# 18-447
# Computer Architecture
# Lecture 11: Branch Prediction

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2013, 2/11/2013

# Reminder: Homework 2 Due Today

- Homework 2
  - Due today

- Homework 3
  - Will be out later today!
  - REP MOVS in Microprogrammed LC-3b, Pipelining, Delay Slots, Interlocking, …

# Reminder: Lab Assignment 2 Due Friday

- Lab Assignment 2
  - Due this Friday, Feb 15
  - Single-cycle MIPS implementation in Verilog
  - All labs are individual assignments
  - No collaboration; please respect the honor code
  - Do not forget the extra credit portion!

# Readings for Next Few Lectures

- P&H Chapter 4.9-4.11

- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

- Recommended:
  - McFarling, "Combining Branch Predictors," DEC WRL Technical Report, 1993.

# Also, …

- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

# Tomorrow: IEEE Tech Talk and CALCM Seminar

- Tuesday, February 12, 4:30-6:30pm, Hamerschlag 1107
- Dr. Richard E. Kessler, Cavium Fellow and Principal Architect
- Designing Efficient Processor Cores for Multicore Networking
- Abstract:
  - The design of CPUs has always required a balance of performance and efficiency in power, area, and complexity. The emergence of multicore SoCs armed with accelerators for packet processing has shifted this balance from solely single-thread performance to a combination of single-thread performance and efficient parallel processing. This shift requires a new style of core with short and deterministic pipelines, caches and memory systems optimized for low latency and high bandwidth, and an architecture that scales to 48-plus cores on a chip. This talk demonstrates how continuously emerging application demands shaped the fundamental principles behind OCTEON processor cores and supporting on-chip accelerators.

# Monday: IEEE Tech Talk and CALCM Seminar

- Monday, February 11, 4:30-6:30pm, Hamerschlag 1107
- Dr. Richard E. Kessler, Cavium Fellow and Principal Architect
- Designing Efficient Processor Cores for Multicore Networking

- Break-out Session:
  - Cavium is building a community of university and industry partners around the 32-core OCTEON II solution, with evaluation boards in use by students and professors at several universities globally. This break-out session for students will be conducted at the conclusion of the talk above to describe the evaluation board hardware, the Cavium SDK, and various semester-long student projects appropriate for upper level undergraduates or first year masters students. Other aspects of the OCTEON program will be briefly described, including a multi-university workshop planned in May for students to present their OCTEON project and compete for the OCTEON Trophy.

# Last Lecture

- Control dependence handling

# Today's Agenda

- Branch prediction techniques
- Wrap up control dependence handling

# Control Dependence Handling

# Review: 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

# Review: How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

# How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

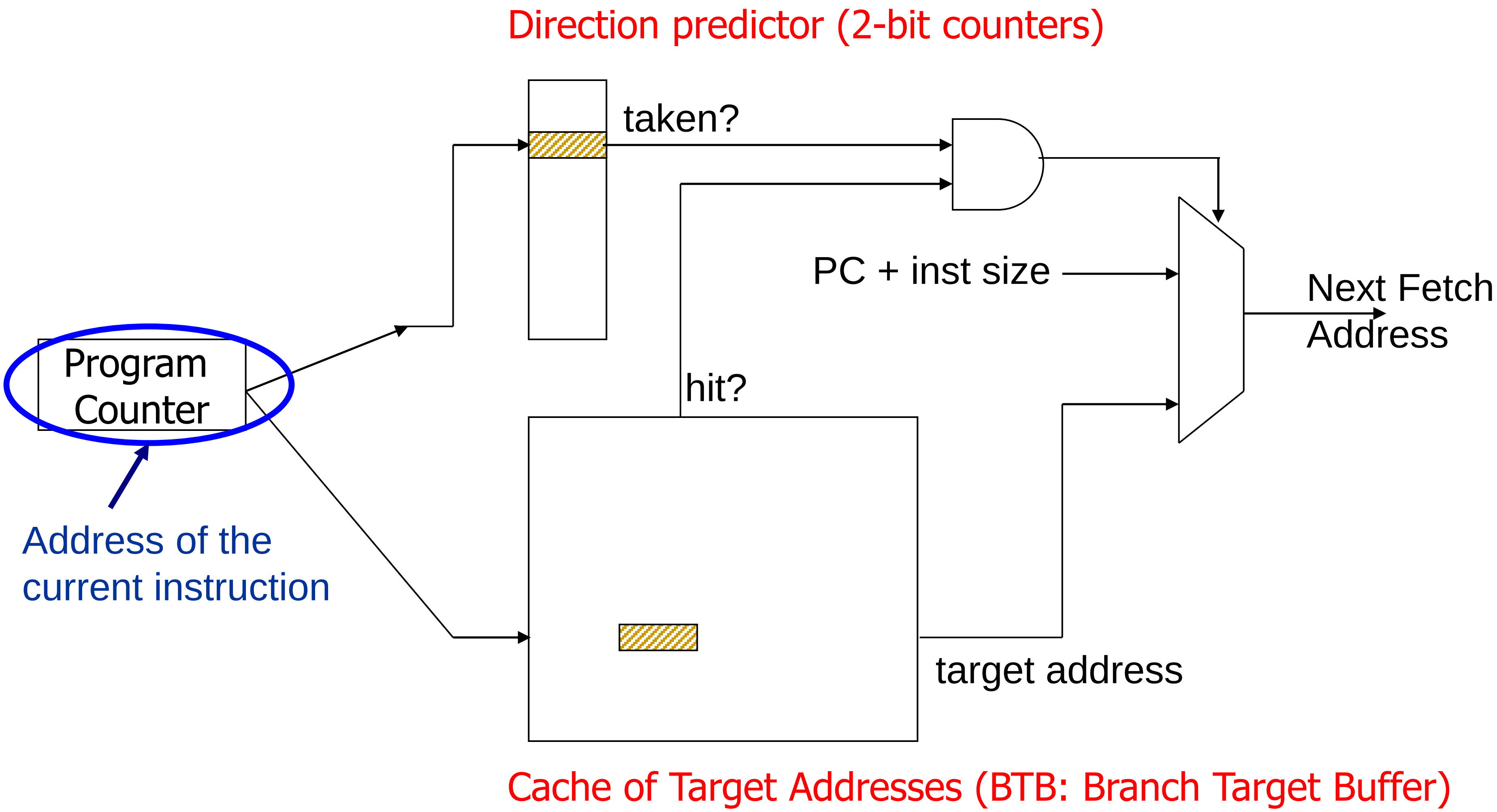
# Review: Branch Prediction

- Idea: Predict the next fetch address (to be used in the next cycle)

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

- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - Idea: Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

# Review: Fetch Stage with BTB


Direction predictor (2-bit counters)
taken?
PC + inst size
Next Fetch Address
Program Counter
hit?
Address of the current instruction
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

Always-taken CPI = [ 1 + (0.20*0.3) * 2 ]  = 1.12   (70% of branches taken)

# Simple Branch Direction Prediction Schemes

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

- Run time (dynamic)
  - Last time prediction (single-bit)

# More Sophisticated 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 prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid

# Static Branch Prediction (I)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40%
  - Compiler can layout code such that the likely path is the "not-taken" path

- Always taken
  - No direction prediction
  - Better accuracy: ~60-70%
    - 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) → accurate if profile is representative!

-- 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 (or, program analysis 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 compiler analysis and ISA support

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

# Static Branch Prediction (III)

- Programmer-based
  - Idea: Programmer provides the statically-predicted direction
  - Via pragmas in the programming language that qualify a branch as likely-taken versus likely-not-taken

+ Does not require profiling or program analysis

+ Programmer may know some branches and their program better than other analysis techniques

-- Requires programming language, compiler, ISA support

-- Burdens the programmer?

# Aside: Pragmas

- Idea: Keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy

- if (likely(x)) { ... }
- if (unlikely(error)) { … }

- Many other hints and optimizations can be enabled with pragmas
  - E.g., whether a loop can be parallelized
  - **#pragma omp parallel**
  - **Description**
    - The omp parallel directive explicitly instructs the compiler to parallelize the chosen segment of code.

# Static Branch Prediction

- All previous techniques can be combined
  - Profile based
  - Program based
  - Programmer based

- How would you do that?

- What are common disadvantages of all three techniques?
  - Cannot adapt to dynamic changes in branch behavior
    - This can be mitigated by a dynamic compiler, but not at a fine granularity (and a dynamic compiler has its overheads…)

# Dynamic Branch Prediction

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

- Advantages

  + Prediction based on history of the execution of branches

    + It can adapt to dynamic changes in branch behavior

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

- 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

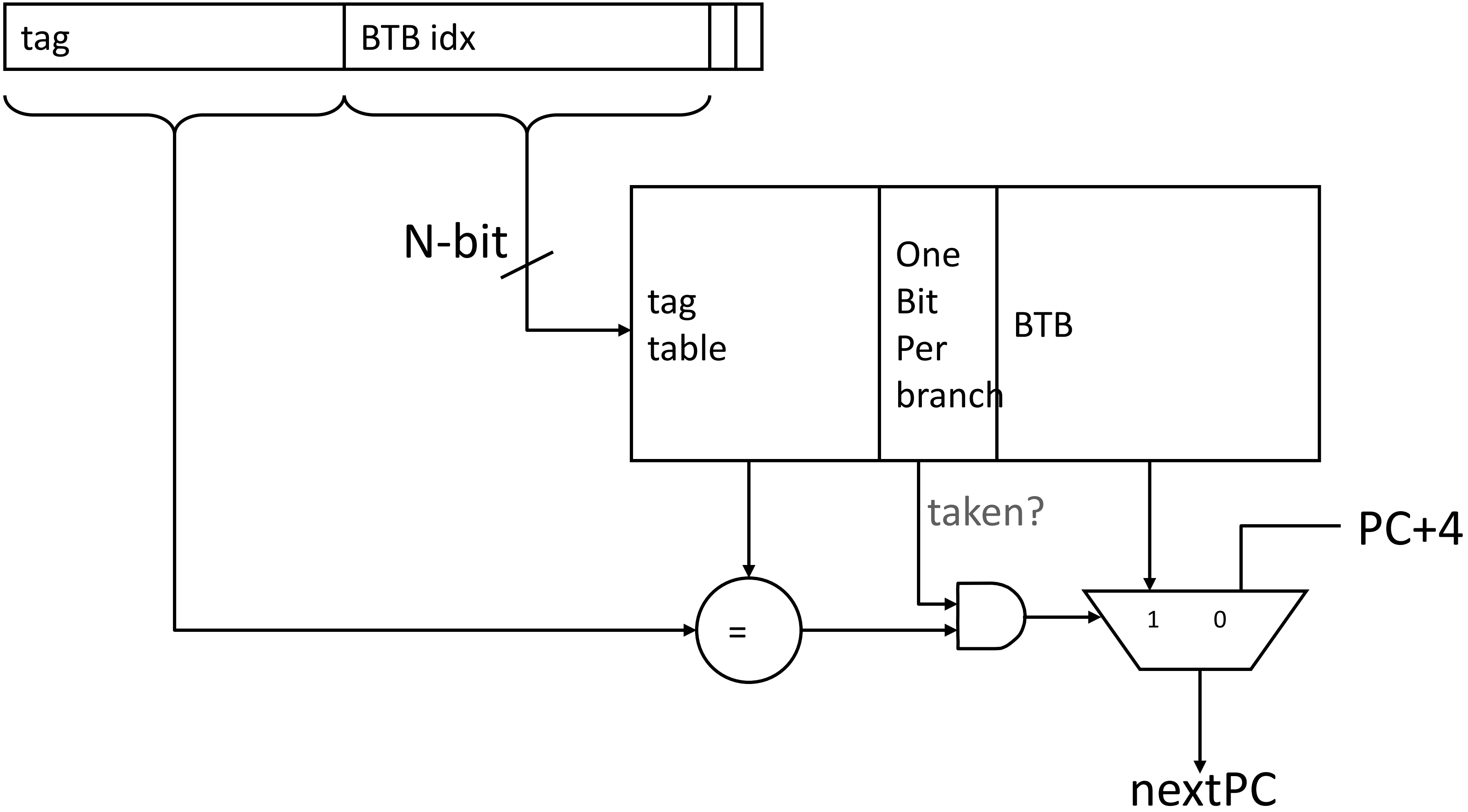
+ Loop branches for loops with large number of iterations

-- Loop branches for loops will small number of iterations

TNTNTNTNTNTNTNTNTNTN → 0% accuracy

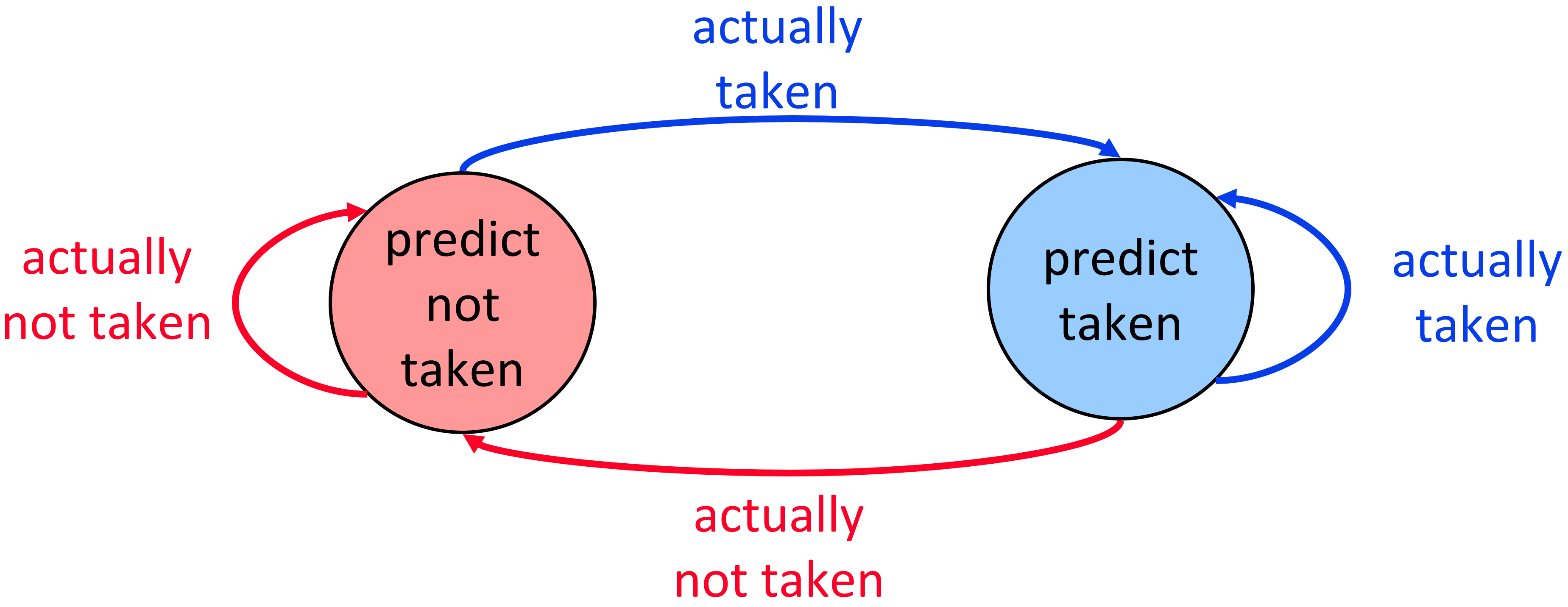
Last-time predictor CPI = [ 1 + (0.20*0.15) * 2 ] = 1.06 (Assuming 85% accuracy)

# Implementing the Last-Time Predictor



The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch

# State Machine for Last-Time Prediction



# Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - even though the branch may be mostly taken or mostly not taken

- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each

- Smith, "A Study of Branch Prediction Strategies," ISCA 1981.

# 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)

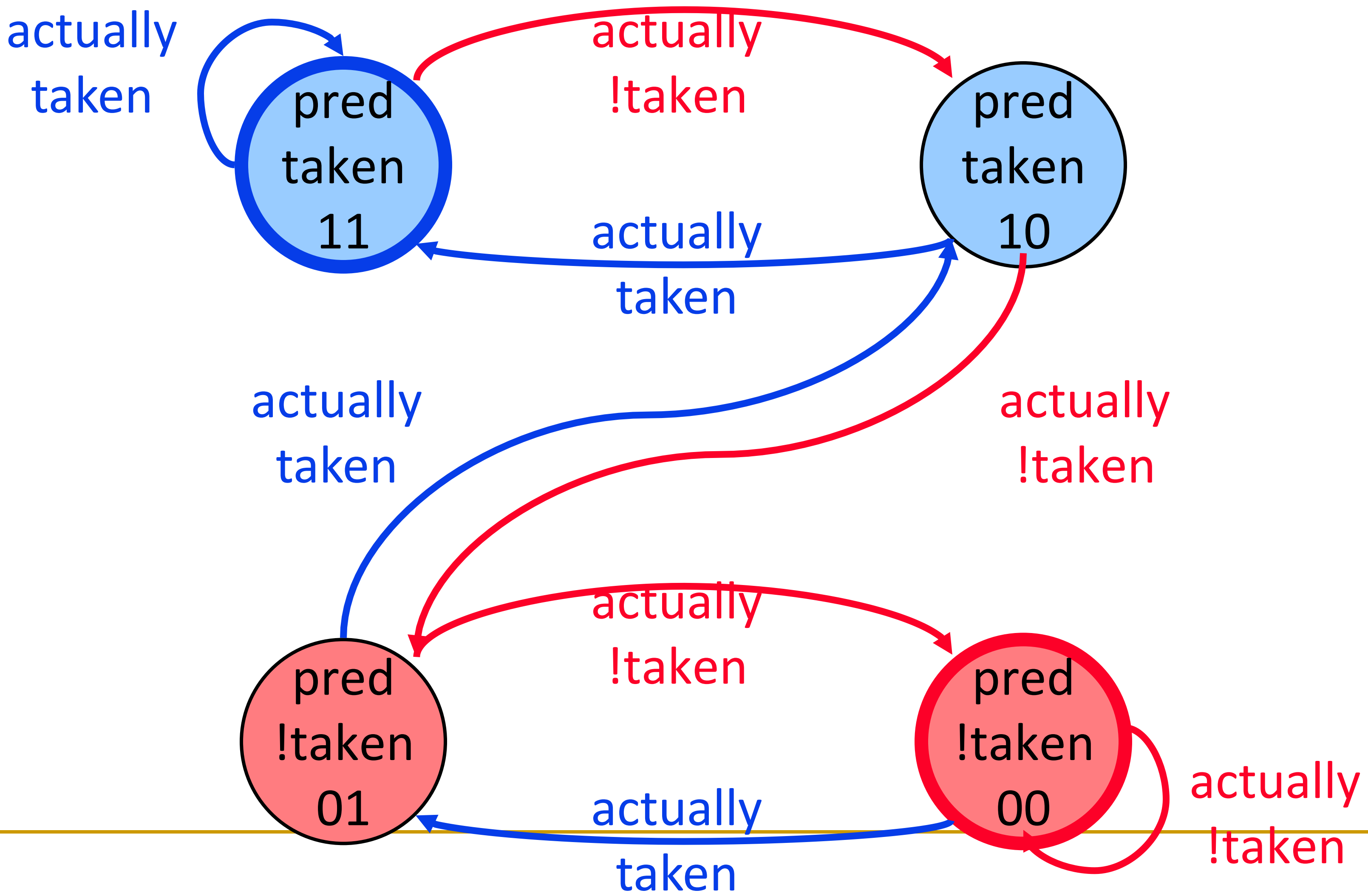
+ Better prediction accuracy

  2BC predictor CPI = [ 1 + (0.20*0.10) * 2 ] = 1.04 (90% accuracy)

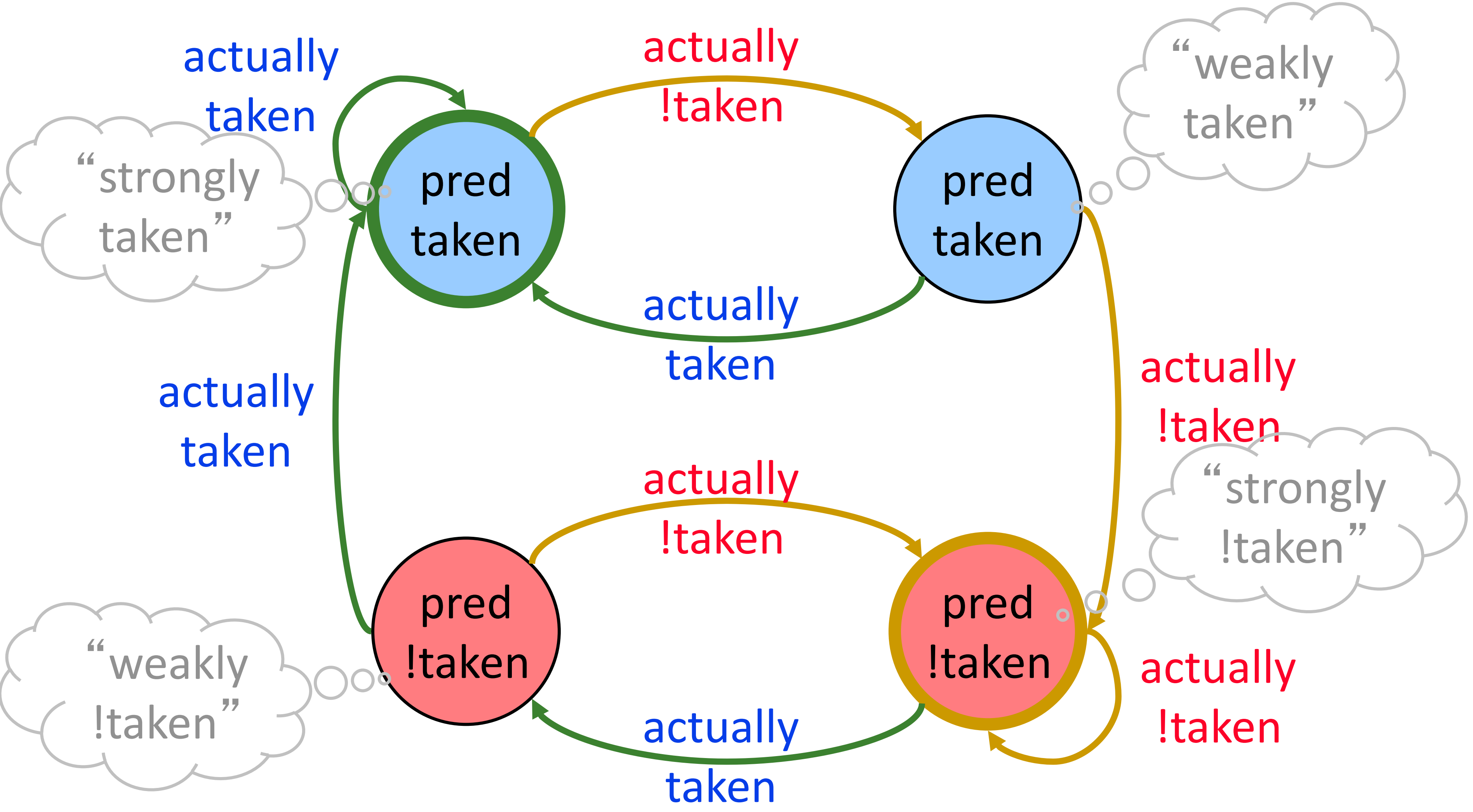
-- More hardware cost (but counter can be part of a BTB entry)

# State Machine for 2-bit Saturating Counter

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



# Hysteresis Using a 2-bit Counter



Change prediction after 2 consecutive mistakes

# Is This Enough?

- ~85-90% accuracy for many programs with 2-bit counter based prediction (also called bimodal prediction)

- Is this good enough?

- How big is the branch problem?

# Rethinking the 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 (IW: issue width)

- 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)

# Importance of 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

# Can We Do Better?

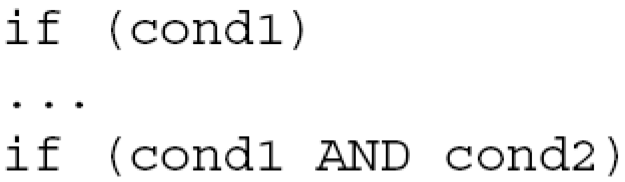
- Last-time and 2BC predictors exploit “last-time” predictability

- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  - Global branch correlation

- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
  - Local branch correlation

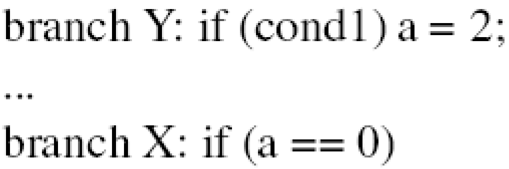
# Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path is correlated with the outcome of the next branch



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

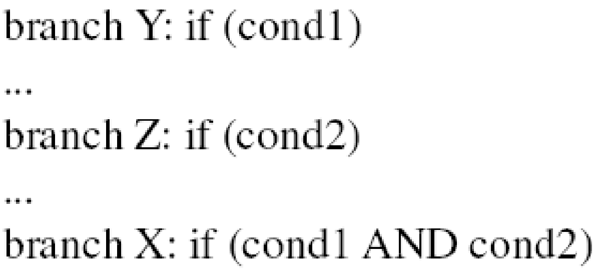
- If first branch not taken, second also not taken



```
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

# Global Branch Correlation (III)

- Eqntott, SPEC 1992

```
if (aa==2)                ;; B1
        aa=0;
if (bb==2)                ;; B2
        bb=0;
if (aa!=bb) {             ;; B3
        ….
 }
```

If B1 is not taken (i.e. aa==0@B3) and B2 is not taken (i.e. bb=0@B3) then B3 is certainly taken

# Capturing Global Branch Correlation

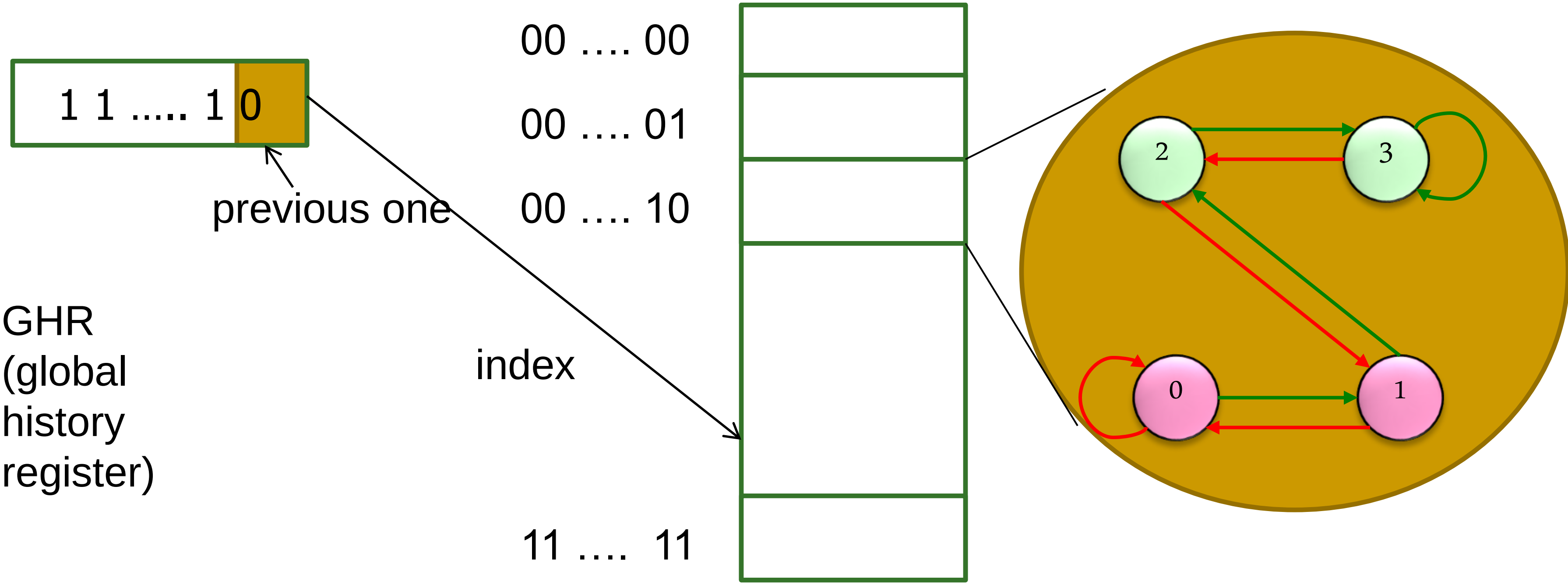
- Idea: Associate branch outcomes with "global T/NT history" of all branches
- Make a prediction based on the outcome of the branch the last time the same global branch history was encountered

- Implementation:
  - Keep track of the "global T/NT history" of all branches in a register → Global History Register (GHR)
  - Use GHR to index into a table of that recorded the outcome that was seen for that GHR value in the recent past → Pattern History Table (table of 2-bit counters)

- Global history/branch predictor
- Uses two levels of history (GHR + history at that GHR)

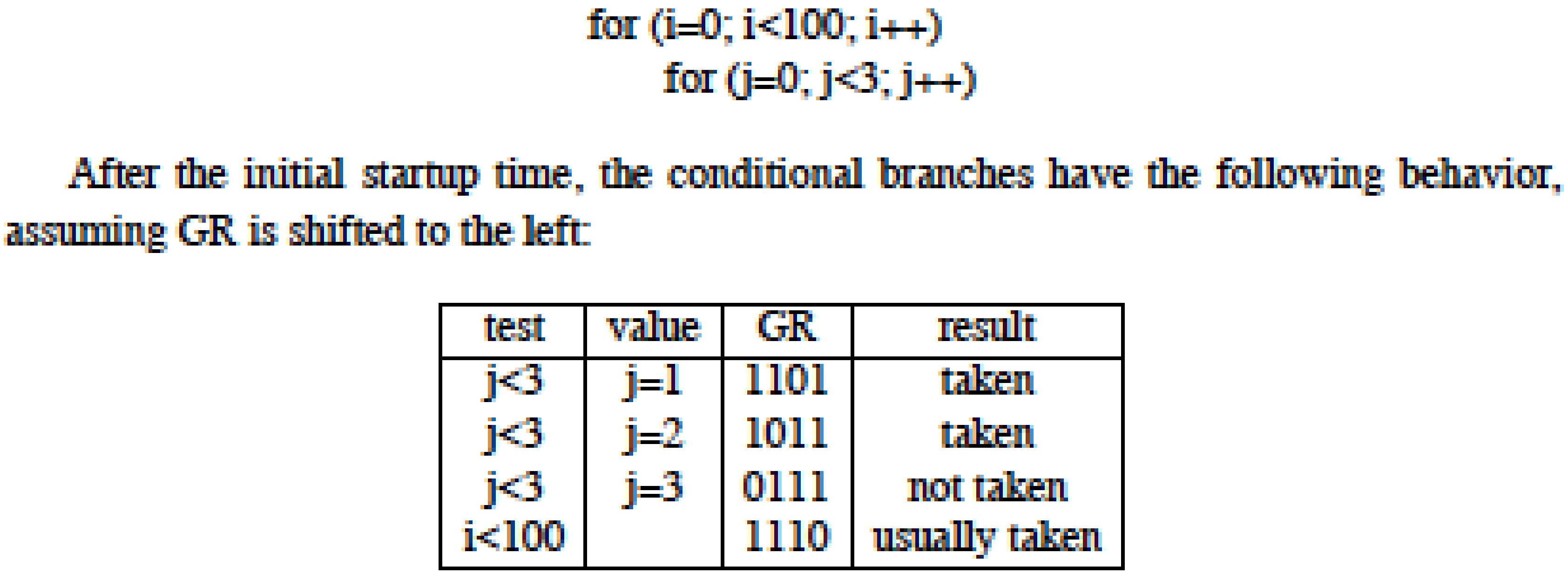
# Two Level Global Branch Prediction

- First level: Global 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



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

# How Does the Global Predictor Work?



```
for (i=0; i<100; i++)
      for (j=0; j<3; j++)
```

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

| test | value | GR | result |
|---|---|---|---|
| j<3 | j=1 | 1101 | taken |
| j<3 | j=2 | 1011 | taken |
| j<3 | j=3 | 0111 | not taken |
| i<100 | | 1110 | usually taken |

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

# Intel Pentium Pro Branch Predictor

- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
    - Which pattern history table to use is determined by lower order bits of the branch address

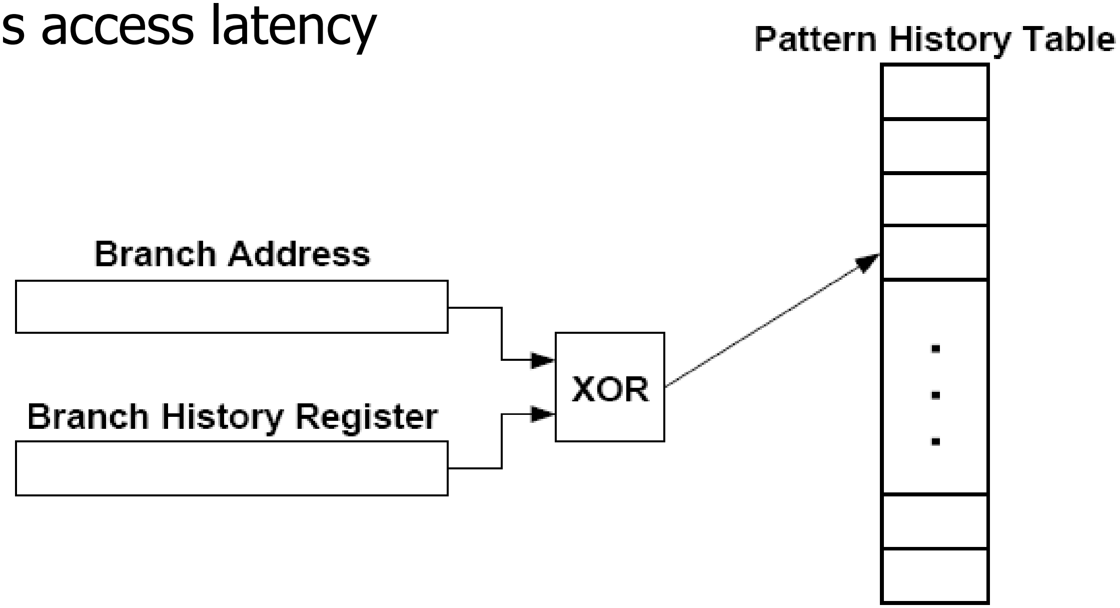
# Improving Global Predictor Accuracy

- Idea: Add more context information to the global predictor to take into account which branch is being predicted
  - Gshare predictor: GHR hashed with the Branch PC

  + More context information

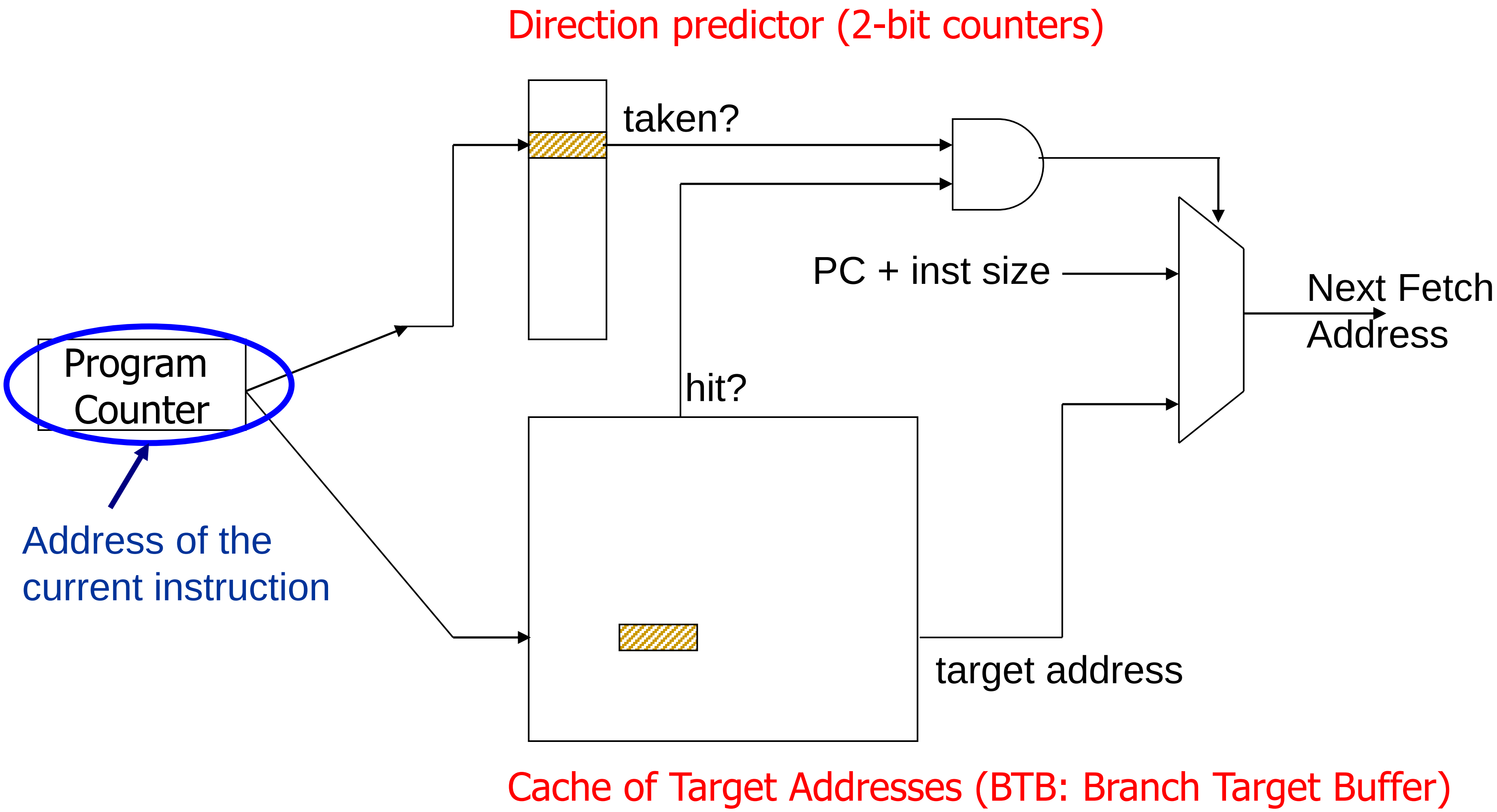
  + Better utilization of PHT

  -- Increases access latency

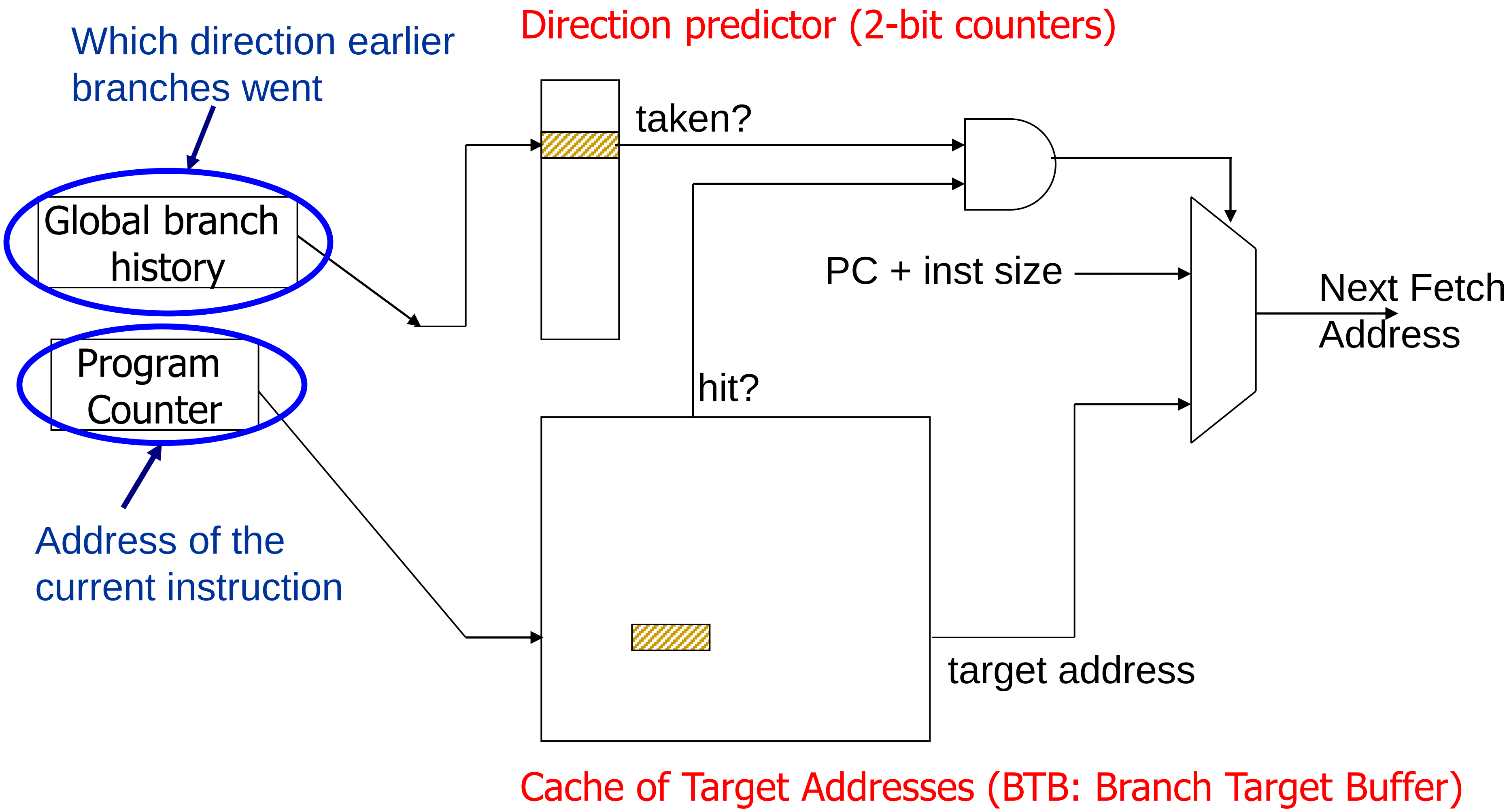


- McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

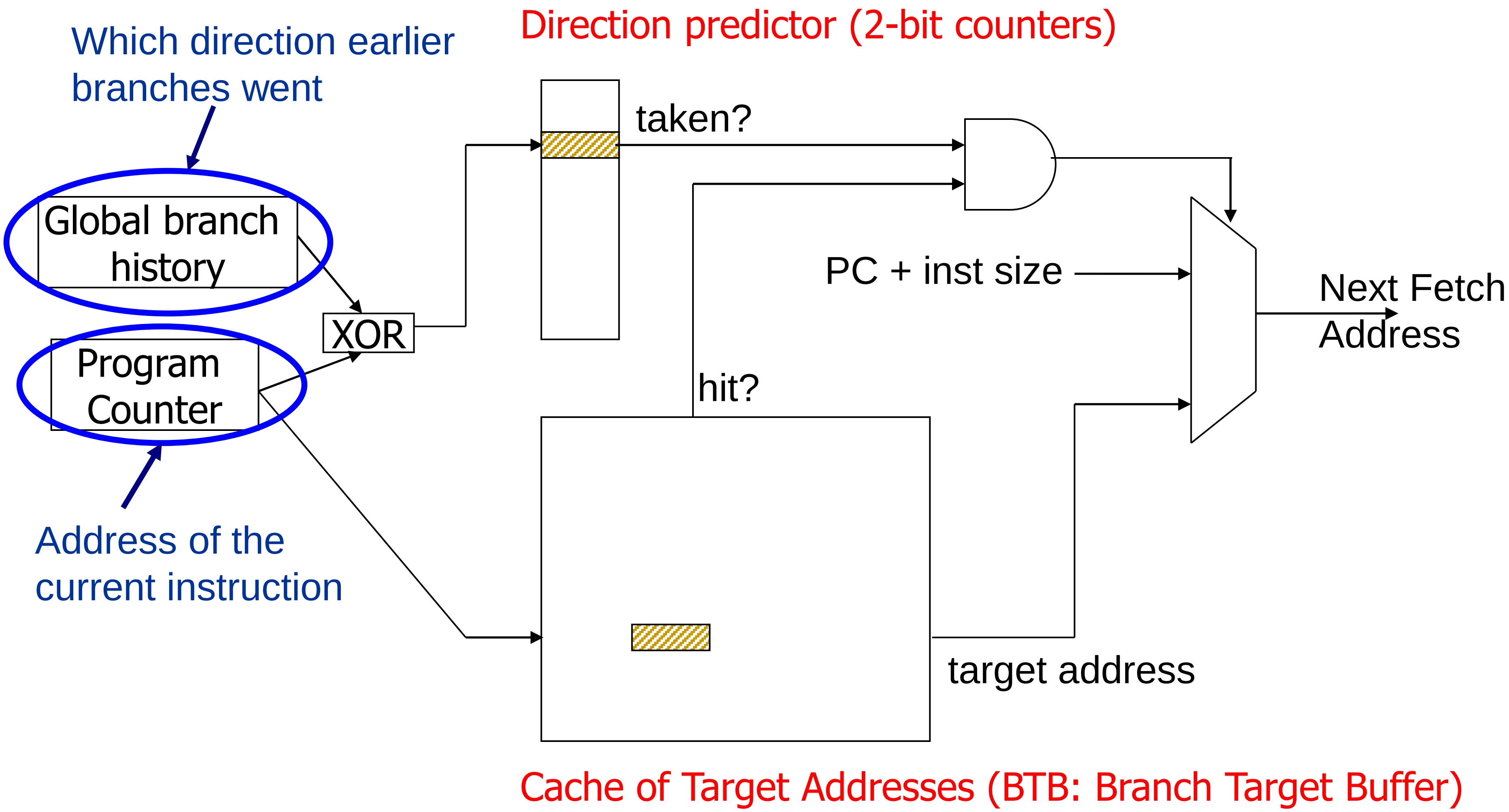
# One-Level Branch Predictor


Direction predictor (2-bit counters)
taken?
PC + inst size
Next Fetch Address
Program Counter
hit?
Address of the current instruction
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

# Two-Level Global History Predictor


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

# Two-Level Gshare Predictor



Which direction earlier branches went

Direction predictor (2-bit counters)

Global branch history

Program Counter

XOR

taken?

hit?

PC + inst size

Next Fetch Address

target address

Address of the current instruction

Cache of Target Addresses (BTB: Branch Target Buffer)

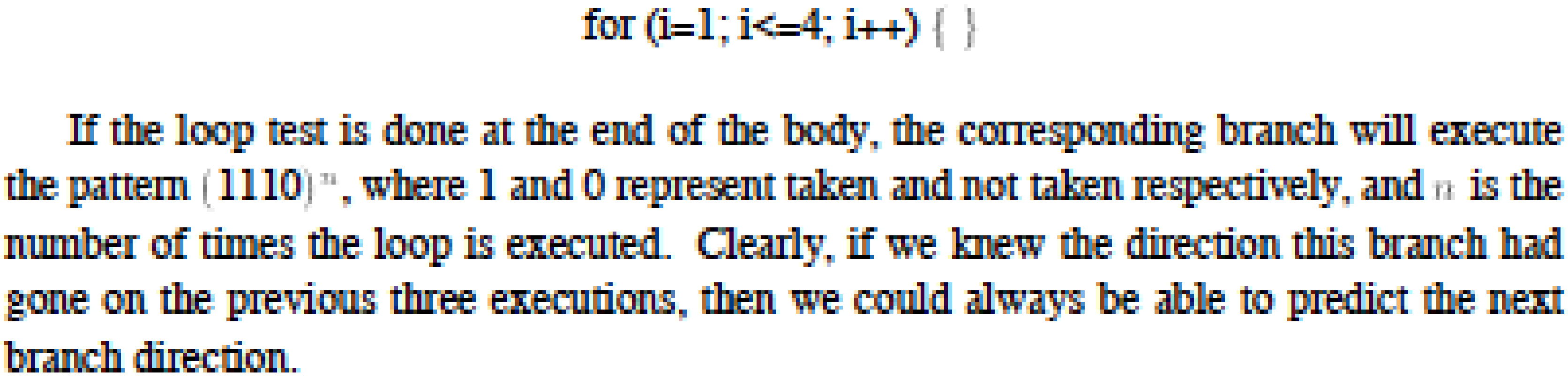
# Can We Do Better?

- Last-time and 2BC predictors exploit "last-time" predictability

- Realization 1: A branch's outcome can be correlated with other branches' outcomes
  - Global branch correlation

- Realization 2: A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)
  - Local branch correlation

# Local Branch Correlation



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.

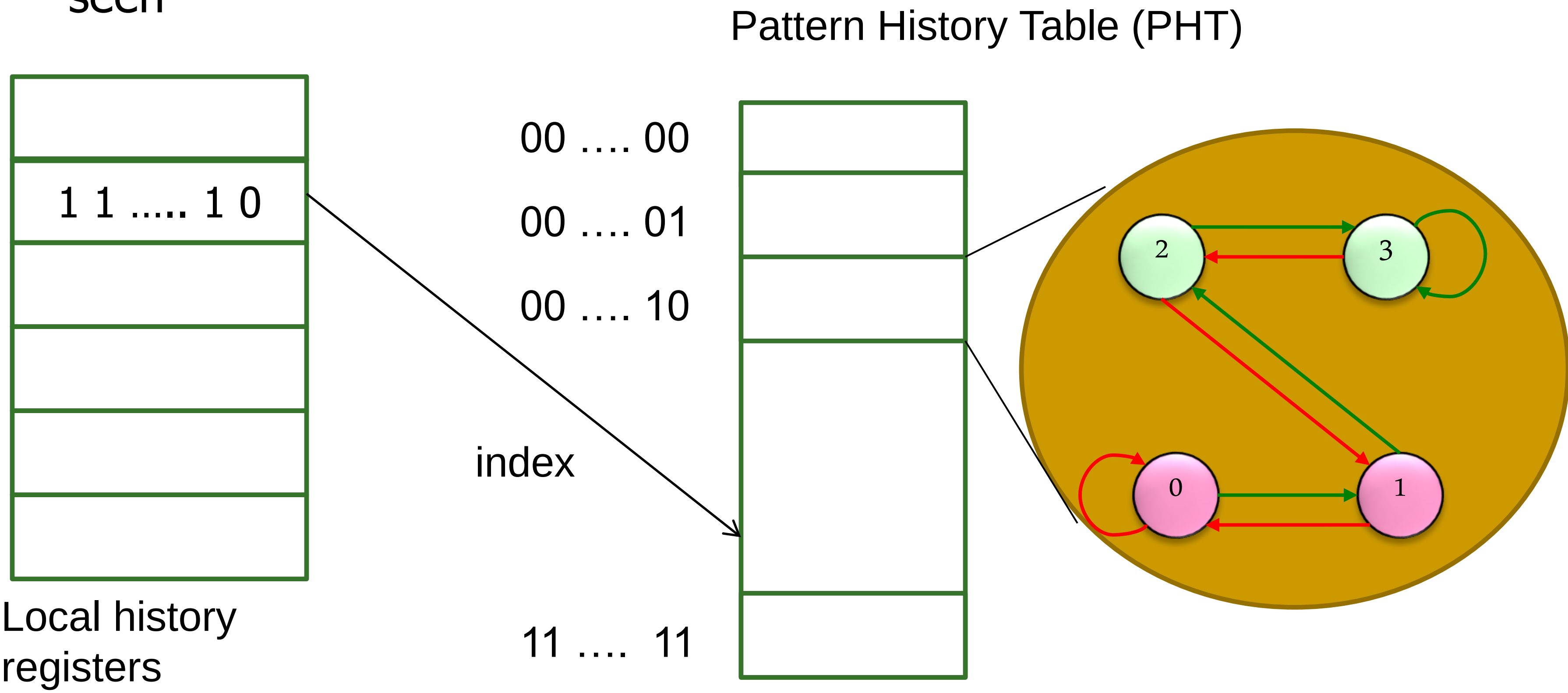
- McFarling, "Combining Branch Predictors," DEC WRL TR 1993.

# Capturing Local Branch Correlation

- Idea: Have a per-branch history register
  - Associate the predicted outcome of a branch with "T/NT history" of the same branch
- Make a prediction is based on the outcome of the branch the last time the same local branch history was encountered
- Called the local history/branch predictor
- Uses two levels of history (Per-branch history register + history at that history register value)

# Two Level Local Branch Prediction

- First level: A set of local history registers (N bits each)
  - Select the history register based on the PC of the branch
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen



Pattern History Table (PHT)

1 1 ….. 1 0

00 …. 00

00 …. 01

00 …. 10

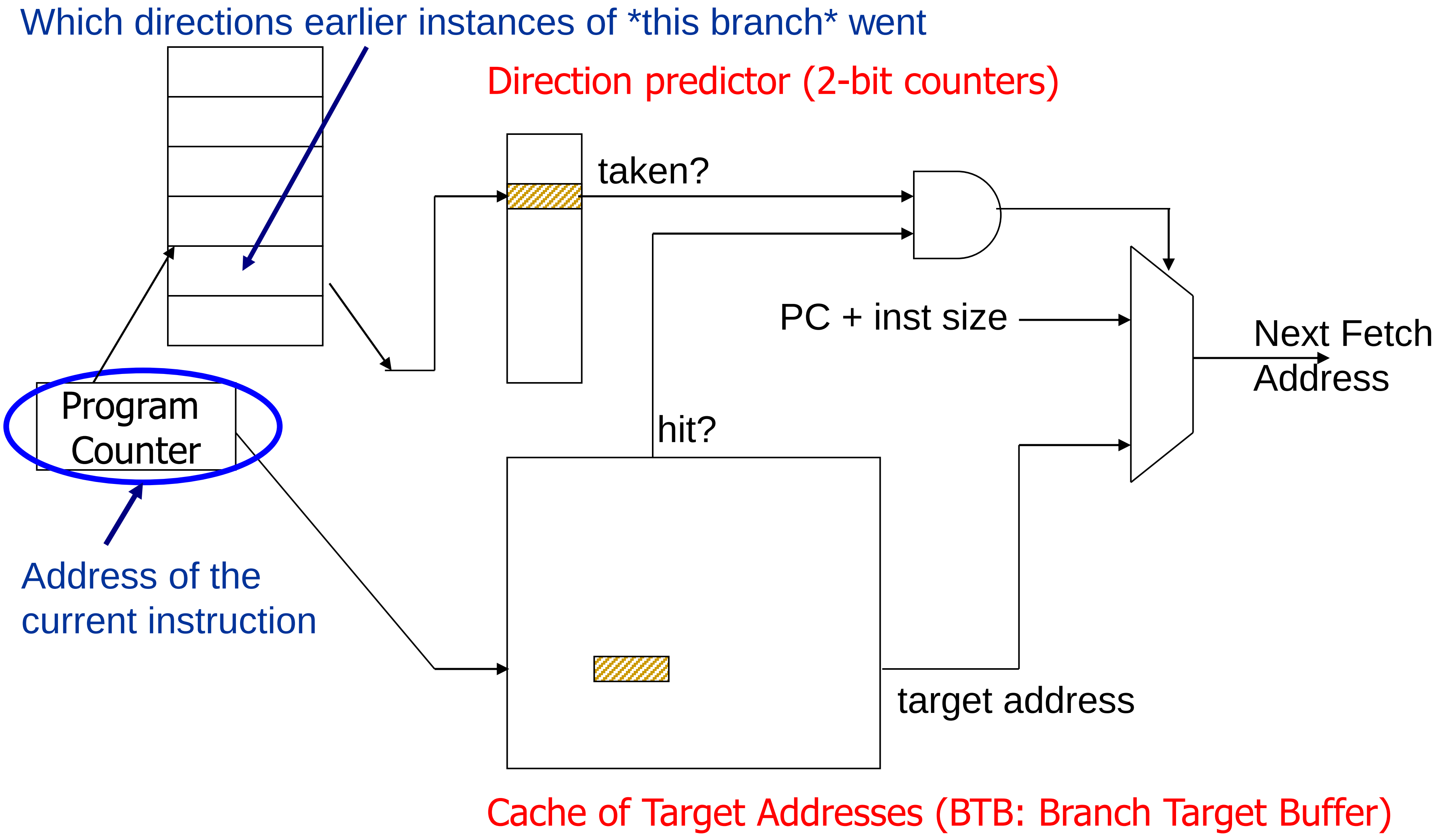
index

11 …. 11

Local history registers

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

# Two-Level Local History Predictor



Which directions earlier instances of *this branch* went

Direction predictor (2-bit counters)

Address of the current instruction

Cache of Target Addresses (BTB: Branch Target Buffer)

# Hybrid Branch Predictors

- Idea: Use more than one type of predictor (i.e., multiple algorithms) and select the "best" prediction
  - E.g., hybrid of 2-bit counters and global predictor

- Advantages:

  + Better accuracy: different predictors are better for different branches

  + Reduced warmup time (faster-warmup predictor used until the slower-warmup predictor warms up)

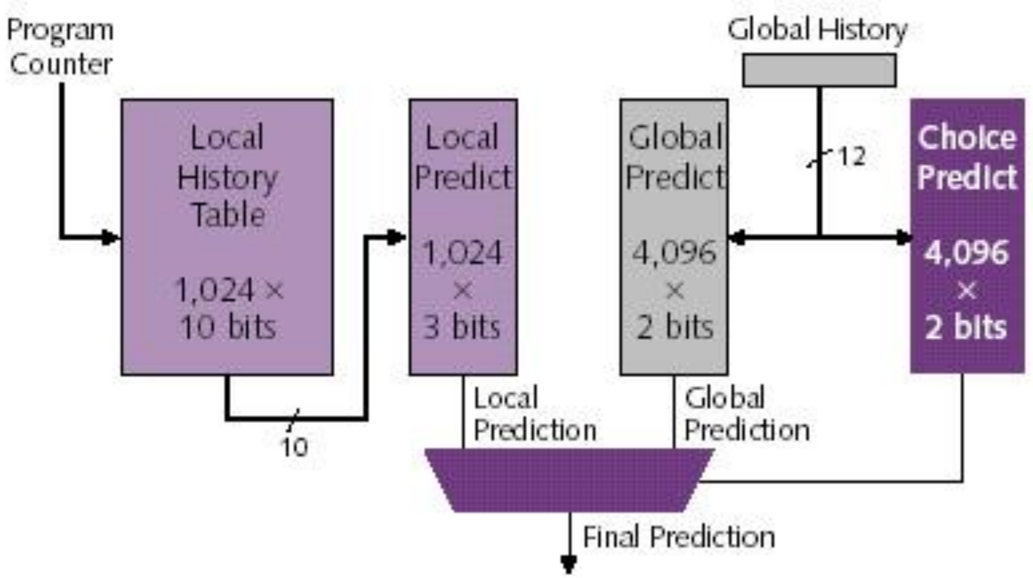
- Disadvantages:

  -- Need "meta-predictor" or "selector"

  -- Longer access latency

  - McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

# Alpha 21264 Tournament Predictor



- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch

- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

# Branch Prediction Accuracy (Example)

- Bimodal: table of 2bc indexed by branch address

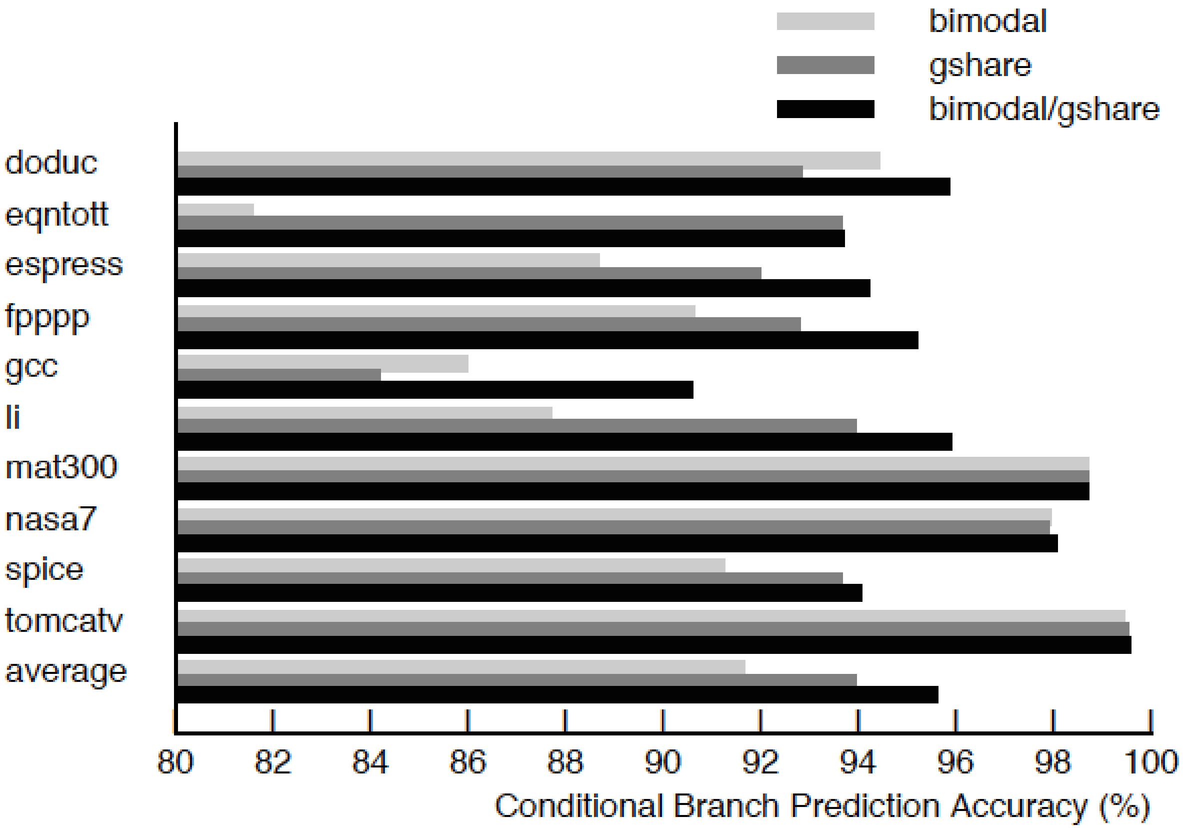


Figure 13: Combined Predictor Performance by Benchmark

# Biased Branches

- Observation: Many branches are biased in one direction (e.g., 99% taken)

- Problem: These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing "interference" in branch prediction tables and history registers

- Solution: Detect such biased branches, and predict them with a simpler predictor

- Chang et al., "Branch classification: a new mechanism for improving branch predictor performance," MICRO 1994.