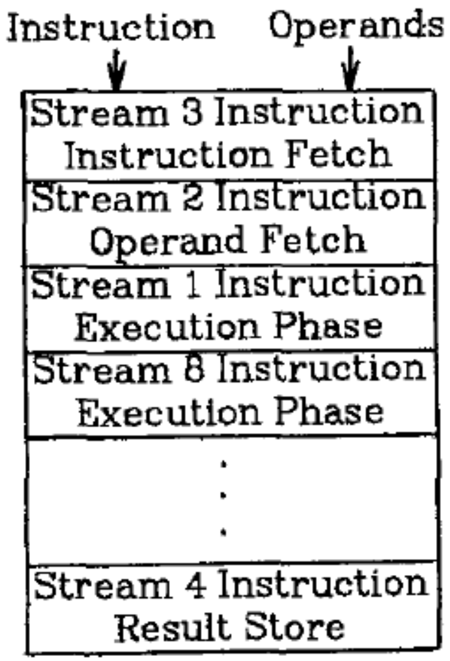
-- Ties ISA semantics to implementation

-- SPARC, MIPS, HP-PA: 1 delay slot

-- What if pipeline implementation changes with the next design?

Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts. Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch resolves, there is no need to fetch another instruction from the same thread
  - Branch resolution latency overlapped with execution of other threads' instructions



+ No logic needed for branch prediction, (also for dependency checking)

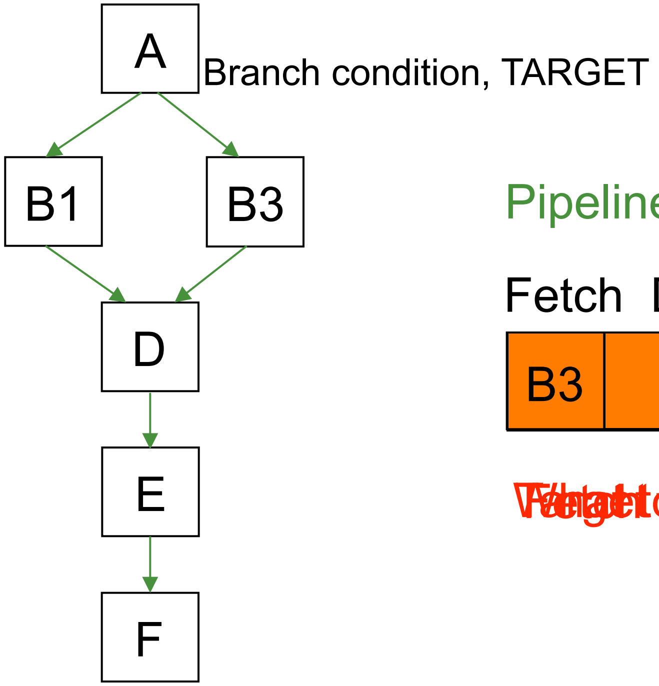
-- Single thread performance suffers

-- Does not overlap latency if not enough threads to cover the whole pipeline

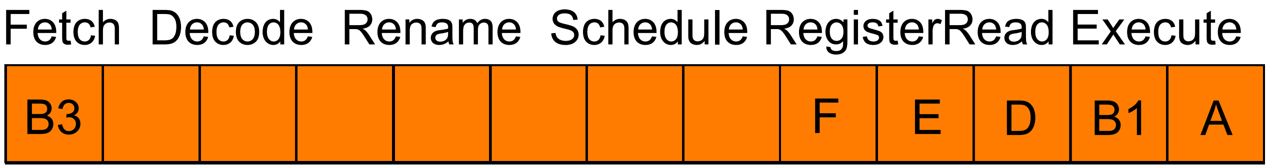
-- Extra logic for keeping thread contexts

Branch Prediction

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch



Pipeline



What to fetch next?

Target Misprediction Detected! Flush the pipeline

Fetch from the correct target

Verify the Prediction

Branch Prediction

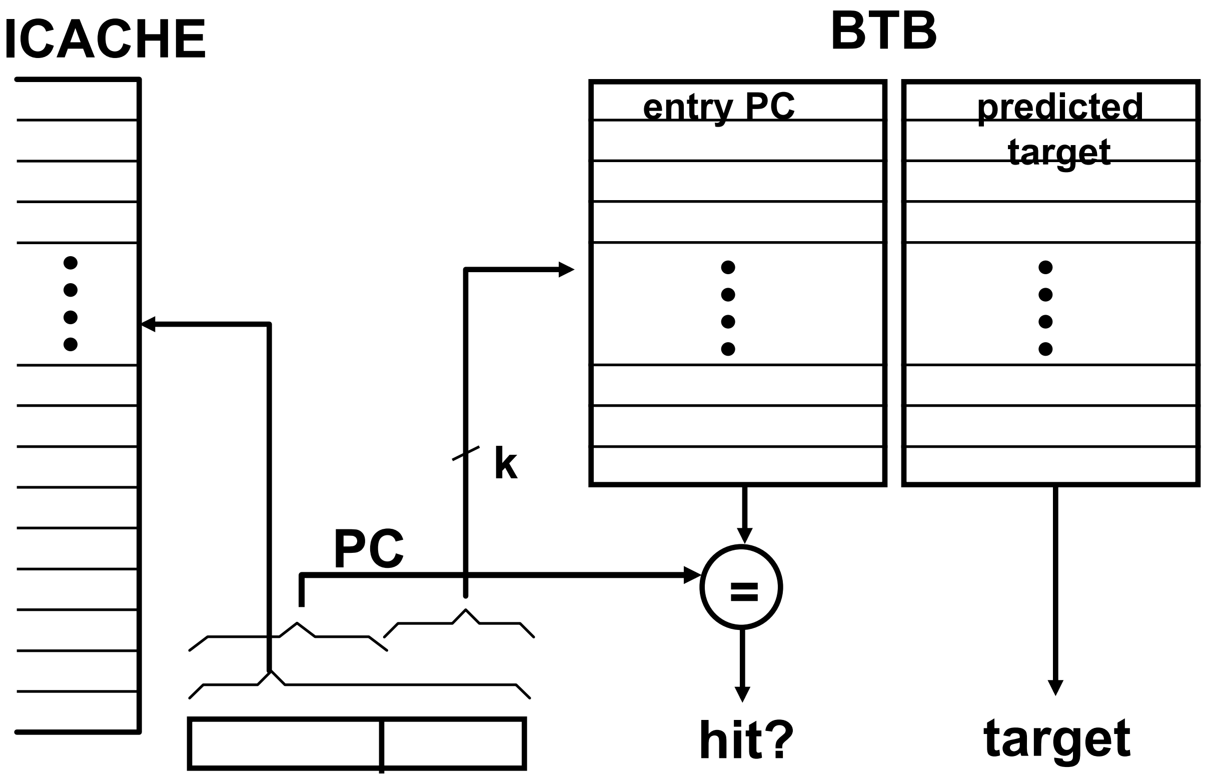
- Idea: Predict the next fetch address (to be used in the next cycle) when the branch is fetched

- Requires three things to be predicted:
  - Whether the fetched instruction is a branch
  - Conditional branch direction
  - Branch target address (if taken)

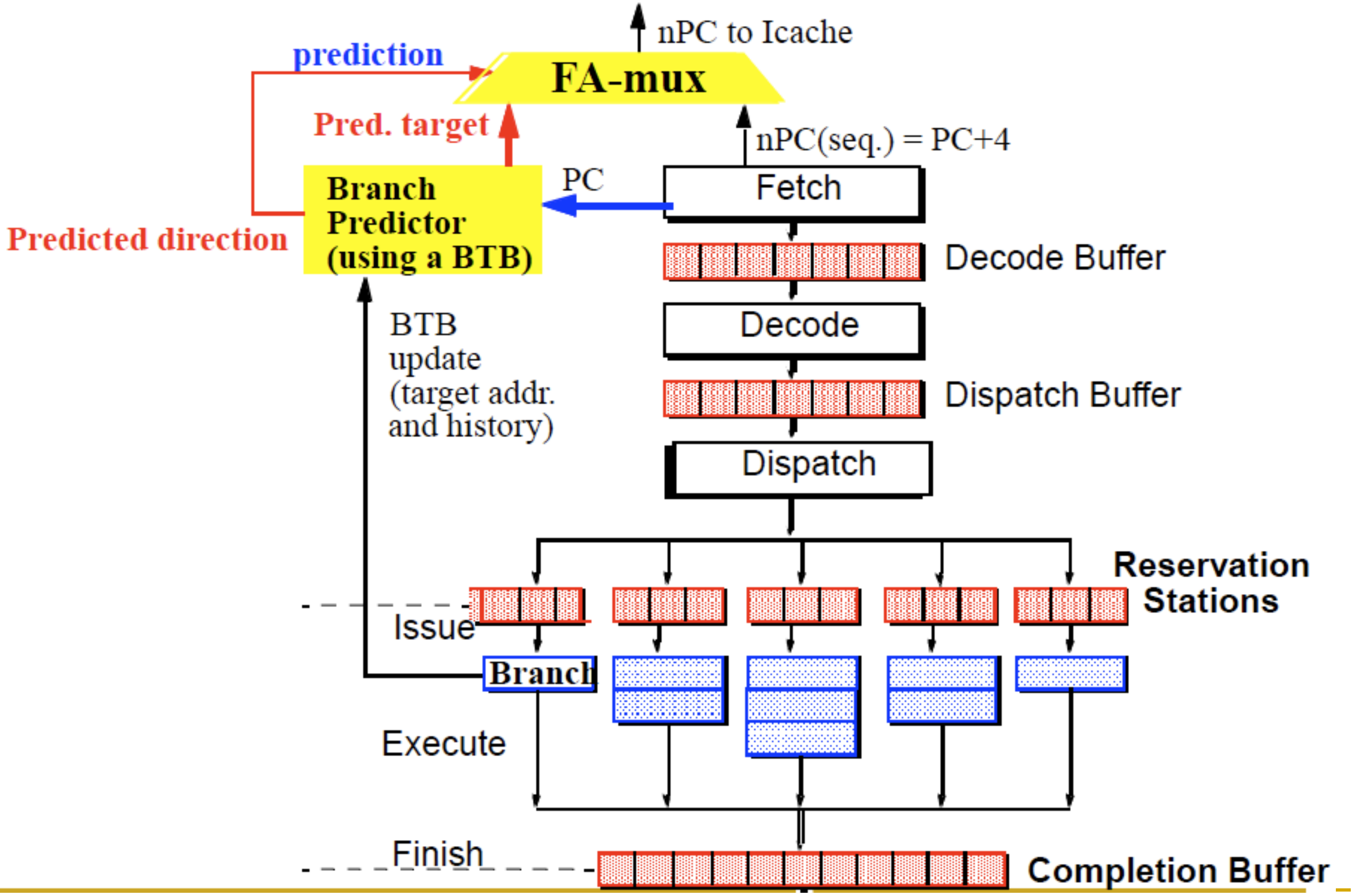
- Target addresses remain the same for conditional direct branches across dynamic instances
  - Idea: Cache the target address from previous instance
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

Branch Target Buffer

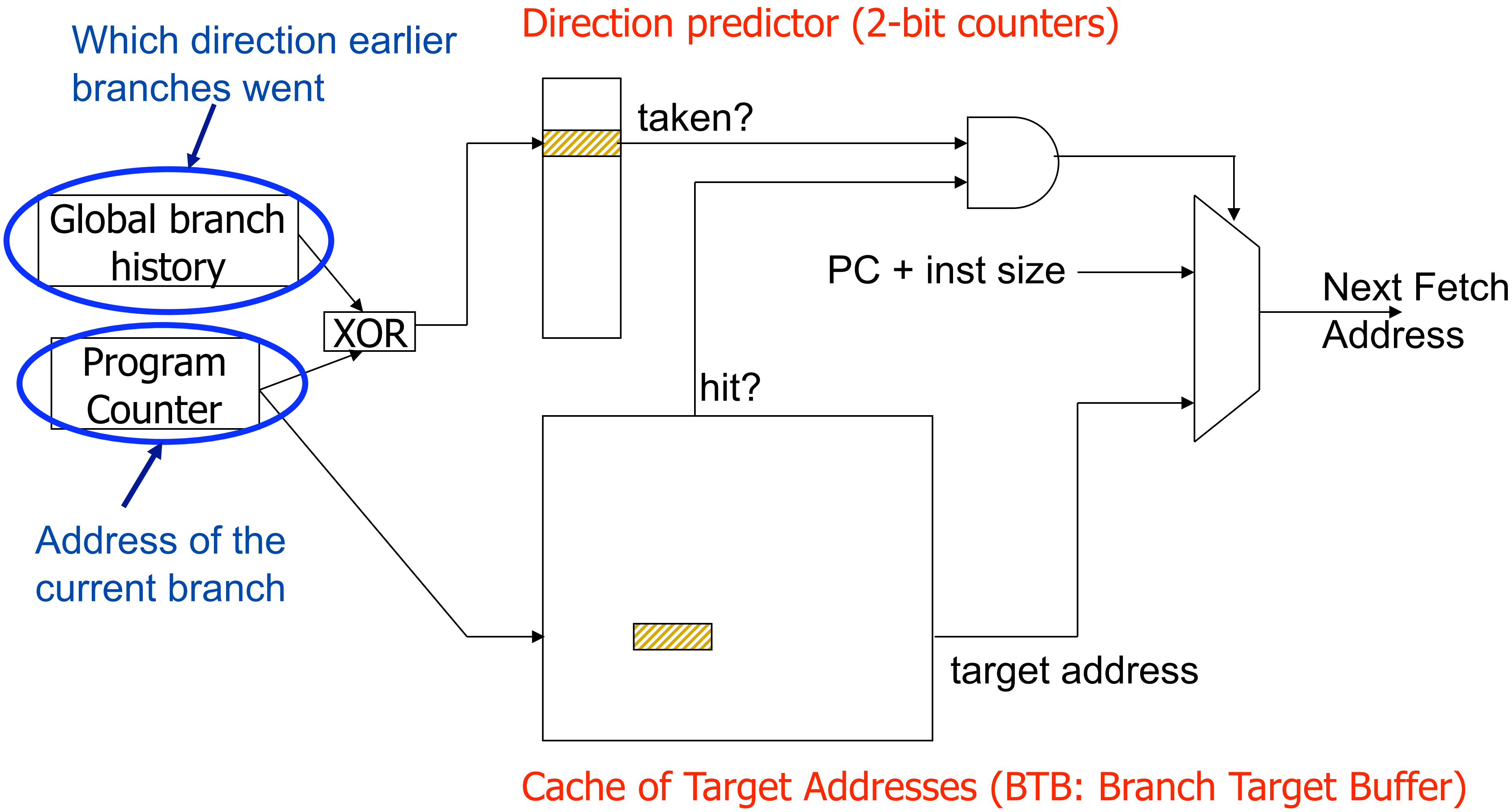
- Cache of branch target addresses accessed in parallel with the I-cache in the fetch stage
- Updated only by taken branches
- If BTB hit and the instruction is a predicted-taken branch
  - target from the BTB (assuming hit) is used as fetch address in the next cycle
- If BTB miss or the instruction is a predicted-not-taken branch
  - PC+N is used as the next fetch address in the next cycle



Branch Target Buffer in Fetch Stage



A Frontend with BTB and Direction Prediction


Which direction earlier branches went
Direction predictor (2-bit counters)
taken?
Global branch history
XOR
Program Counter
PC + inst size
Next Fetch Address
hit?
Address of the current branch
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time (single-bit)
  - Two-bit counter based
  - Two-level (global vs. local)
  - Hybrid

Static Branch Prediction (I)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~40%
  - Compiler can layout code such that the likely path is the "not-taken" path: Good for wide fetch as well!

- Always taken
  - No direction prediction
  - Better accuracy: ~60%
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC

- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

Static Branch Prediction (II)

- Profile-based
  - Idea: Compiler determines likely direction for each branch using profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction (more accurate than schemes in previous slide)

-- Requires hint bits in the branch instruction format

-- Accuracy depends on dynamic branch behavior:

TTTTTTTTTTNNNNNNNNNN → 50% accuracy
TNTNTNTNTNTNTNTNTNTN → 50% accuracy

-- Accuracy depends on the representativeness of profile input set

Static Branch Prediction (III)

- Program-based
  - Idea: Use heuristics based on program analysis to determine statically-predicted direction
  - Opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
  - Loop heuristic: Predict a branch guarding a loop execution as taken (i.e., execute the loop)
  - Pointer and FP comparisons: Predict not equal

+ Does not require profiling

-- Heuristics might be not representative or good

-- Requires ISA support

- Ball and Larus, ”Branch prediction for free,” PLDI 1993.
  - 20% misprediction rate

Dynamic Branch Prediction

- Idea: Predict branches based on dynamic information (collected at run-time)

- Advantages

  + No need for profiling: input set representativeness problem goes away

  + Prediction based on history of the execution of branches

    + It can adapt to dynamic changes in branch behavior

- Disadvantages

  -- More complex (requires additional hardware)

Last Time Predictor

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed

    TTTTTTTTTTNNNNNNNNNN → 90% accuracy

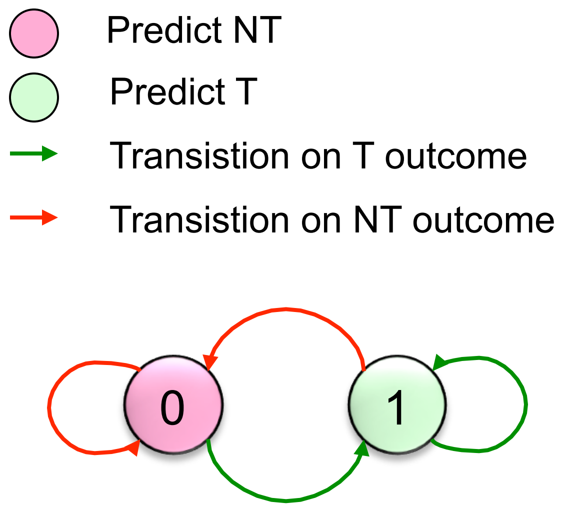
- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

+ Loop branches for loops with large number of iterations

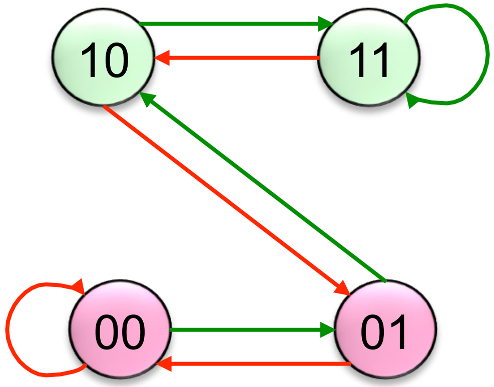
-- Loop branches for loops will small number of iterations

TNTNTNTNTNTNTNTNTNTN → 0% accuracy

Two-Bit Counter Based Prediction



Finite State Machine for Last-time Predictor



Finite State machine for 2BC (**2**-**B**it **C**ounter)

- Counter using saturating arithmetic
  - There is a symbol for maximum and minimum values

Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

- Accuracy for a loop with N iterations = (N-1)/N

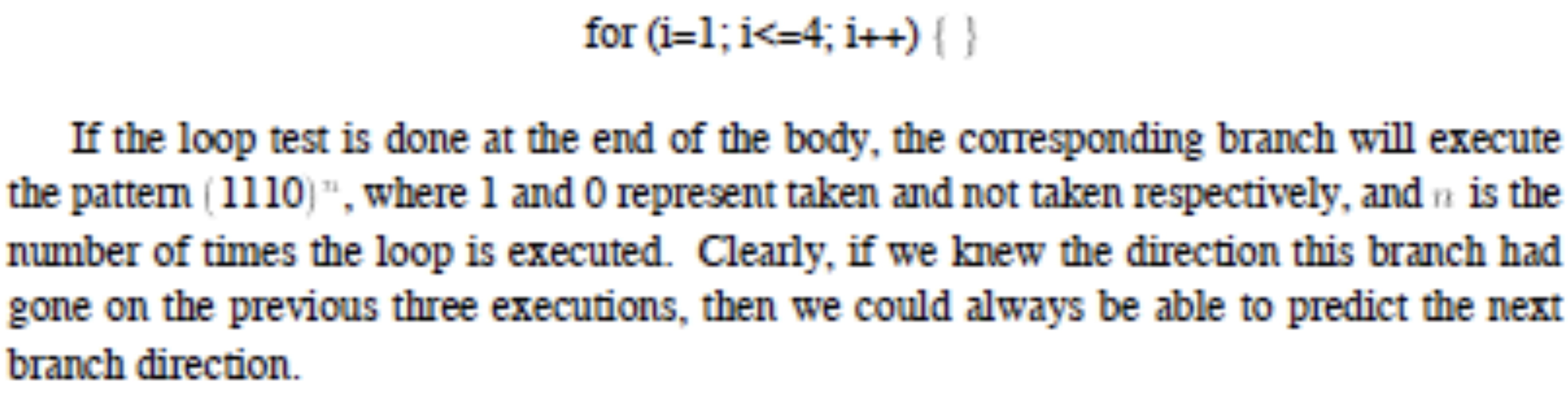
  TNTNTNTNTNTNTNTNTNTN → 50% accuracy

  (assuming init to weakly taken)

+ Better prediction accuracy

-- More hardware cost (but counter can be part of a BTB entry)

Can We Do Better?



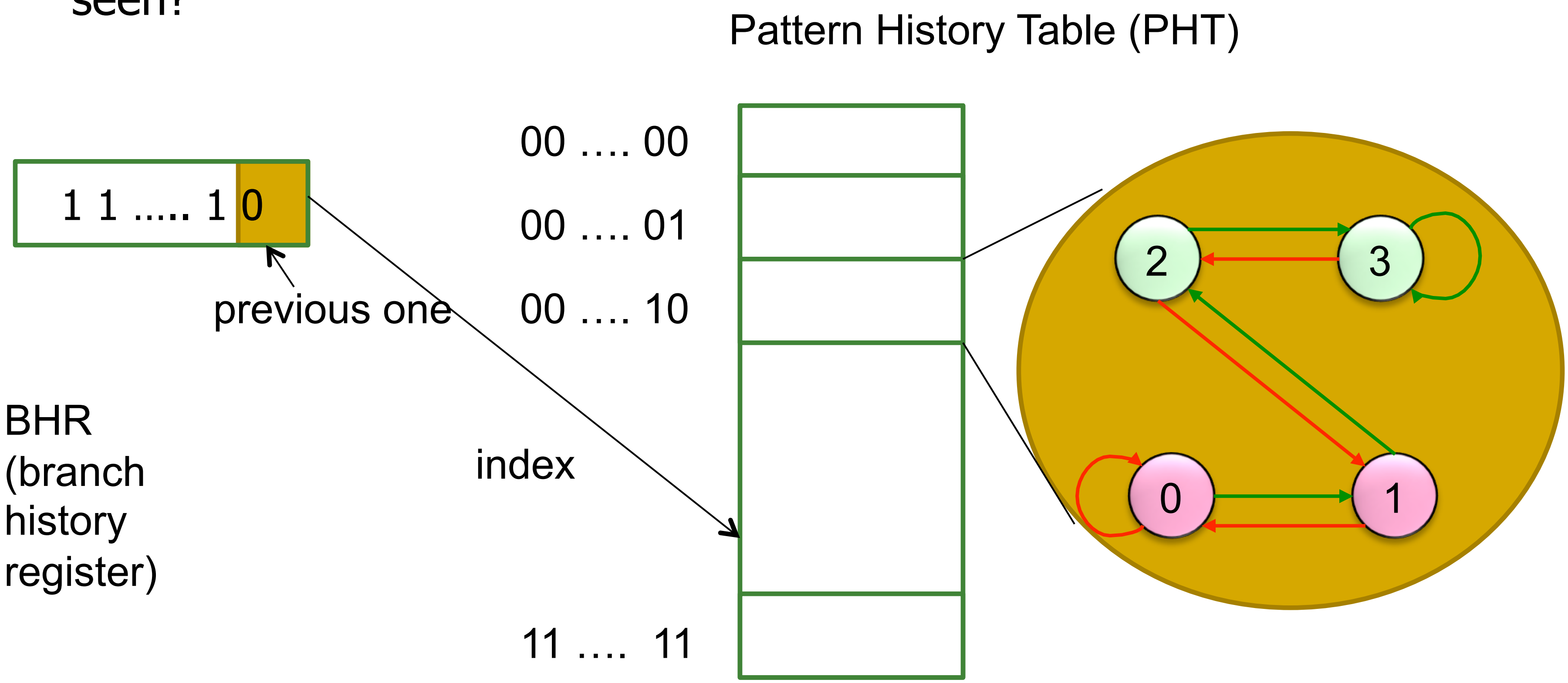
for (i=1; i<=4; i++) { }

If the loop test is done at the end of the body, the corresponding branch will execute the pattern $(1110)^n$, where 1 and 0 represent taken and not taken respectively, and n is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

- McFarling, "Combining Branch Predictors," DEC WRL TR 1993.

Two Level Branch Predictors

- First level: Branch history register (N bits)
  - The direction of last N branches
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen?



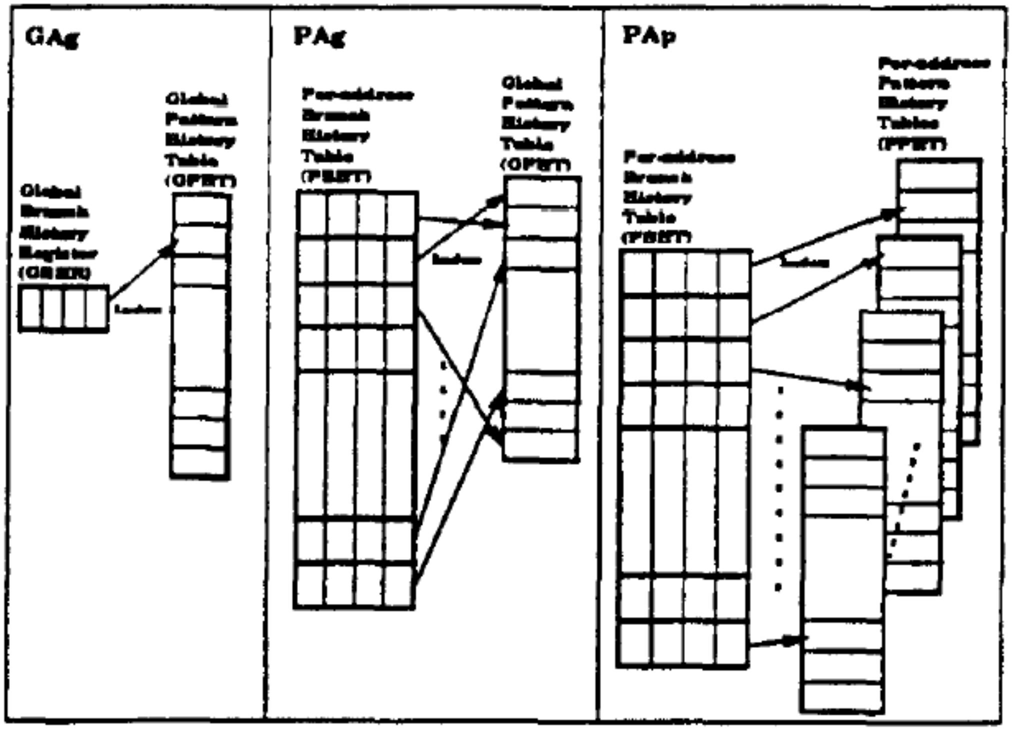
Prediction and Update Functions

- Prediction
  - Pattern History Table accessed at fetch time to generate a prediction
  - Top bit of the 2-bit counter determines predicted direction

- Update
  - Pattern History Table accessed when the branch is retired to update the counters that generated the prediction
  - If branch
    - actually taken: increment the counter
    - actually not-taken: decrement the counter

Two-Level Predictor Variations

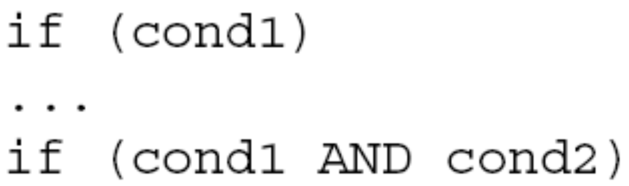
- BHR can be global (G), per set of branches (S), or per branch (P)
- PHT counters can be adaptive (A) or static (S)
- PHT can be global (g), per set of branches (s), or per branch (p)


GAg
Global Branch History Register (GBHR)
Global Pattern History Table (GPHT)
Index
PAg
Per-address Branch History Table (PBHT)
Global Pattern History Table (GPHT)
Index
PAp
Per-address Branch History Table (PBHT)
Per-address Pattern History Tables (PPHT)
Index

- Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

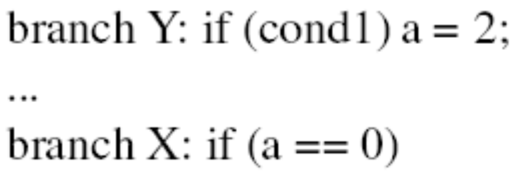
Global Branch Correlation (I)

- GAg: Global branch predictor (commonly called)
- Exploits global correlation across branches
- Recently executed branch outcomes in the execution path is correlated with the outcome of the next branch



```
if (cond1)
...
if (cond1 AND cond2)
```

- If first branch not taken, second also not taken



```
branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)
```

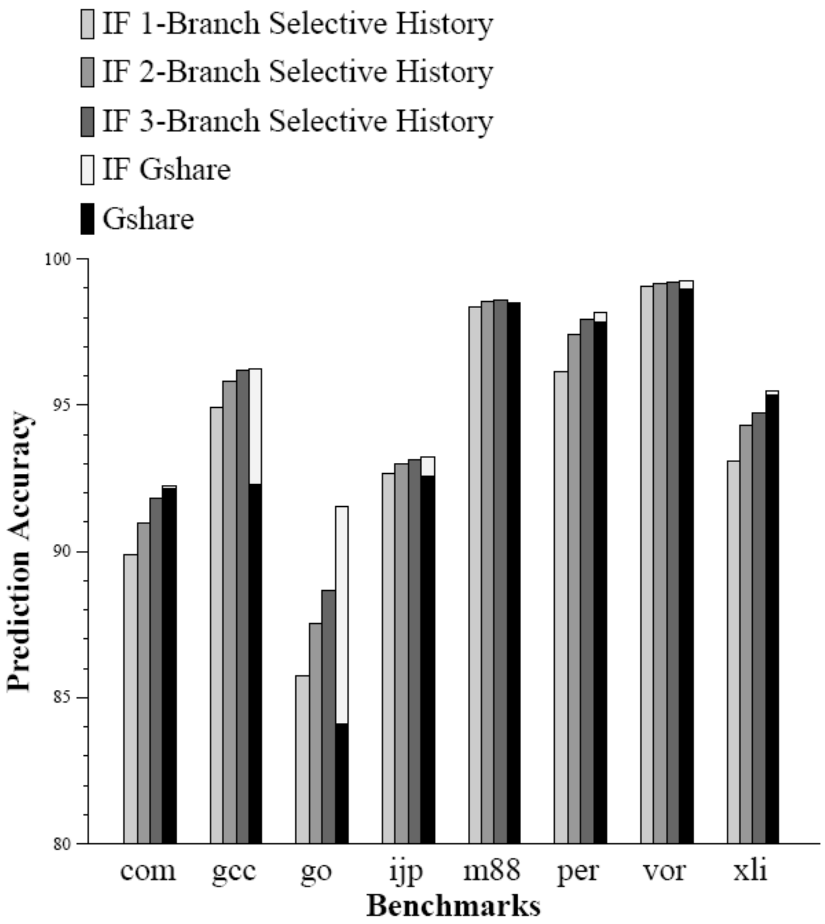
- If first branch taken, second definitely not taken

Global Branch Correlation (II)

```
branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND cond2)
```

- If Y and Z both taken, then X also taken
- If Y or Z not taken, then X also not taken

- Only 3 past branches' directions really matter (not necessarily the last 3 past branches)
- Evers et al., "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work," ISCA 1998.


IF 1-Branch Selective History
IF 2-Branch Selective History
IF 3-Branch Selective History
IF Gshare
Gshare
Prediction Accuracy
com
gcc
go
ijp
m88
per
vor
xli
Benchmarks