18-740/640
Computer Architecture
Lecture 5: Advanced Branch Prediction

Prof. Onur Mutlu
Carnegie Mellon University
Fall 2015, 9/16/2015

Required Readings

- Required Reading Assignment:
  - **Chapter 5 of Shen and Lipasti (SnL).**

- Recommended References:

  - T. Yeh and Y. Patt, "Two-Level Adaptive Training Branch Prediction," Intl. Symposium on Microarchitecture, November 1991.
    MICRO Test of Time Award Winner (after 24 years)

  - Kessler, R. E., "The Alpha 21264 Microprocessor," IEEE Micro, March/April 1999, pp. 24-36 (available at ieeexplore.ieee.org).

  - McFarling, S., "Combining Branch Predictors," DEC WRL Technical Report, TN-36, June 1993.

Also Recommended …

- 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

Control Dependence

- Question: What should the fetch PC be in the next cycle?
- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?
- In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?

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)

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

- If we are fetching W instructions per cycle (i.e., if the pipeline is W wide)
  - A branch misprediction leads to **N x W** wasted instruction slots

Importance of The Branch Problem

- Assume N = 20 (20 pipe stages), W = 5 (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

- How long does it take to fetch 500 instructions?
  - 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

Simplest: Always Guess NextPC = PC + 4

- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of next fetch address prediction (and branch prediction)

- How can you make this more effective?

- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed
  - Software: Lay out the control flow graph such that the "likely next instruction" is on the not-taken path of a branch
    - Profile guided code positioning → Pettis & Hansen, PLDI 1990.
  - Hardware: ??? (how can you do this in hardware…)
    - Cache traces of executed instructions → Trace cache

Guessing NextPC = PC + 4

- How else can you make this more effective?
- Idea: Get rid of control flow instructions (or minimize their occurrence)
- How?

  1. Get rid of unnecessary control flow instructions → combine predicates (predicate combining)
  2. Convert control dependences into data dependences → predicated execution

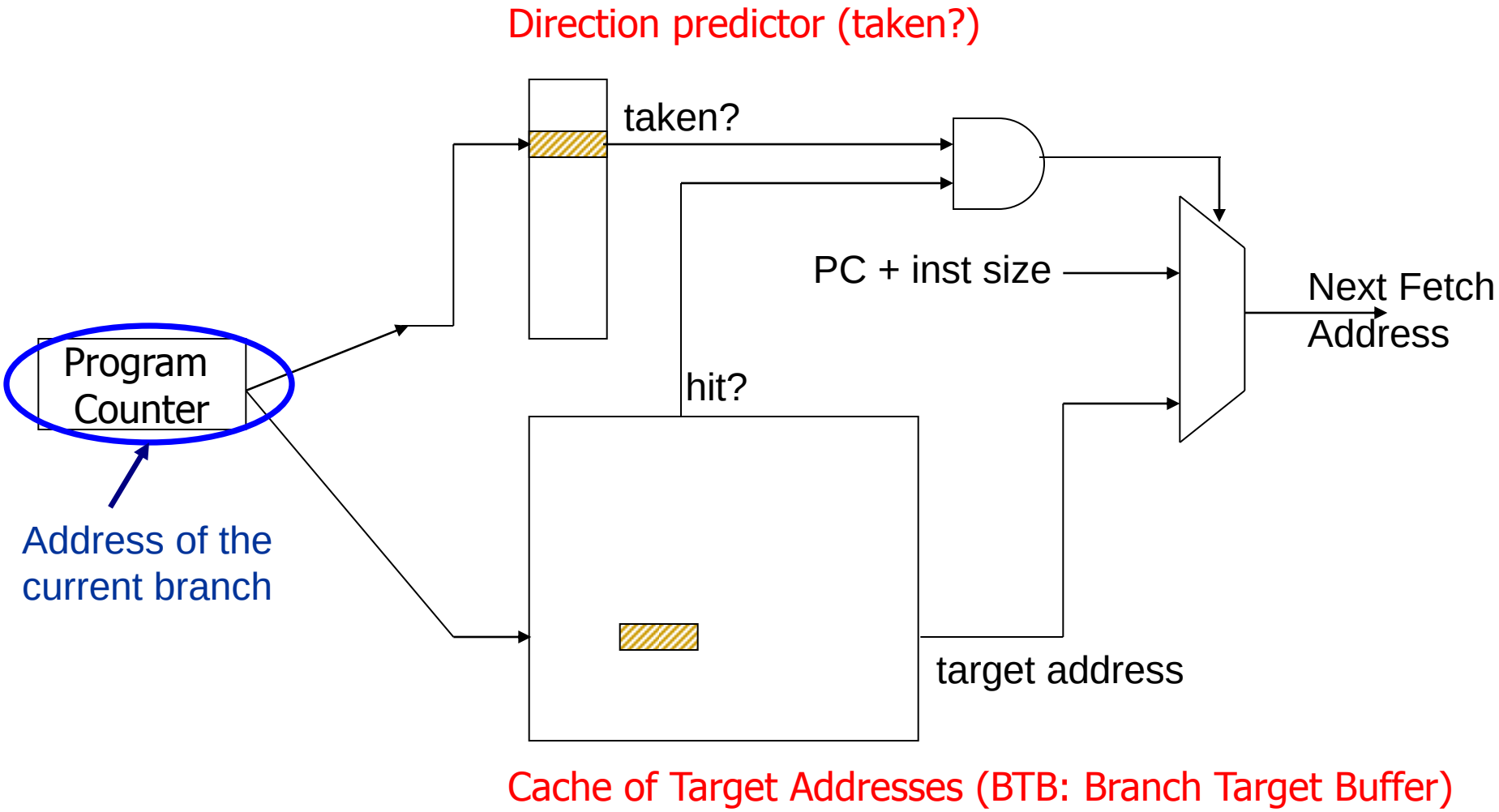
Branch Prediction (A Bit More Enhanced)

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

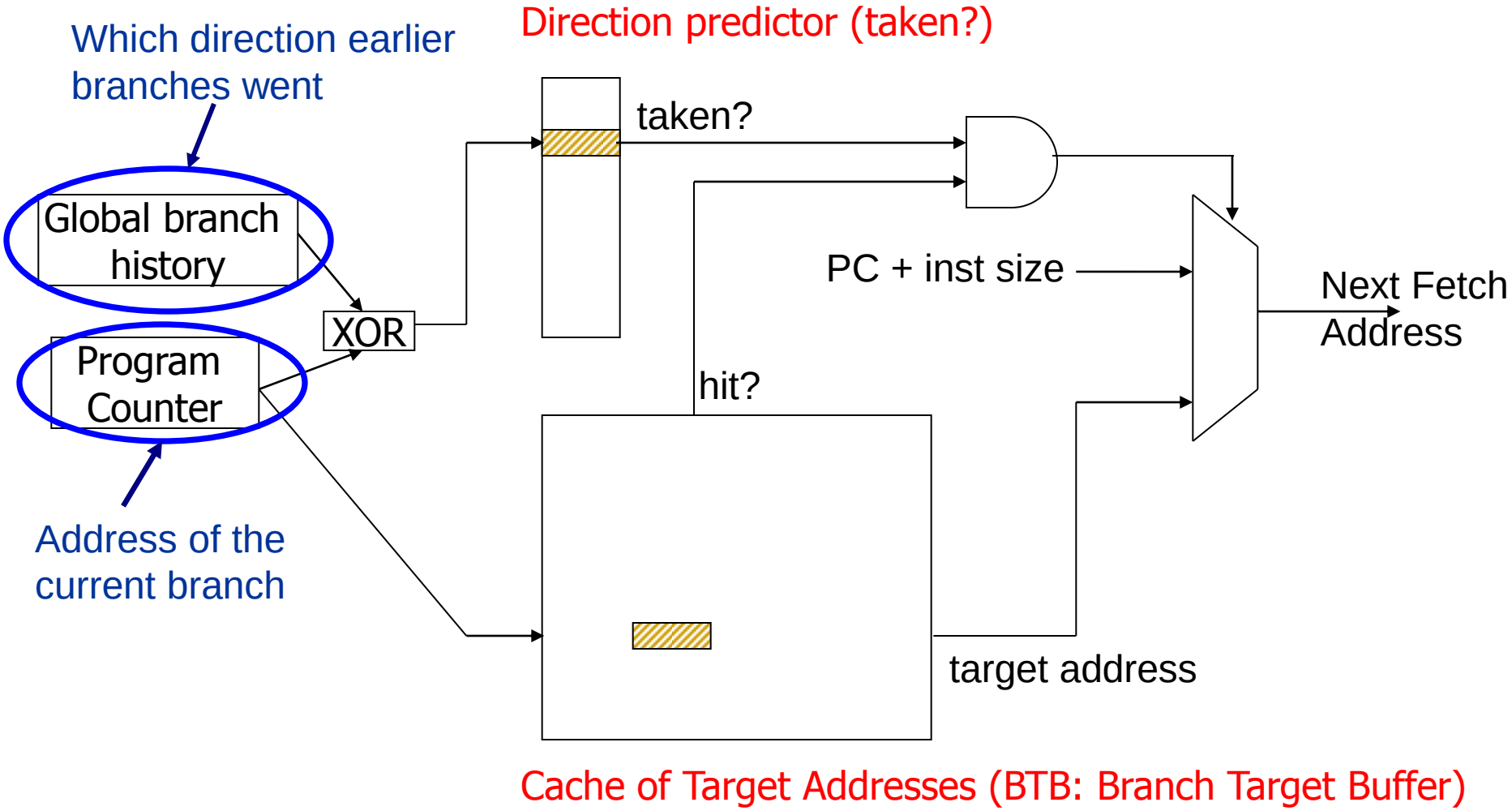
- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)

- 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

Fetch Stage with BTB and Direction Prediction



More Sophisticated Branch Direction Prediction



Three Things to Be Predicted

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

- Third (3.) can be accomplished using a BTB
  - Remember target address computed last time branch was executed
- First (1.) can be accomplished using a BTB
  - If BTB provides a target address for the program counter, then it must be a branch
  - Or, we can store "branch metadata" bits in instruction cache/memory → partially decoded instruction stored in I-cache
- Second (2.): How do we predict the direction?

Simple Branch Direction Prediction Schemes

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

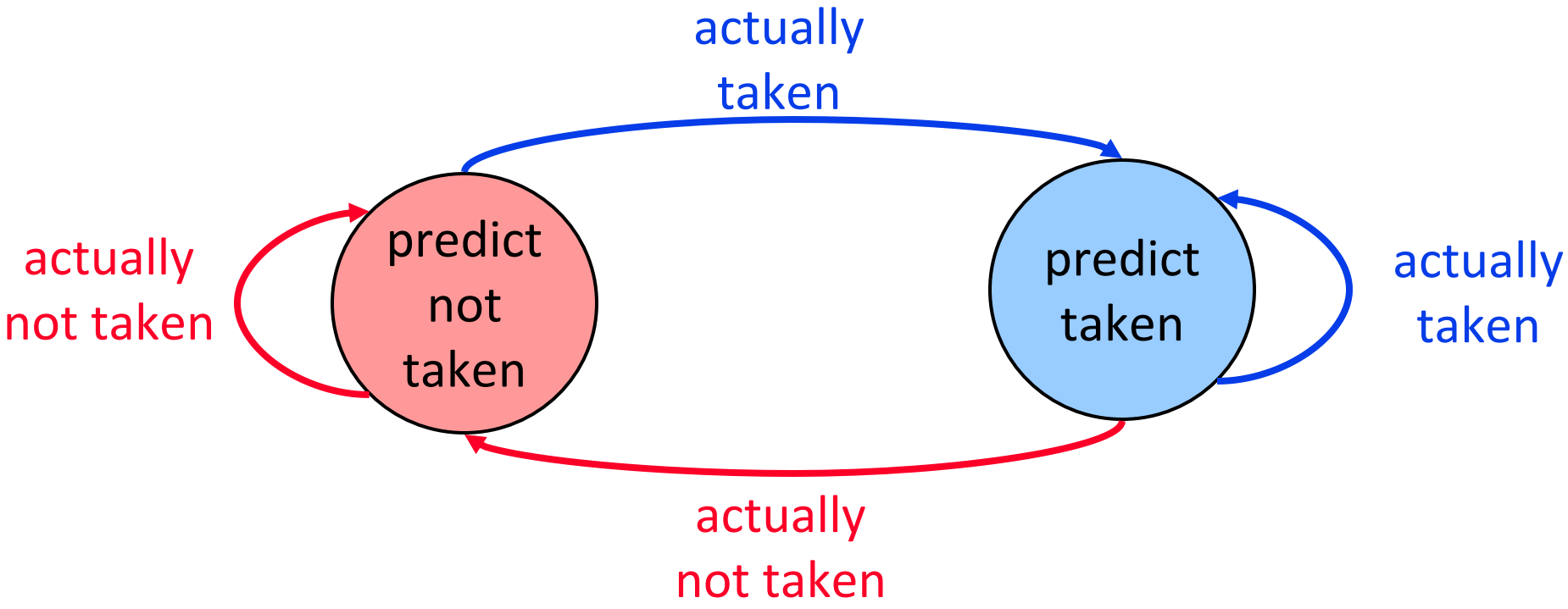
- Run time (dynamic)
  - Last time prediction (single-bit)

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
  - Advanced algorithms (e.g., using perceptrons)

Review: State Machine for Last-Time Prediction



Review: 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.

Review: 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

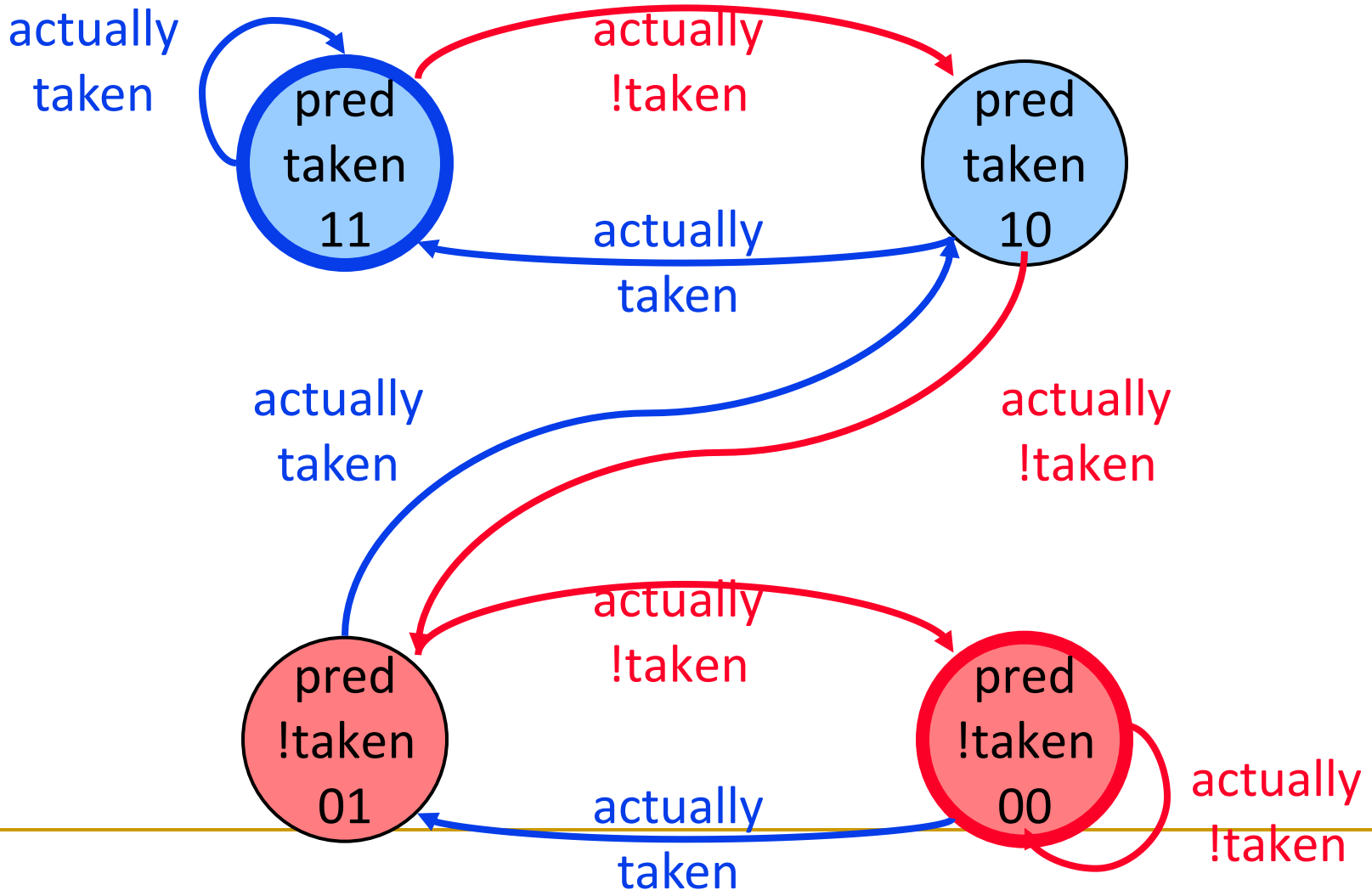
- Accuracy for a loop with N iterations = (N-1)/N
  TNTNTNTNTNTNTNTNTNTNTN → 50% accuracy

(assuming counter initialized to weakly taken)

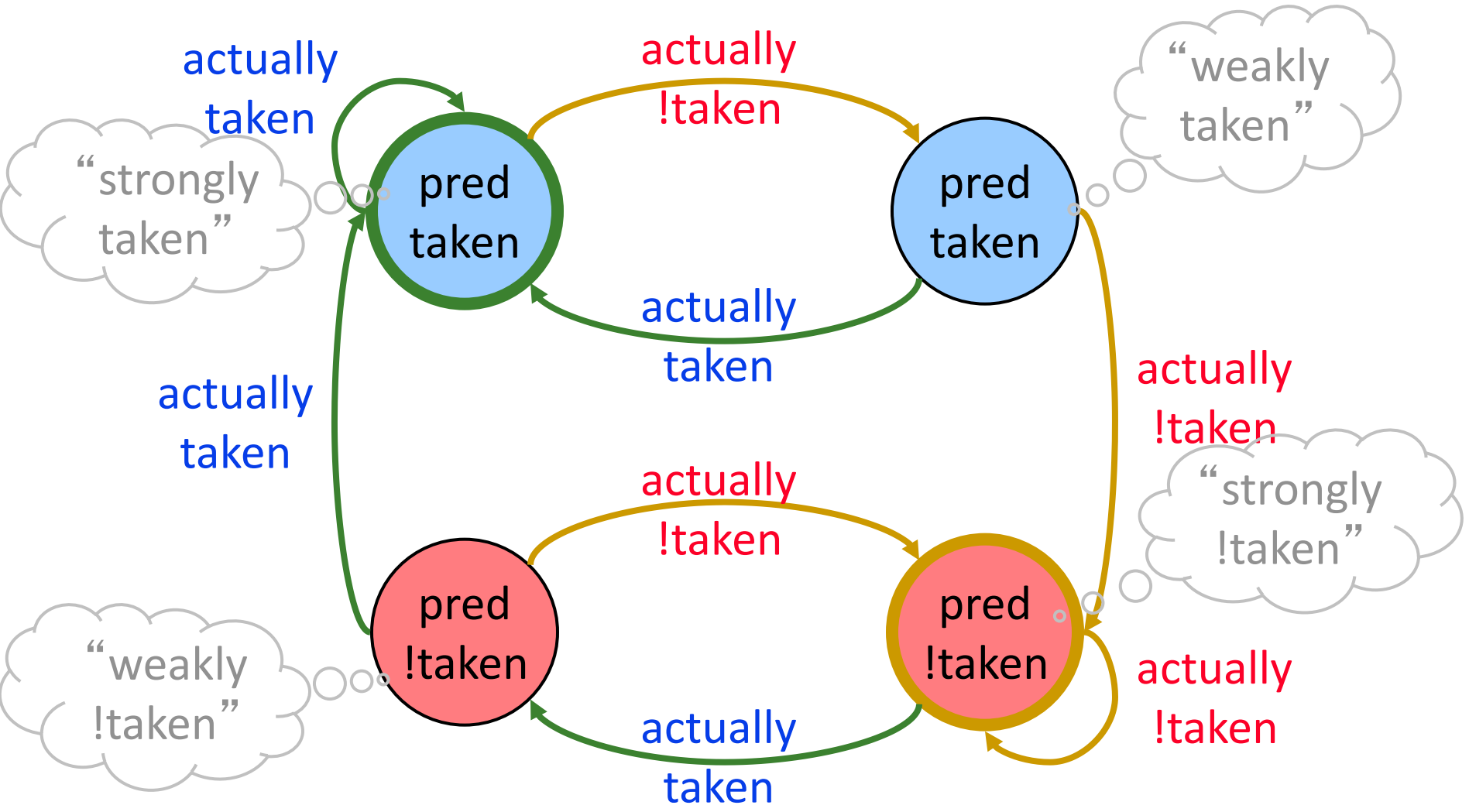
+ Better prediction accuracy
-- More hardware cost (but counter can be part of a BTB entry)

Review: State Machine for 2-bit Counter

- Counter using *saturating arithmetic*
  - Arithmetic with maximum and minimum values



Review: Hysteresis Using a 2-bit Counter



Change prediction after 2 consecutive mistakes

Is This Good 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?

Let's Do the Exercise Again

- Assume N = 20 (20 pipe stages), W = 5 (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

- How long does it take to fetch 500 instructions?
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 95% accuracy
    - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
    - 100% extra instructions fetched
  - 90% accuracy
    - 100 (correct path) + 20 * 10 (wrong path) = 300 cycles
    - 200% extra instructions fetched
  - 85% accuracy
    - 100 (correct path) + 20 * 15 (wrong path) = 400 cycles
    - 300% extra instructions fetched

Can We Do Better: Two-Level Prediction

- 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

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

Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path are 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'92: Generates truth table from Boolean expr.

```
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

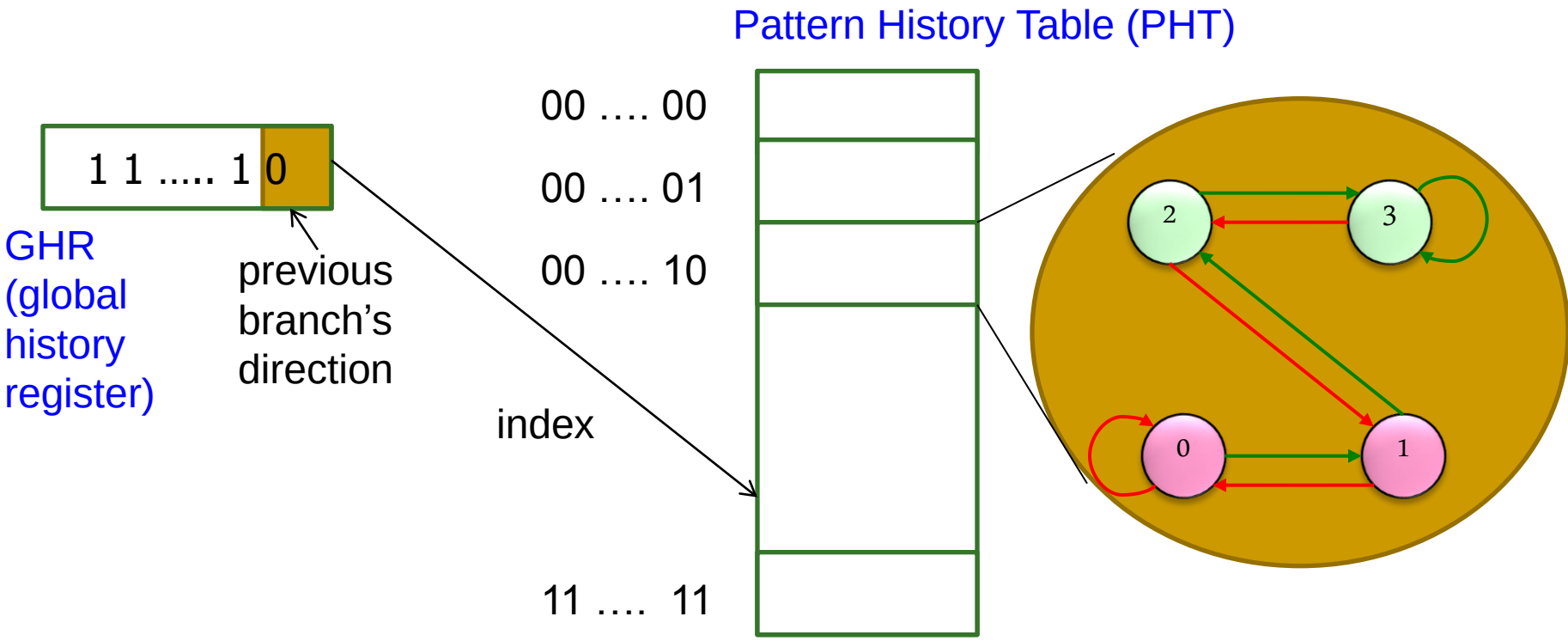
Capturing Global Branch Correlation

- 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 that recorded the outcome that was seen for each 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)

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

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

This branch tests i
Last 4 branches test j
History: TTTN
Predict taken for i
Next history: TTNT
(shift in last outcome)

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

Intel Pentium Pro Branch Predictor

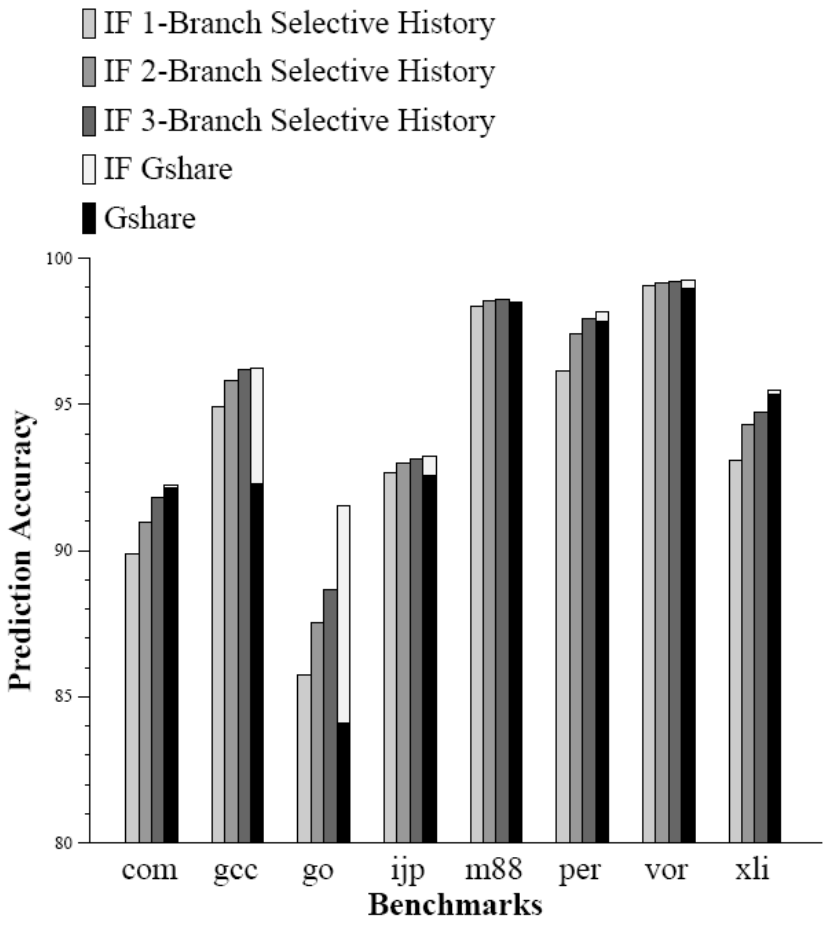
- Two level global branch predictor
- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
  - Which pattern history table to use is determined by lower order bits of the branch address

Global Branch Correlation Analysis

```
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
- Evers et al., "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work," ISCA 1998.



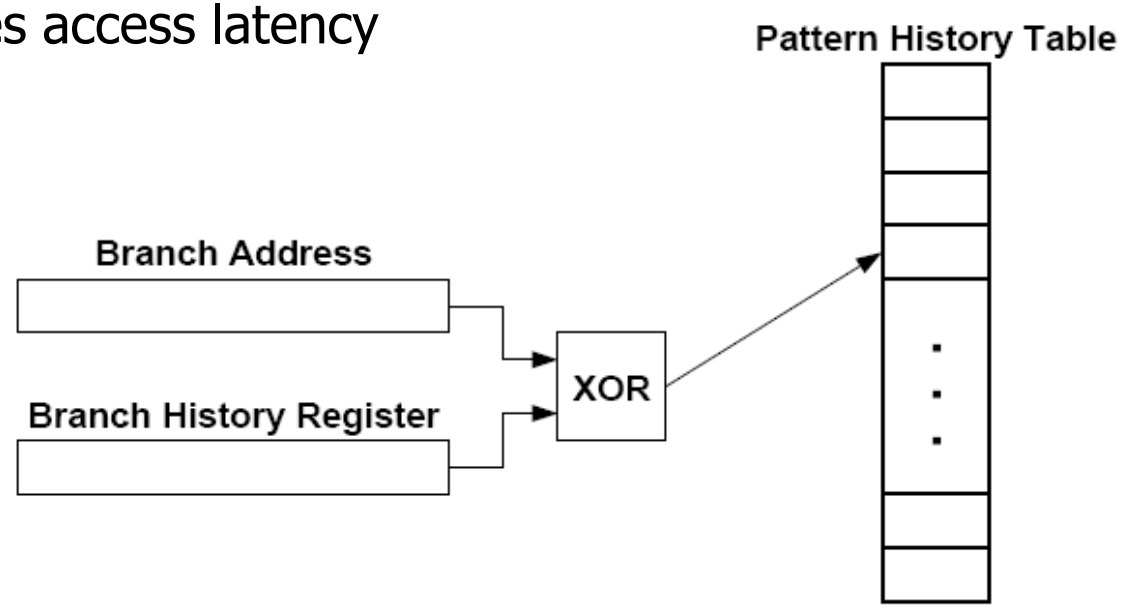
Improving Global Predictor Accuracy

- 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

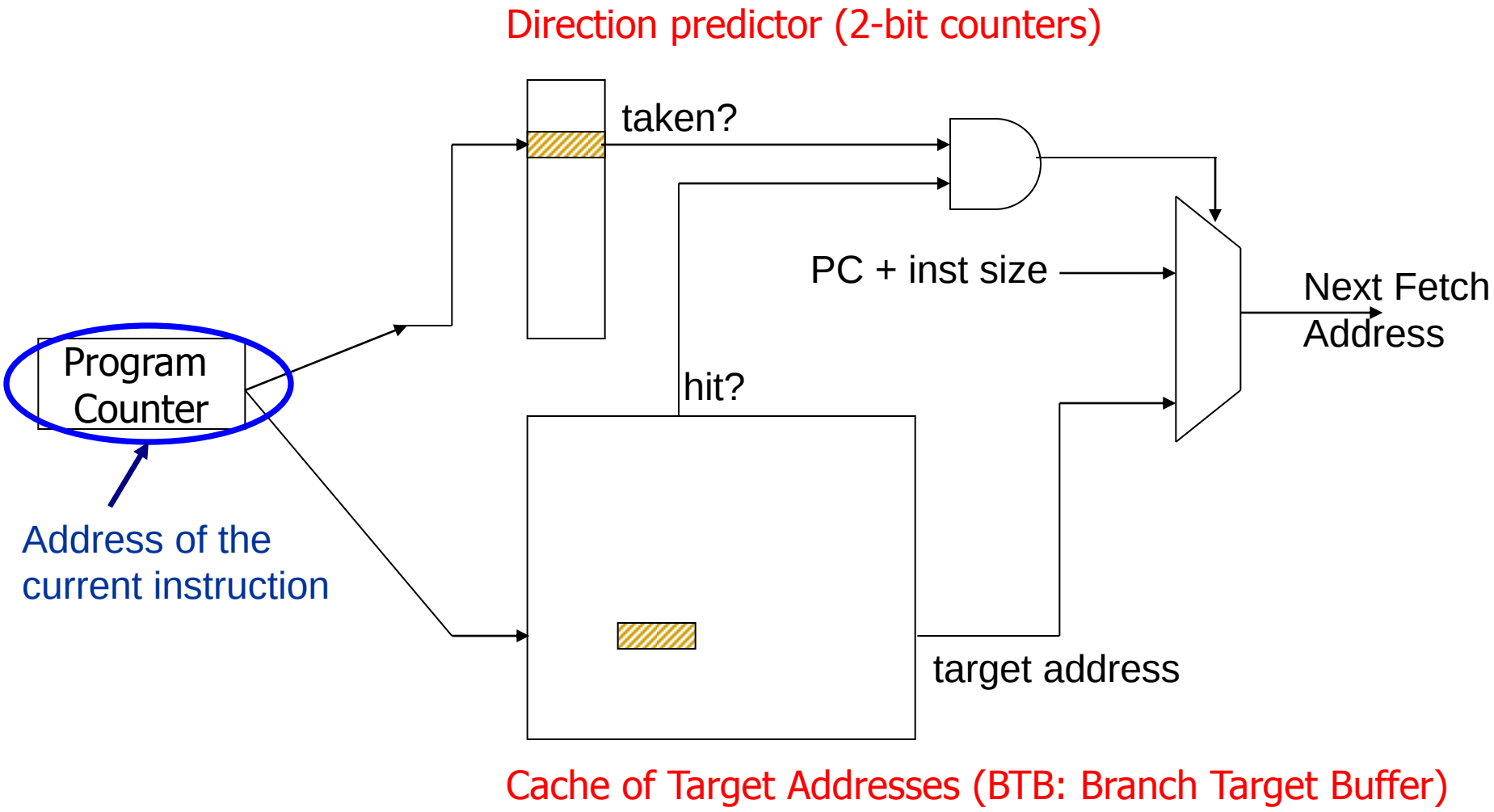
  + Better utilization of PHT

  -- Increases access latency

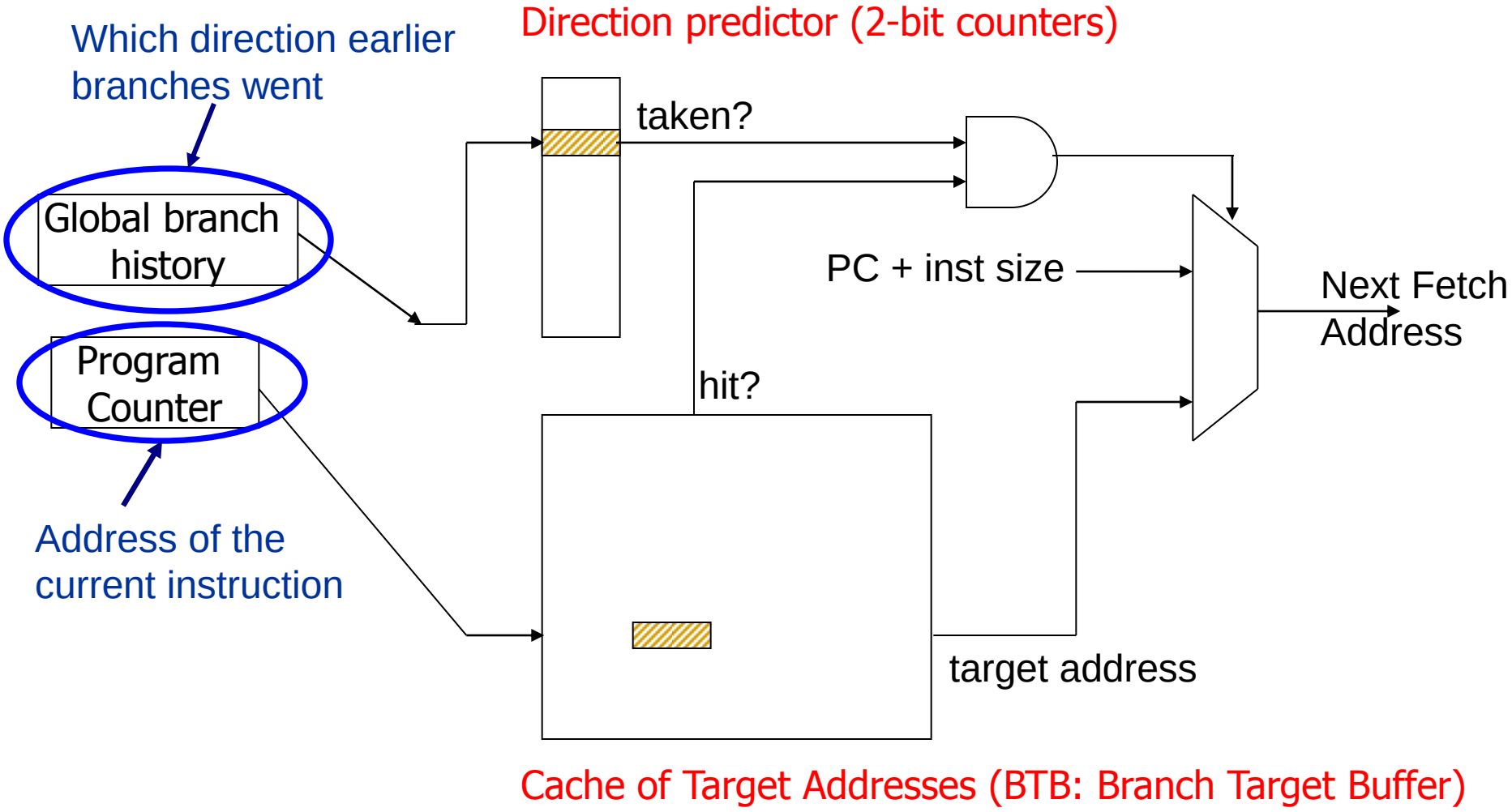


- McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

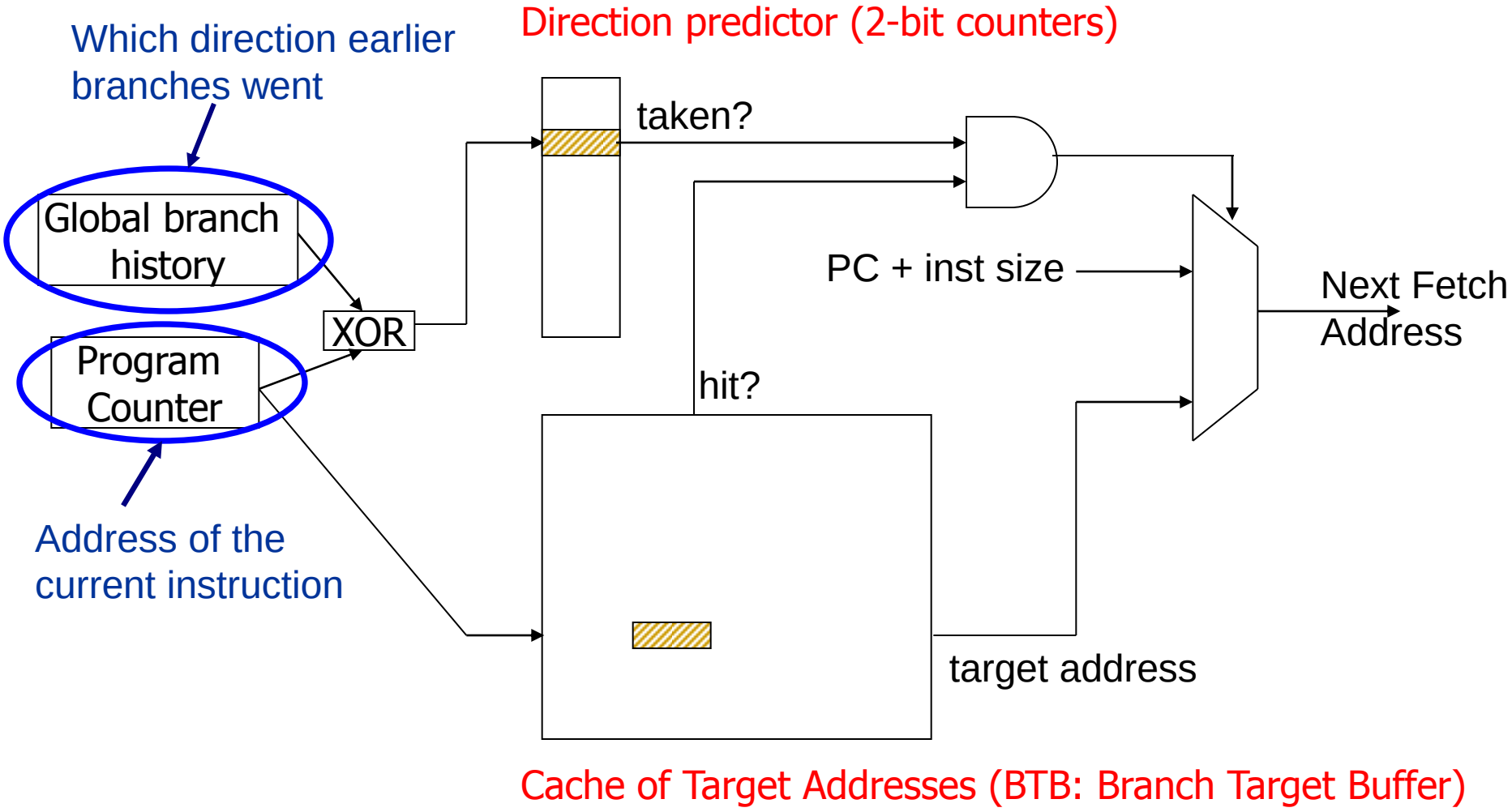
Review: One-Level Branch Predictor



Two-Level Global History Branch Predictor

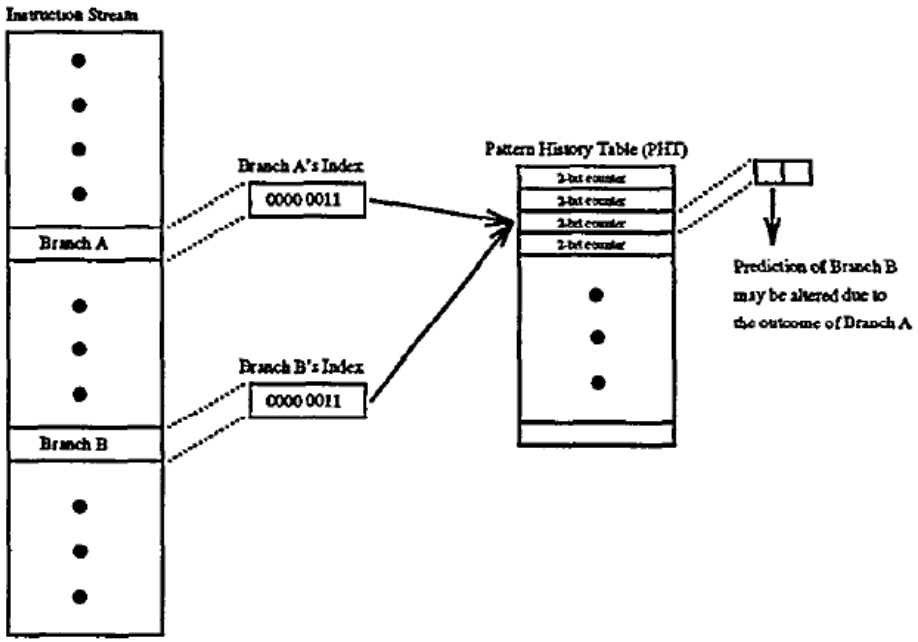


Two-Level Gshare Branch Predictor



An Issue: Interference in the PHTs

- Sharing the PHTs between histories/branches leads to interference
  - Different branches map to the same PHT entry and modify it
  - Interference can be positive, negative, or neutral



- Interference can be eliminated by dedicating a PHT per branch
  -- Too much hardware cost
- How else can you eliminate or reduce interference?

Reducing Interference in PHTs (I)

- Increase size of PHT
- Branch filtering
  - Predict highly-biased branches separately so that they do not consume PHT entries
  - E.g., static prediction or BTB based prediction
- Hashing/index-randomization
  - Gshare
  - Gskew
- Agree prediction

Biased Branches and Branch Filtering

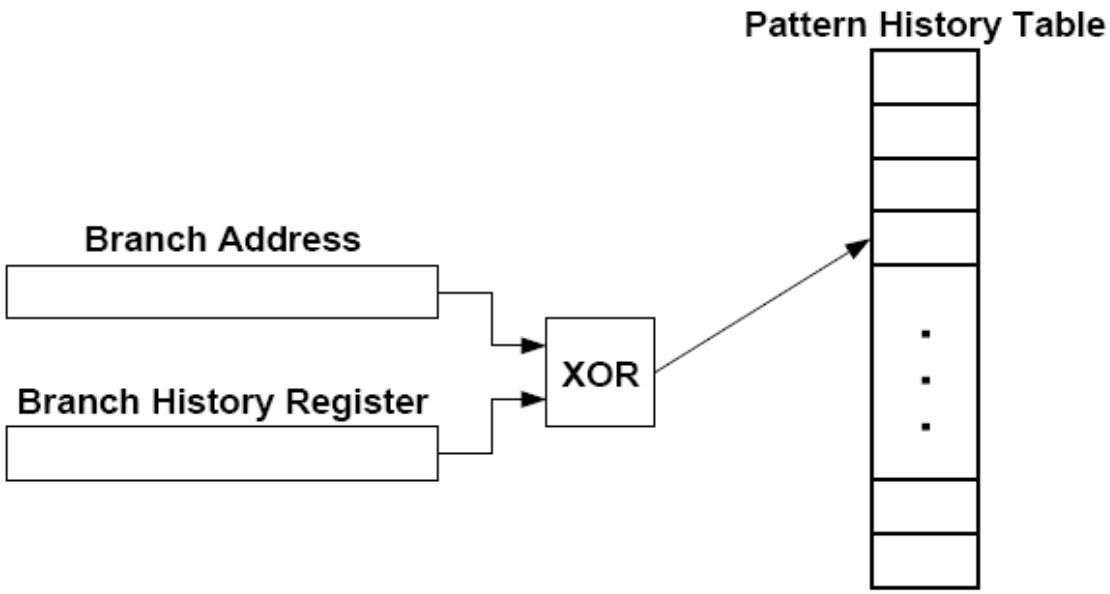
- Observation: Many branches are biased in one direction (e.g., 99% taken)
- Problem: These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing "interference" in branch prediction tables and history registers
- Solution: Detect such biased branches, and predict them with a simpler predictor (e.g., last time, static, …)
- Chang et al., "Branch classification: a new mechanism for improving branch predictor performance," MICRO 1994.

Reducing Interference: Gshare

- Idea 1: Randomize the indexing function into the PHT such that probability of two branches mapping to the same entry reduces
  - Gshare predictor: GHR hashed with the Branch PC

  + Better utilization of PHT + More context information

  - Increases access latency



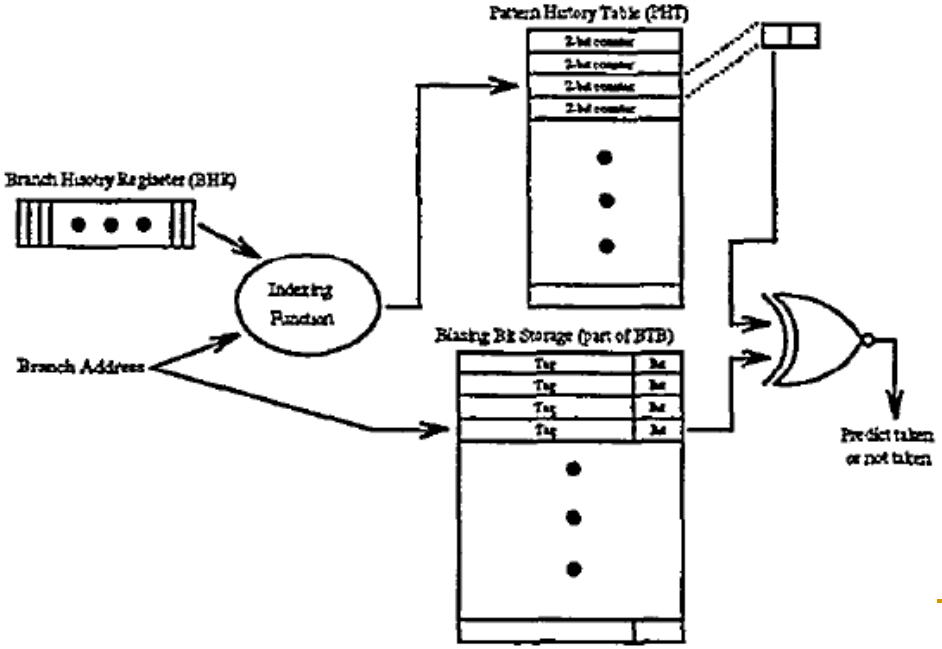
  - McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

Reducing Interference: Agree Predictor

- Idea 2: Agree prediction
  - Each branch has a "bias" bit associated with it in BTB
    - Ideally, most likely outcome for the branch
  - High bit of the PHT counter indicates whether or not the prediction agrees with the bias bit (not whether or not prediction is taken)

+ Reduces negative interference (Why???)

-- Requires determining bias bits (compiler vs. hardware)



Sprangle et al., "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference," ISCA 1997.

Why Does Agree Prediction Make Sense?

- Assume two branches have taken rates of 85% and 15%.
- Assume they conflict in the PHT

- Let's compute the probability they have opposite outcomes
  - Baseline predictor:
    - P (b1 T, b2 NT) + P (b1 NT, b2 T)
      = (85%*85%) + (15%*15%) = 74.5%
  - Agree predictor:
    - Assume bias bits are set to T (b1) and NT (b2)
    - P (b1 agree, b2 disagree) + P (b1 disagree, b2 agree)
      = (85%*15%) + (15%*85%) = 25.5%

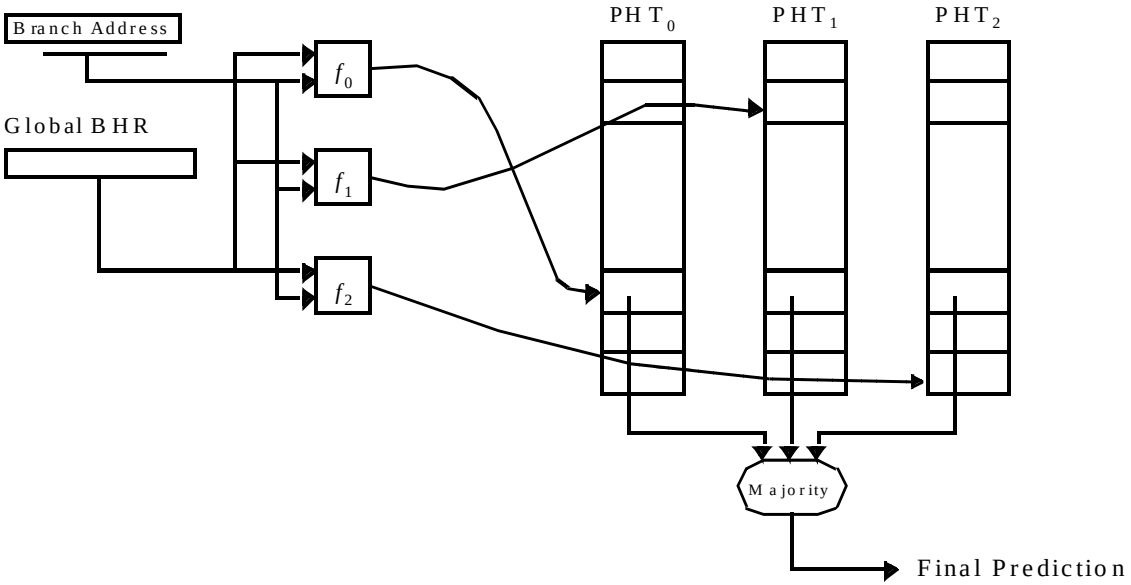
- Works because most branches are biased (not 50% taken)

Reducing Interference: Gskew

- Idea 3: Gskew predictor
  - Multiple PHTs
  - Each indexed with a different type of hash function
  - Final prediction is a majority vote

+ Distributes interference patterns in a more randomized way (interfering patterns less likely in different PHTs at the same time)

-- More complexity (due to multiple PHTs, hash functions)



Seznec, "An optimized 2bcgskew branch predictor," IRISA Tech Report 1993.

Michaud, "Trading conflict and capacity aliasing in conditional branch predictors," ISCA 1997

More Techniques to Reduce PHT Interference

- The bi-mode predictor
  - Separate PHTs for mostly-taken and mostly-not-taken branches
  - Reduces negative aliasing between them
  - Lee et al., "The bi-mode branch predictor," MICRO 1997.

- The YAGS predictor
  - Use a small tagged "cache" to predict branches that have experienced interference
  - Aims to not to mispredict them again
  - Eden and Mudge, "The YAGS branch prediction scheme," MICRO 1998.

- Alpha EV8 (21464) branch predictor
  - Seznec et al., "Design tradeoffs for the Alpha EV8 conditional branch predictor," ISCA 2002.

Can We Do Better: Two-Level Prediction

- Last-time and 2BC predictors exploit only "last-time" predictability for a given branch
- 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 (in addition to the outcome of the branch "last-time" it was executed)
  - Local branch correlation

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

Local Branch Correlation

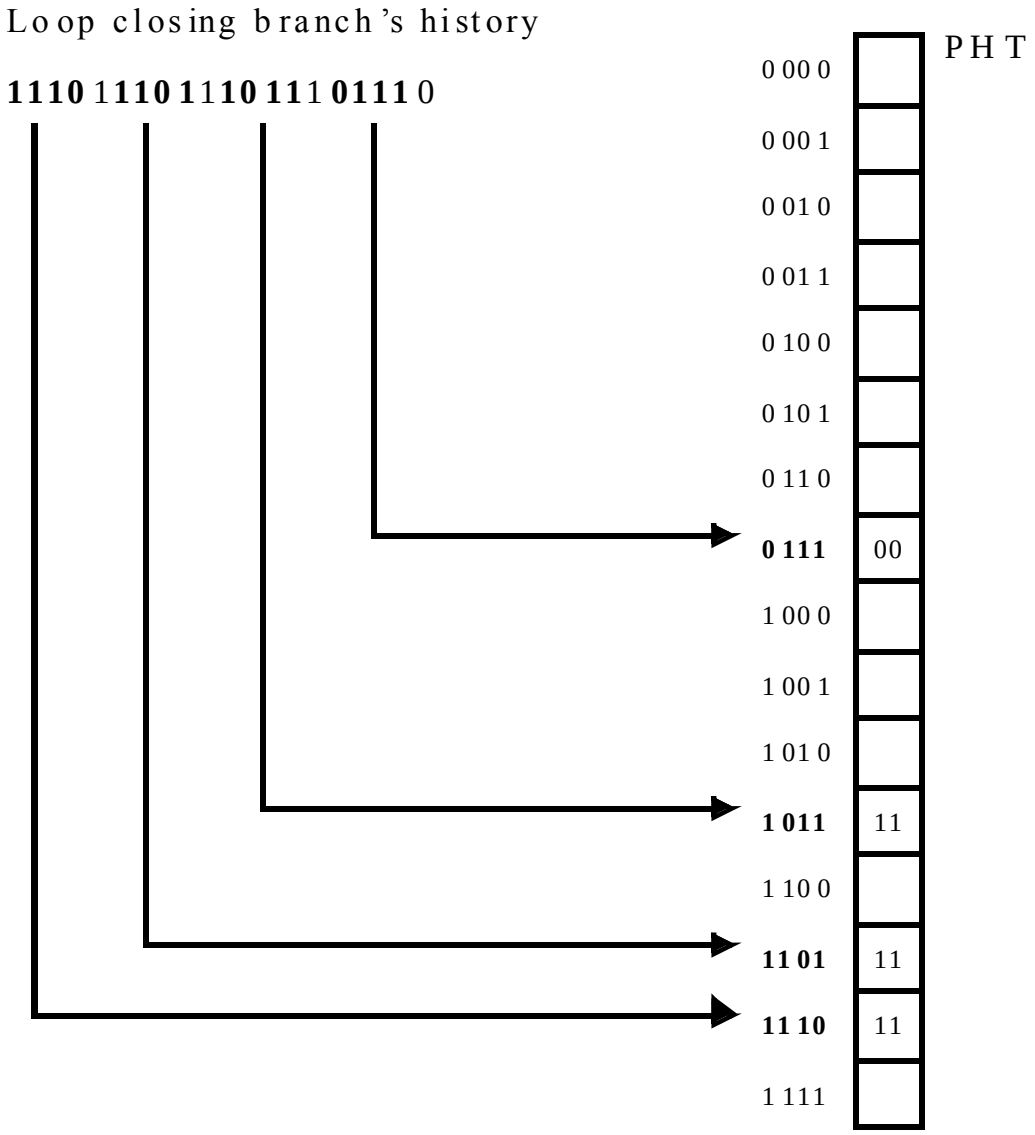
for (i=1; i<=4; i++) { }

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.

More Motivation for Local History

- To predict a loop branch "perfectly", we want to identify the last iteration of the loop
- By having a separate PHT entry for each local history, we can distinguish different iterations of a loop
- Works for "short" loops



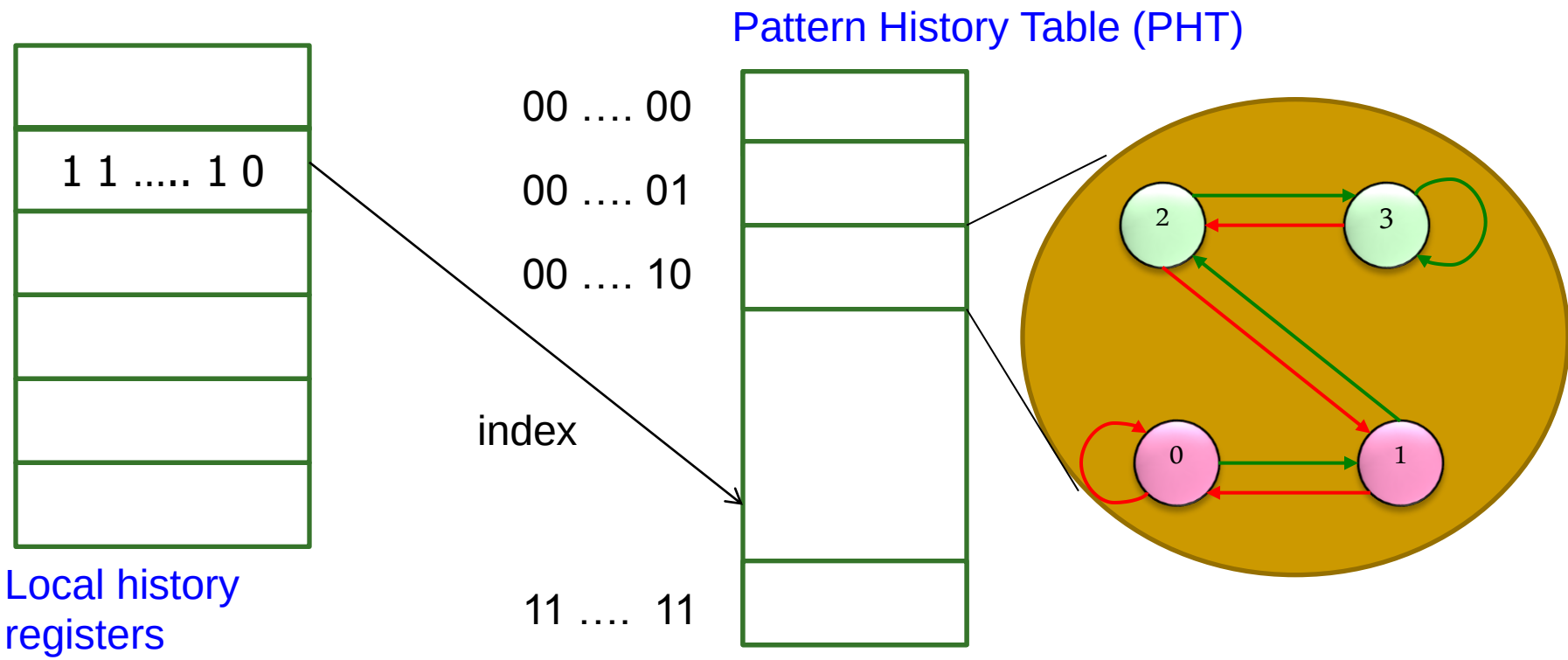
Capturing Local Branch Correlation

- 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 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)

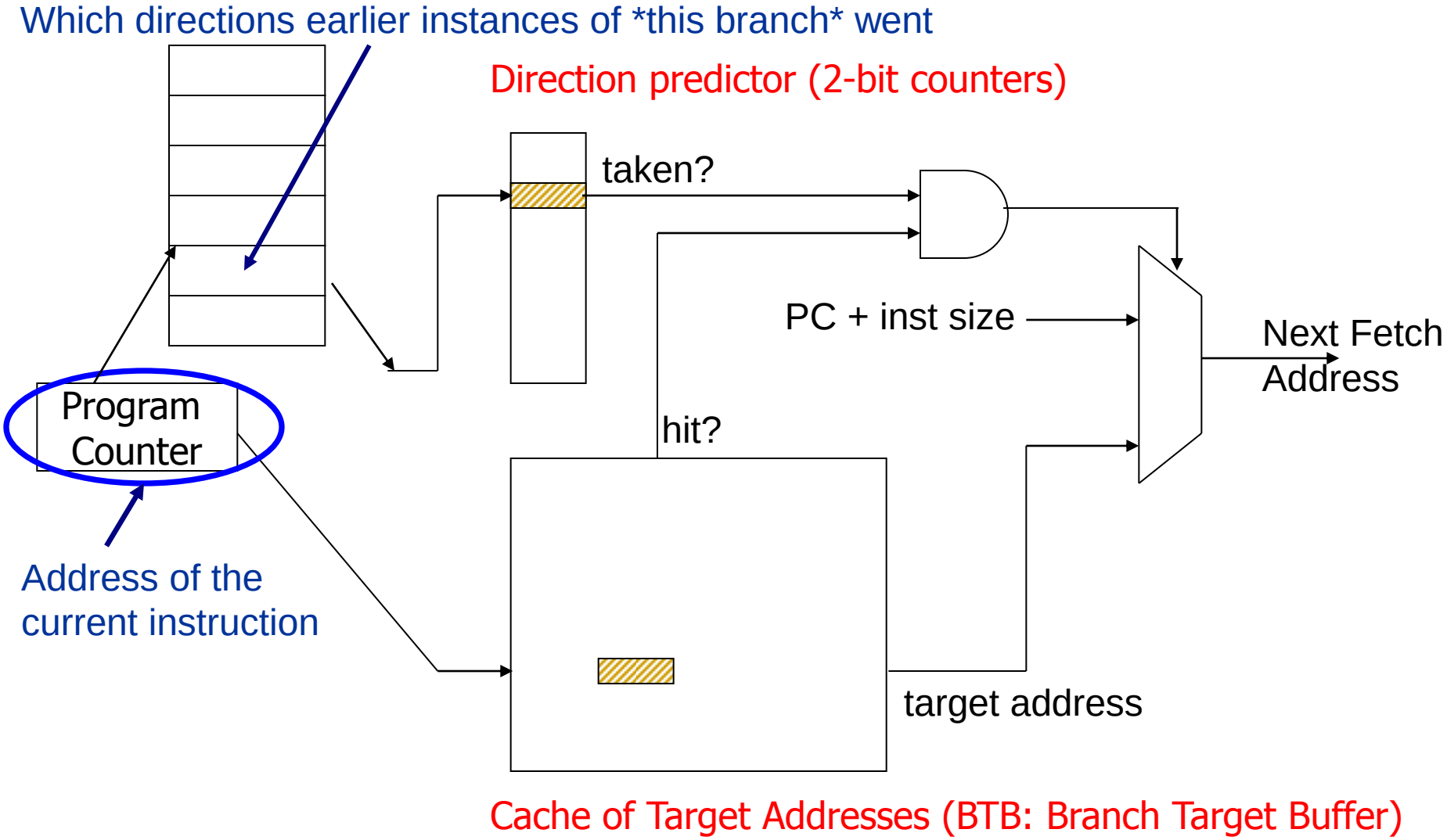
Two Level Local Branch Prediction

- 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



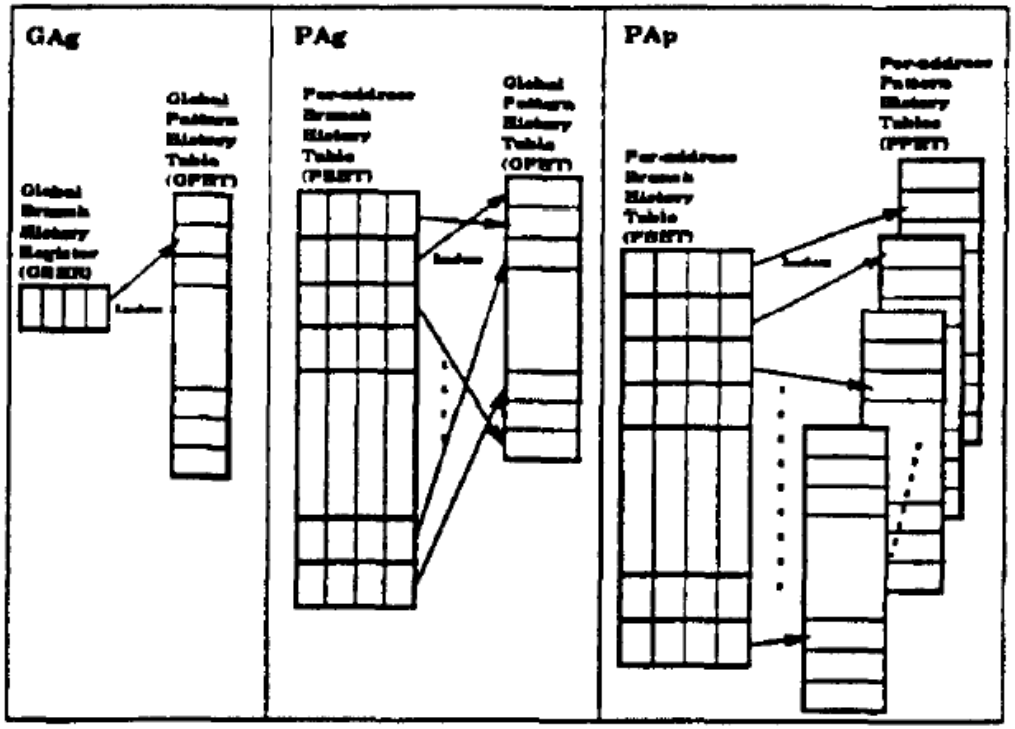
Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

Two-Level Local History Branch Predictor



Two-Level Predictor Taxonomy

- 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)



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

Can We Do Even Better?

- Predictability of branches varies

- Some branches are more predictable using local history
- Some using global
- For others, a simple two-bit counter is enough
- Yet for others, a bit is enough

- Observation: There is heterogeneity in predictability behavior of branches
  - No one-size fits all branch prediction algorithm for all branches

- Idea: Exploit that heterogeneity by designing heterogeneous branch predictors

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)

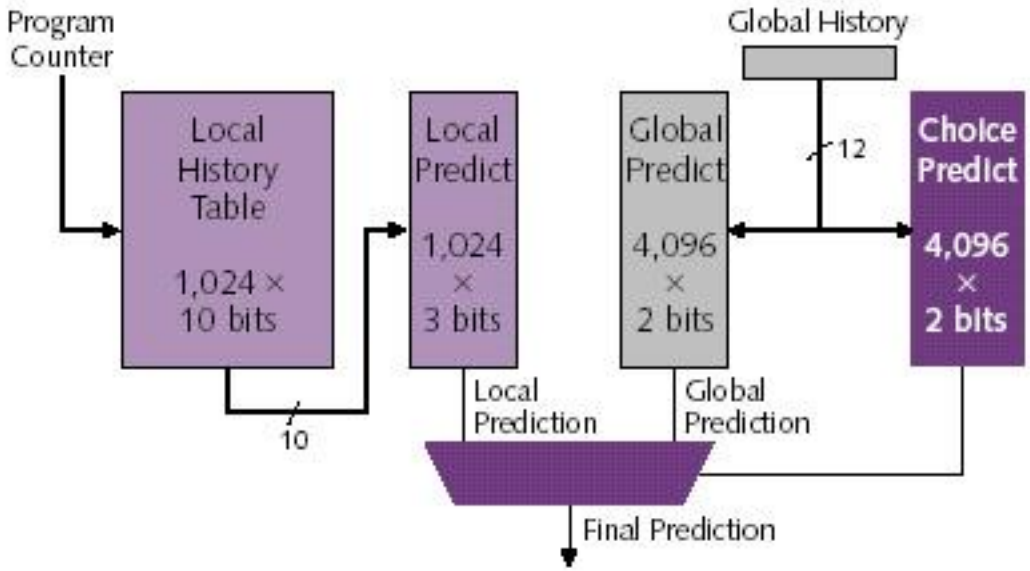
- Disadvantages:

  -- 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

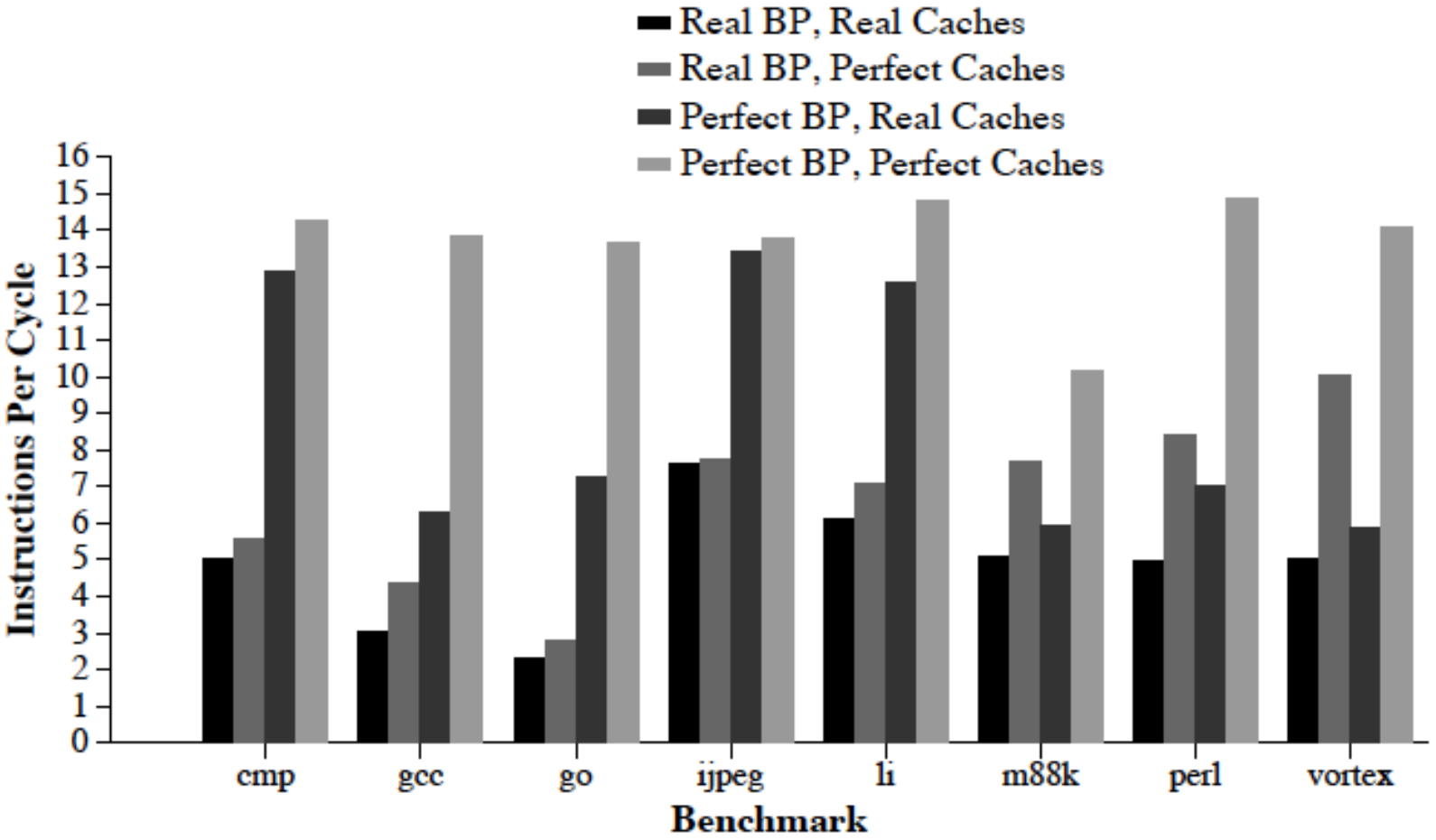
- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Are We Done w/ Branch Prediction?

- Hybrid branch predictors work well
  - E.g., 90-97% prediction accuracy on average

- Some "difficult" workloads still suffer, though!
  - E.g., gcc
  - Max IPC with tournament prediction: 9
  - Max IPC with perfect prediction: 35

Are We Done w/ Branch Prediction?



Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.

Some Other Branch Predictor Types

- Loop branch detector and predictor
  - Loop iteration count detector/predictor
  - Works well for loops with small number of iterations, where iteration count is predictable
  - Used in Intel Pentium M
- Perceptron branch predictor
  - Learns the *direction correlations* between individual branches
  - Assigns weights to correlations
  - Jimenez and Lin, "Dynamic Branch Prediction with Perceptrons," HPCA 2001.
- Hybrid history length based predictor
  - Uses different tables with different history lengths
  - Seznec, "Analysis of the O-Geometric History Length branch predictor," ISCA 2005.

Intel Pentium M Predictors

The advanced branch prediction in the Pentium M processor is based on the Intel Pentium® 4 processor's [6] branch predictor. On top of that, two additional predictors to capture special program flows, were added: a Loop Detector and an Indirect Branch Predictor.

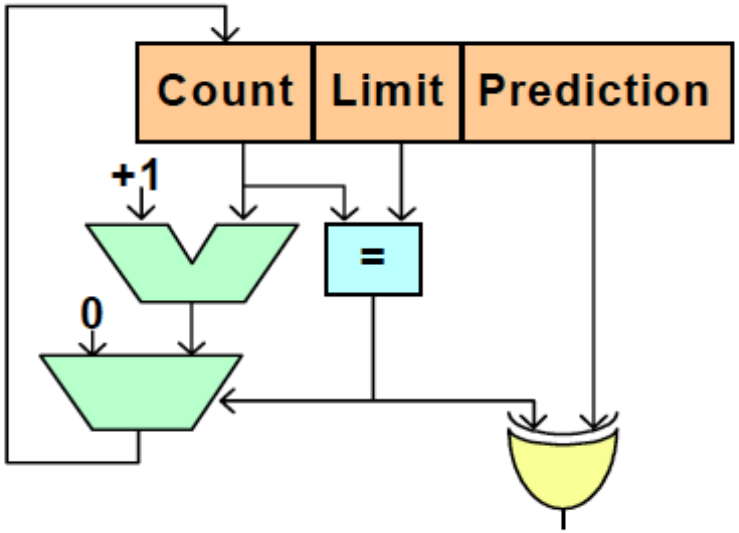


Figure 2: The Loop Detector logic

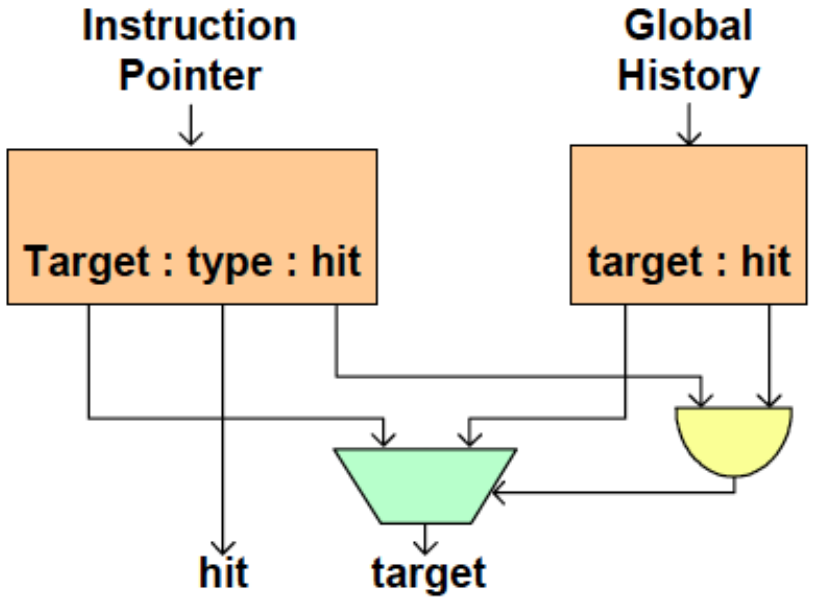
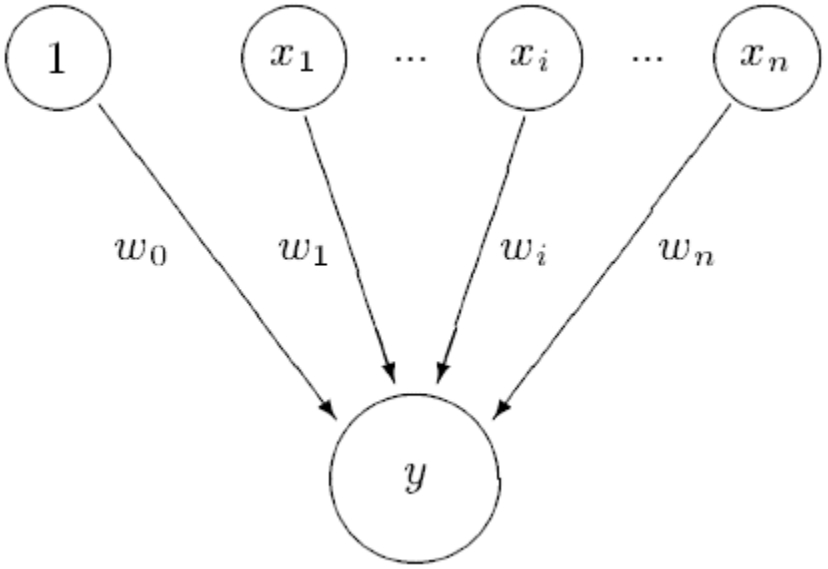


Figure 3: The Indirect Branch Predictor logic

Gochman et al.,
"The Intel Pentium M Processor: Microarchitecture and Performance,"
Intel Technology Journal, May 2003.

Perceptron Branch Predictor (I)

- Idea: Use a perceptron to learn the correlations between branch history register bits and branch outcome
- A perceptron learns a target Boolean function of N inputs



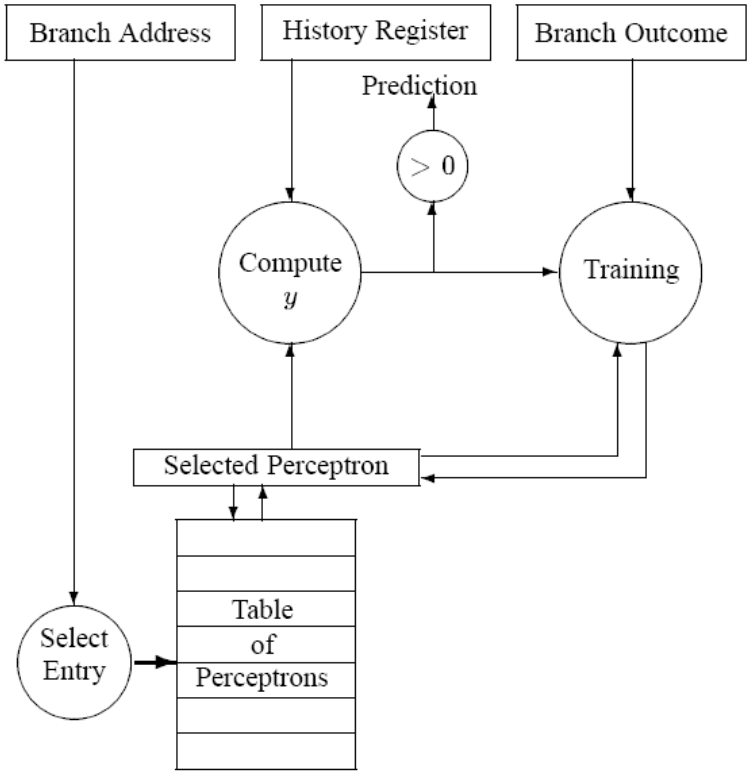
Each branch associated with a perceptron

A perceptron contains a set of weights wi
→ Each weight corresponds to a bit in the GHR
→How much the bit is correlated with the direction of the branch
→ Positive correlation: large + weight
→ Negative correlation: large - weight

Prediction:
→ Express GHR bits as 1 (T) and -1 (NT)
→ Take dot product of GHR and weights
→ If output > 0, predict taken

- Jimenez and Lin, "Dynamic Branch Prediction with Perceptrons," HPCA 2001.
- Rosenblatt, "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms," 1962

Perceptron Branch Predictor (II)



Prediction function:

Dot product of GHR and perceptron weights

$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

Output compared to 0

Bias weight (bias of branch independent of the history)

Training function:

```
if sign(y_out) ≠ t or |y_out| ≤ θ then
        for i := 0 to n do
                w_i := w_i + t x_i
        end for
end if
```

Perceptron Branch Predictor (III)

- Advantages

  + More sophisticated learning mechanism → better accuracy

- Disadvantages

  -- Hard to implement (adder tree to compute perceptron output)

  -- Can learn only linearly-separable functions

  e.g., cannot learn XOR type of correlation between 2 history bits and branch outcome

Prediction Using Multiple History Lengths

- Observation: Different branches require different history lengths for better prediction accuracy

- Idea: Have multiple PHTs indexed with GHRs with different history lengths and intelligently allocate PHT entries to different branches

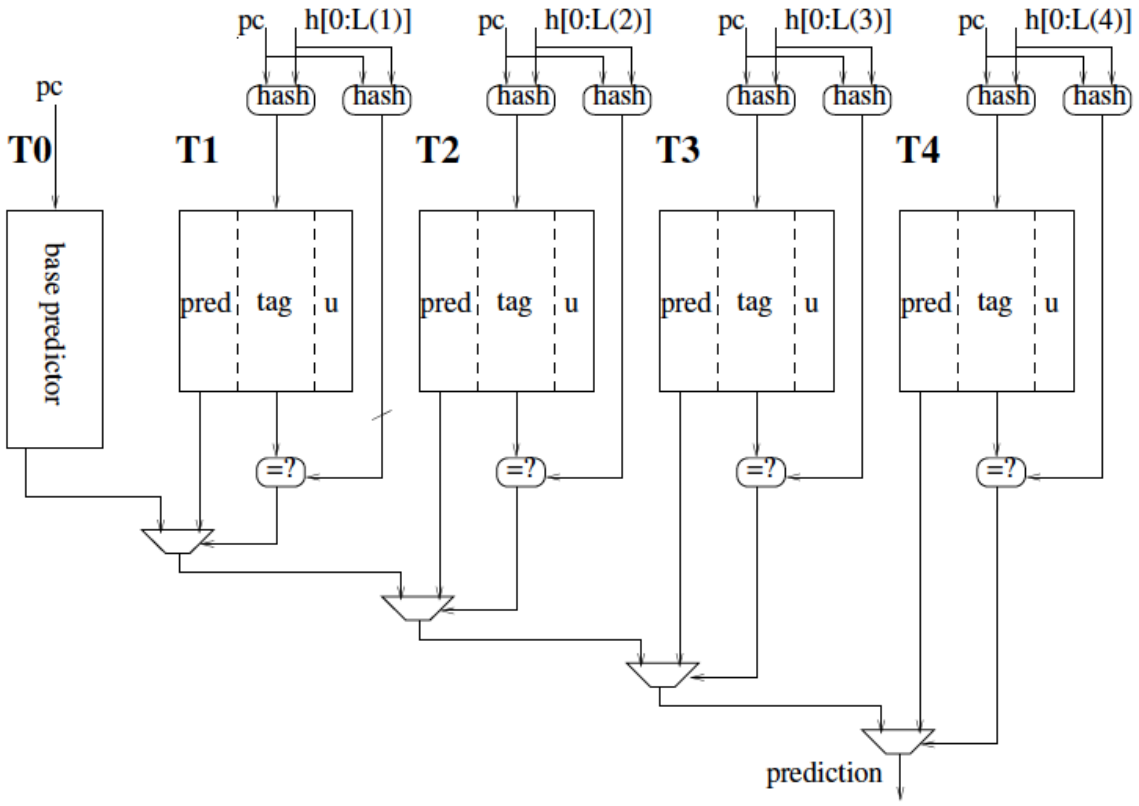
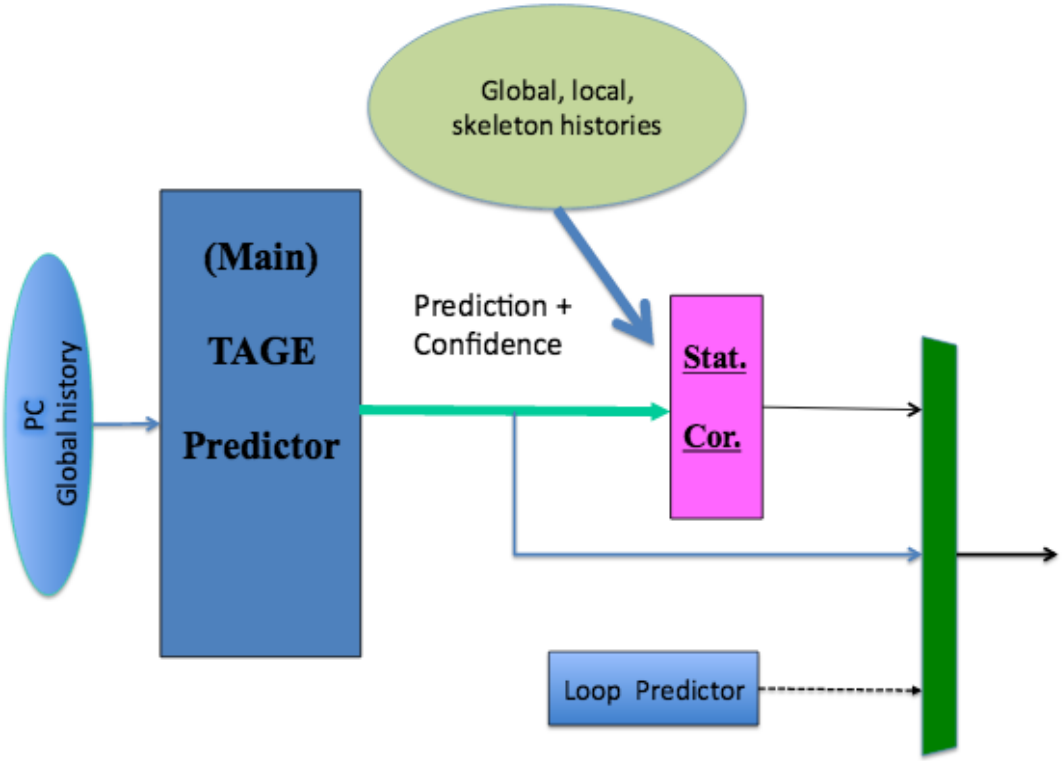


Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

Seznec and Michaud, "A case for (partially) tagged Geometric History Length Branch Prediction," JILP 2006.

State of the Art in Branch Prediction

- See the Branch Prediction Championship
  - http://www.jilp.org/cbp2014/program.html



Andre Seznec,
"TAGE-SC-L branch predictors,"
CBP 2014.

Figure 1. The TAGE-SC-L predictor: a TAGE predictor backed with a Statistical Corrector predictor and a loop predictor

Another Direction: Helper Threading

- Idea: Pre-compute the outcome of the branch with a separate, customized thread (i.e., a helper thread)

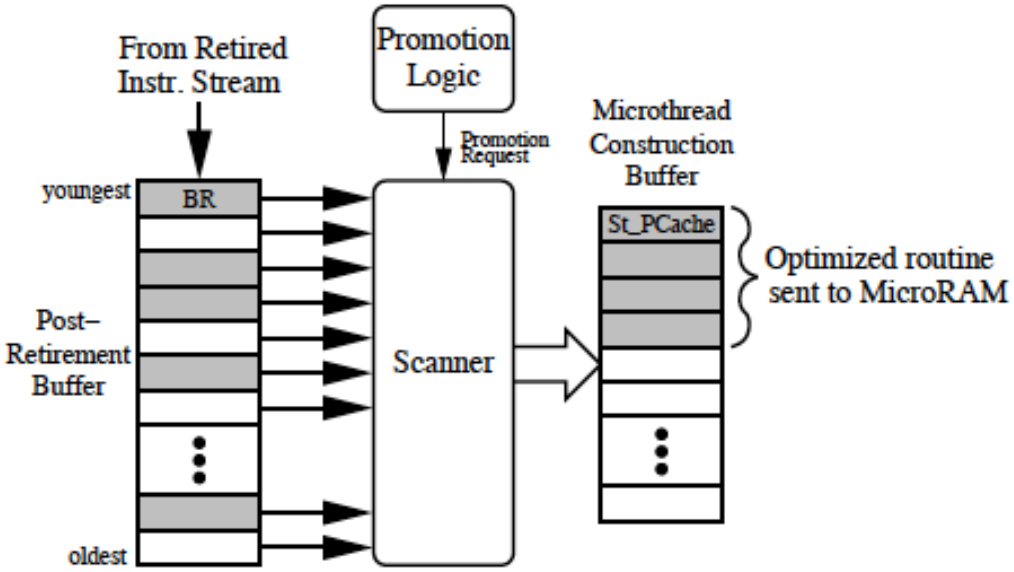


Figure 3. The Microthread Builder

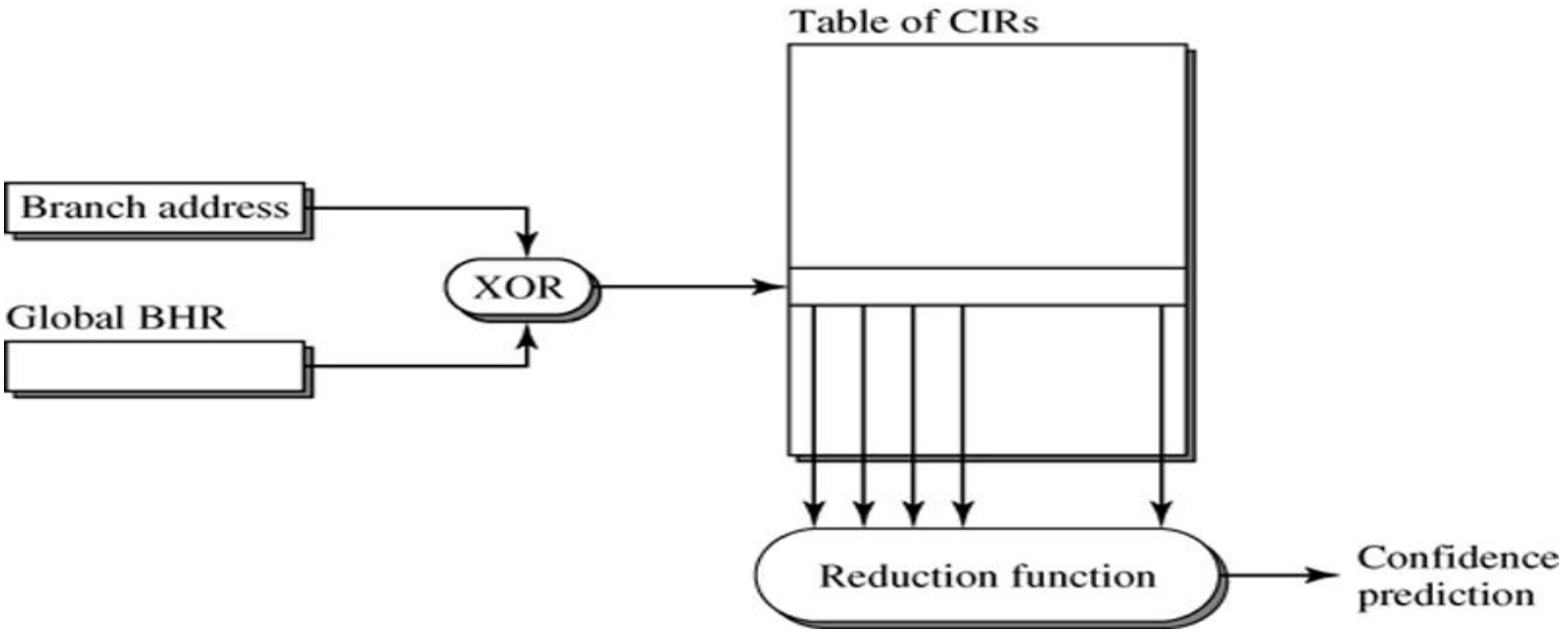
- Chappell et al., "Difficult-Path Branch Prediction Using Subordinate Microthreads," ISCA 2002.
- Chappell et al., "Simultaneous Subordinate Microthreading," ISCA 1999.

Branch Confidence Estimation

- Idea: Estimate if the prediction is likely to be correct
  - i.e., estimate how "confident" you are in the prediction
- Why?
  - Could be very useful in deciding how to speculate:
    - What predictor/PHT to choose/use
    - Whether to keep fetching on this path
    - Whether to switch to some other way of handling the branch, e.g. dual-path execution (eager execution) or dynamic predication
    - …
- Jacobsen et al., "Assigning Confidence to Conditional Branch Predictions," MICRO 1996.

How to Estimate Confidence

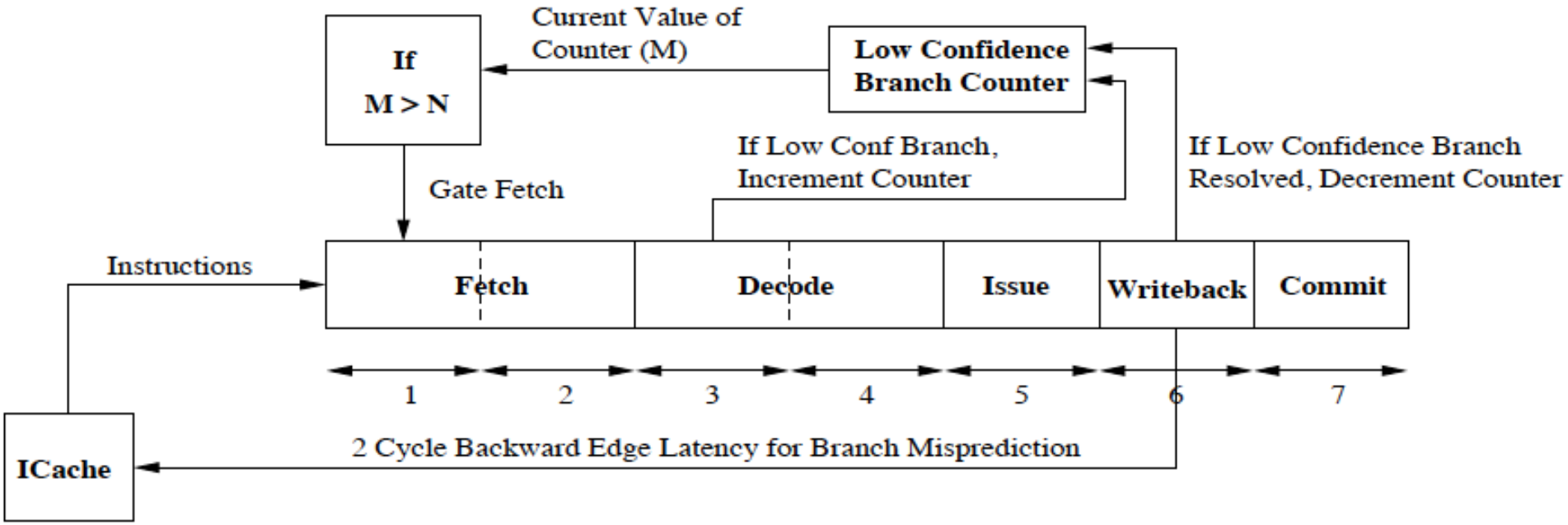
- An example estimator:
  - Keep a record of correct/incorrect outcomes for the past N instances of the "branch"
  - Based on the correct/incorrect patterns, guess if the curent prediction will likely be correct/incorrect



Jacobsen et al., "Assigning Confidence to Conditional Branch Predictions," MICRO 1996.

What to Do With Confidence Estimation?

- An example application: Pipeline Gating



Manne et al., "Pipeline Gating: Speculation Control for Energy Reduction," ISCA 1998.

Issues in Fast & Wide Fetch Engines

I-Cache Line and Way Prediction

- Problem: Complex branch prediction can take too long (many cycles)
- Goal
  - Quickly generate (a reasonably accurate) next fetch address
  - Enable the fetch engine to run at high frequencies
  - Override the quick prediction with more sophisticated prediction
- Idea: Get the predicted next cache line and way at the time you fetch the current cache line

- Example Mechanism (e.g., Alpha 21264)
  - Each cache line tells which line/way to fetch next (prediction)
  - On a fill, line/way predictor points to next sequential line
  - On branch resolution, line/way predictor is updated
  - If line/way prediction is incorrect, one cycle is wasted

Alpha 21264 Line & Way Prediction

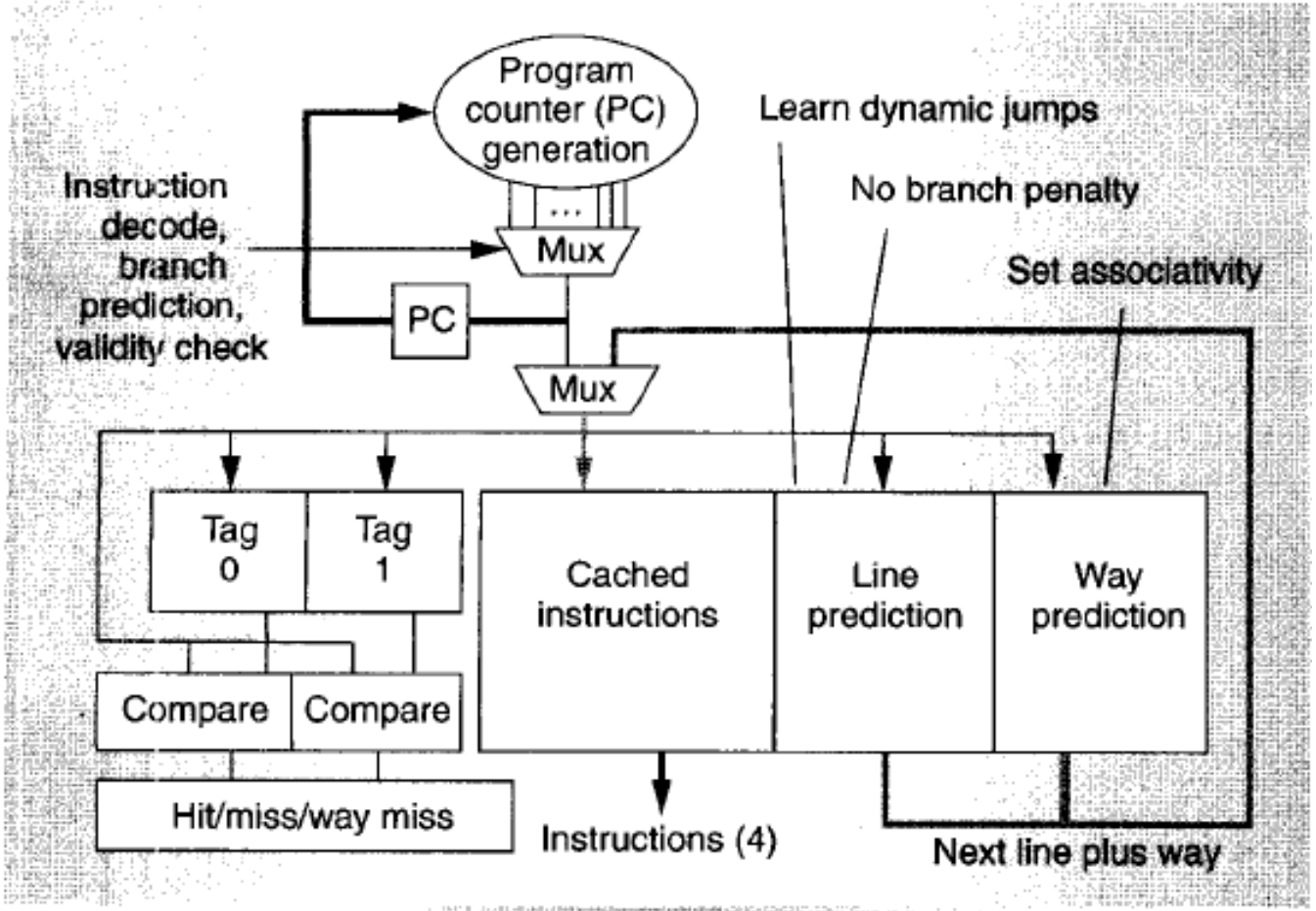
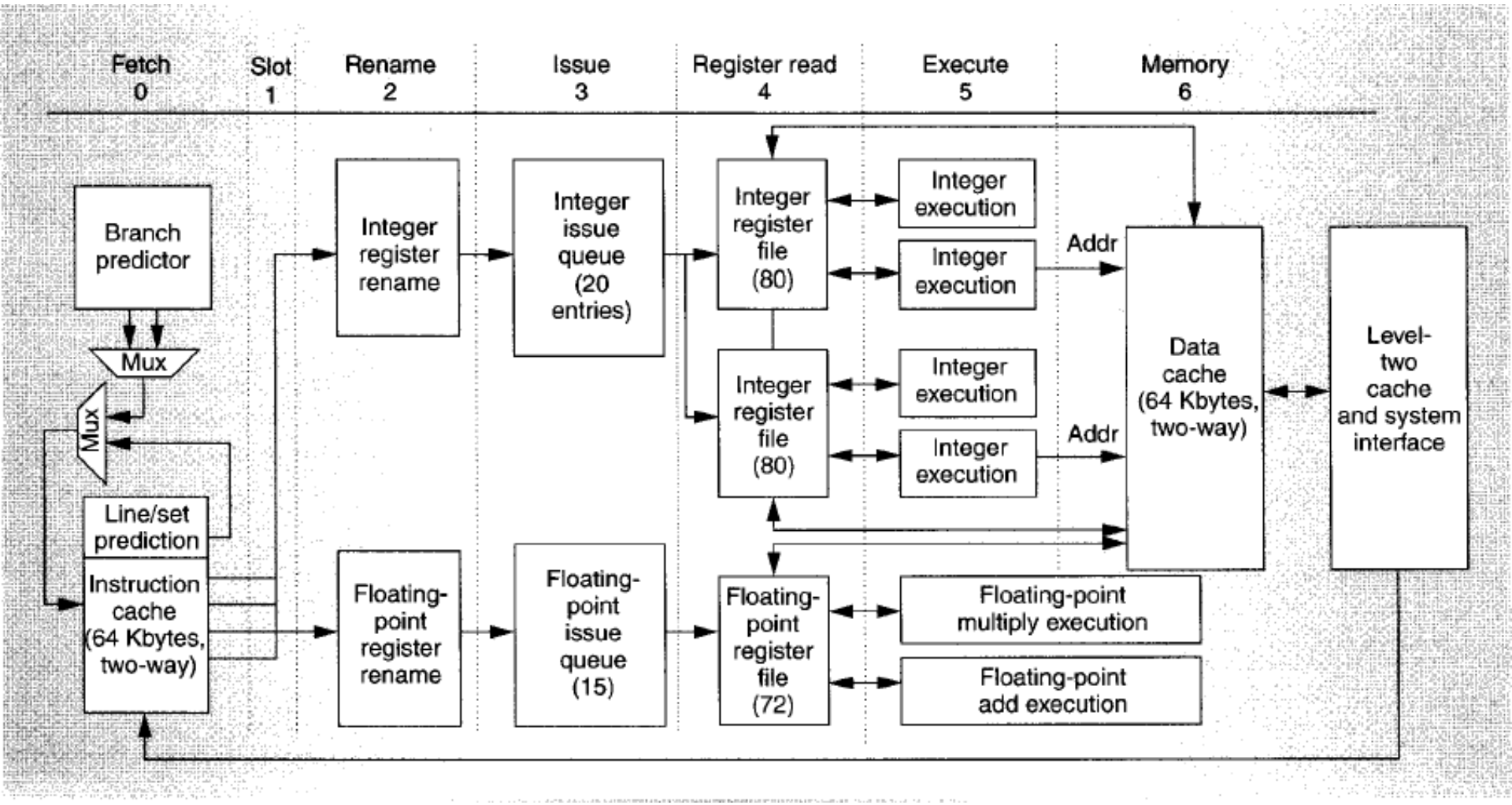


Figure 3. Alpha 21264 instruction fetch. The line and way prediction (wrap-around path on the right side) provides a fast instruction fetch path that avoids common fetch stalls when the predictions are correct.

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

Alpha 21264 Line & Way Prediction



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

Issues in Wide Fetch Engines

- Wide Fetch: Fetch multiple instructions per cycle

- Superscalar
- VLIW
- SIMT (GPUs' single-instruction multiple thread model)

- Wide fetch engines suffer from the branch problem:
  - How do you feed the wide pipeline with useful instructions in a single cycle?
  - What if there is a taken branch in the "fetch packet"?
  - What is there are "multiple (taken) branches" in the "fetch packet"?

Fetching Multiple Instructions Per Cycle

- Two problems

1. Alignment of instructions in I-cache
   - What if there are not enough (N) instructions in the cache line to supply the fetch width?

2. Fetch break: Branches present in the fetch block
   - Fetching sequential instructions in a single cycle is easy
   - What if there is a control flow instruction in the N instructions?
   - Problem: The direction of the branch is not known but we need to fetch more instructions

- These can cause effective fetch width < peak fetch width

Wide Fetch Solutions: Alignment

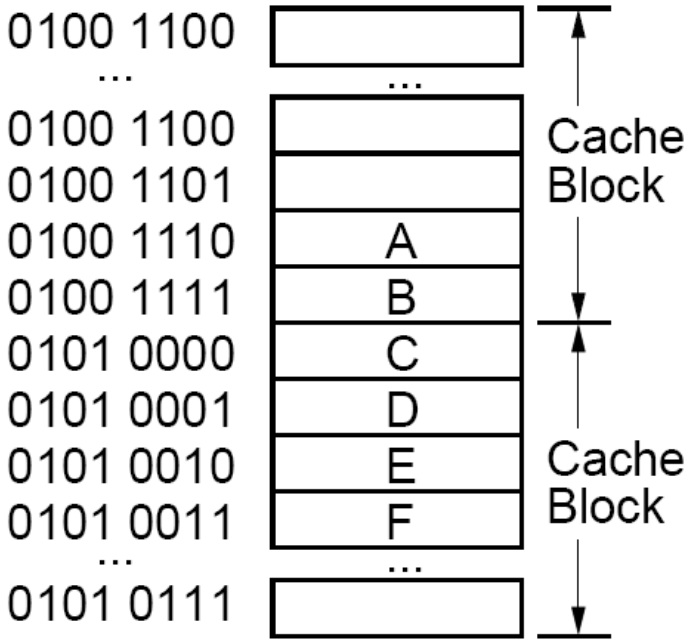
- Large cache blocks: Hope N instructions contained in the block

- Split-line fetch: If address falls into second half of the cache block, fetch the first half of next cache block as well
  - Enabled by banking of the cache
  - Allows sequential fetch across cache blocks in one cycle
  - Intel Pentium and AMD K5

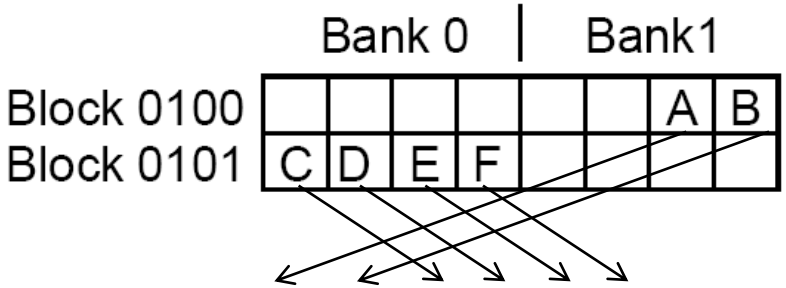
Split Line Fetch

Cache Banking

Memory Map

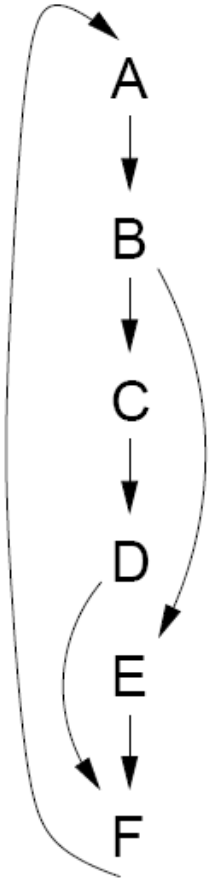


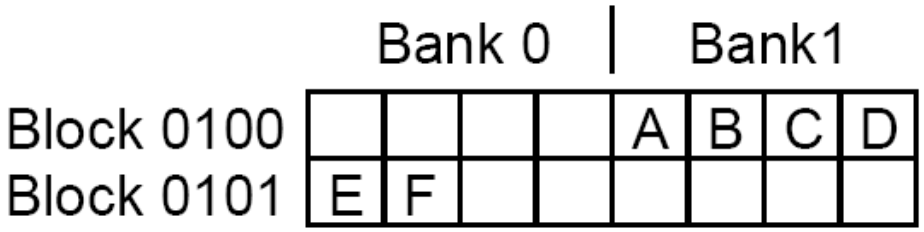
Cache



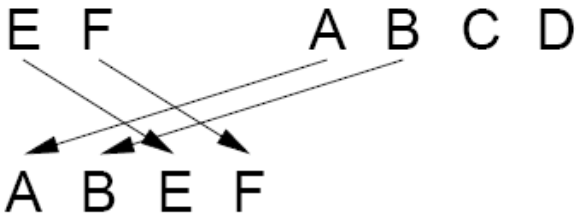
Need alignment logic:

Short Distance Predicted-Taken Branches

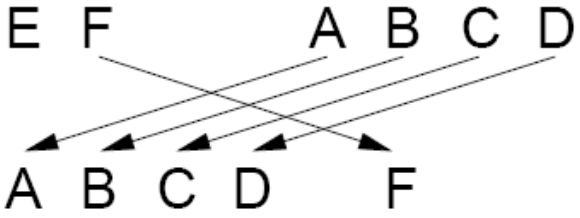




First Iteration (Branch B taken to E)



Second Iteration (Branch B fall through to C)

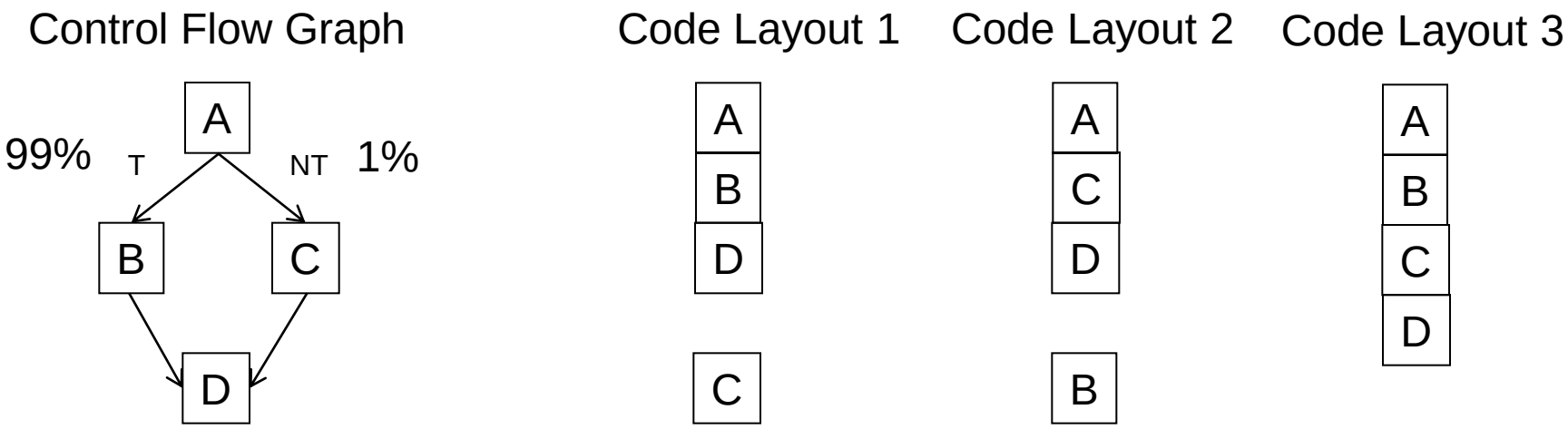


Techniques to Reduce Fetch Breaks

- Compiler
  - Code reordering (basic block reordering)
  - Superblock
- Hardware
  - Trace cache
- Hardware/software cooperative
  - Block structured ISA

Basic Block Reordering

- Not-taken control flow instructions not a problem: no fetch break: make the likely path the not-taken path
- Idea: Convert taken branches to not-taken ones
  - i.e., reorder basic blocks (after profiling)
  - Basic block: code with a single entry and single exit point



- Code Layout 1 leads to the fewest fetch breaks

Basic Block Reordering

- Pettis and Hansen, "Profile Guided Code Positioning," PLDI 1990.

- Advantages:
  + Reduced fetch breaks (assuming profile behavior matches runtime behavior of branches)
  + Increased I-cache hit rate
  + Reduced page faults

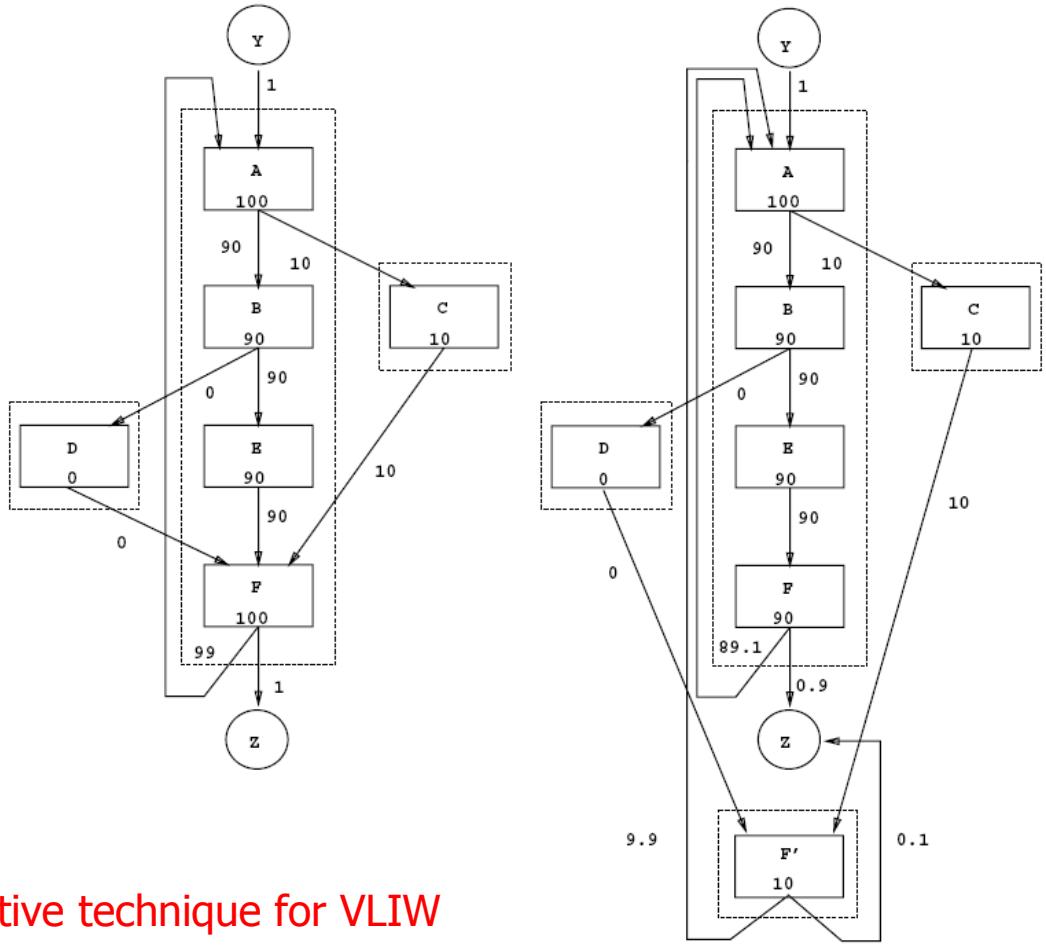
- Disadvantages:
  -- Dependent on compile-time profiling
  -- Does not help if branches are not biased
  -- Requires recompilation

Superblock

- Idea: Combine frequently executed basic blocks such that they form a single-entry multiple exit larger block, which is likely executed as straight-line code

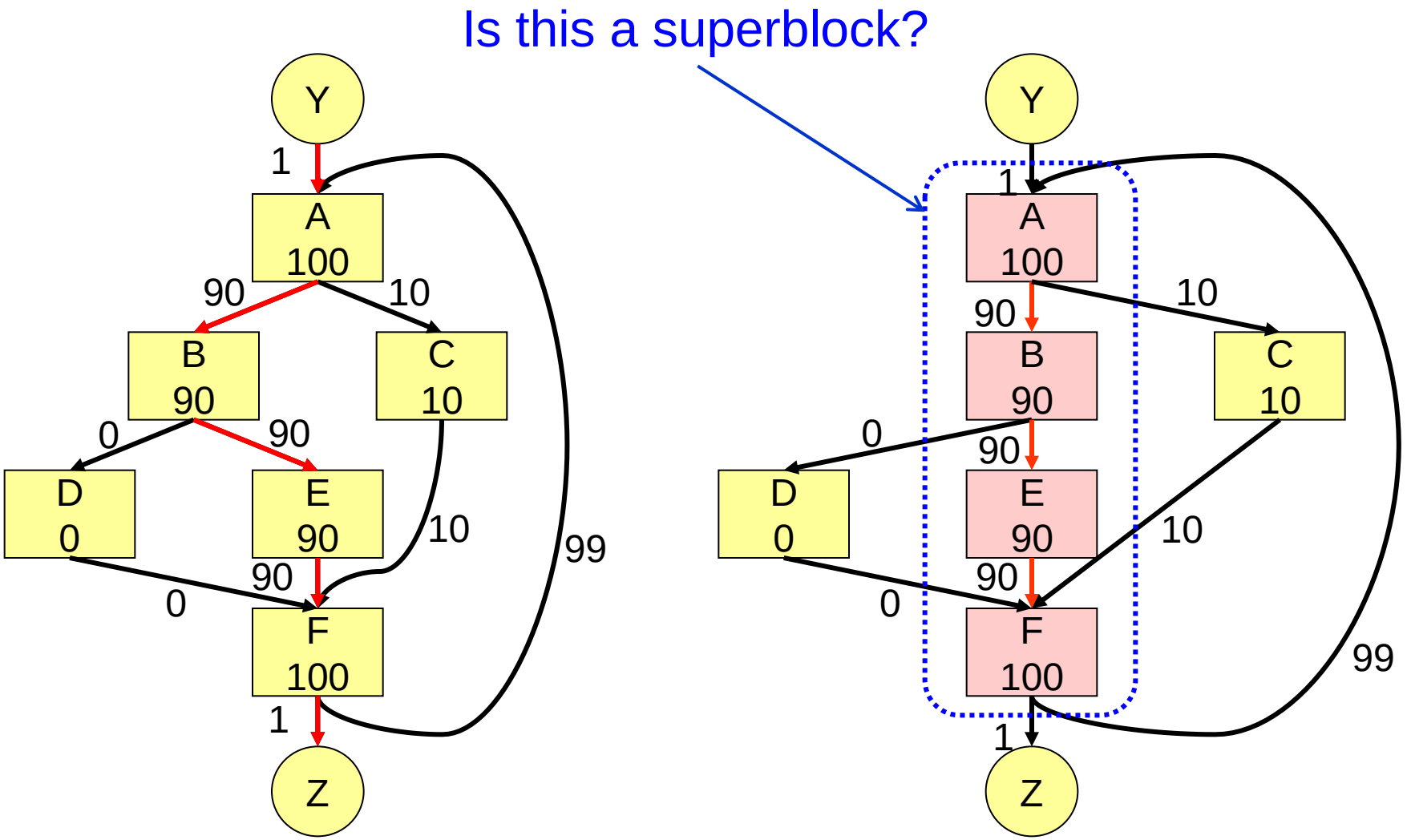
+ Helps wide fetch
+ Enables aggressive compiler optimizations and code reordering within the superblock

-- Increased code size
-- Profile dependent
-- Requires recompilation

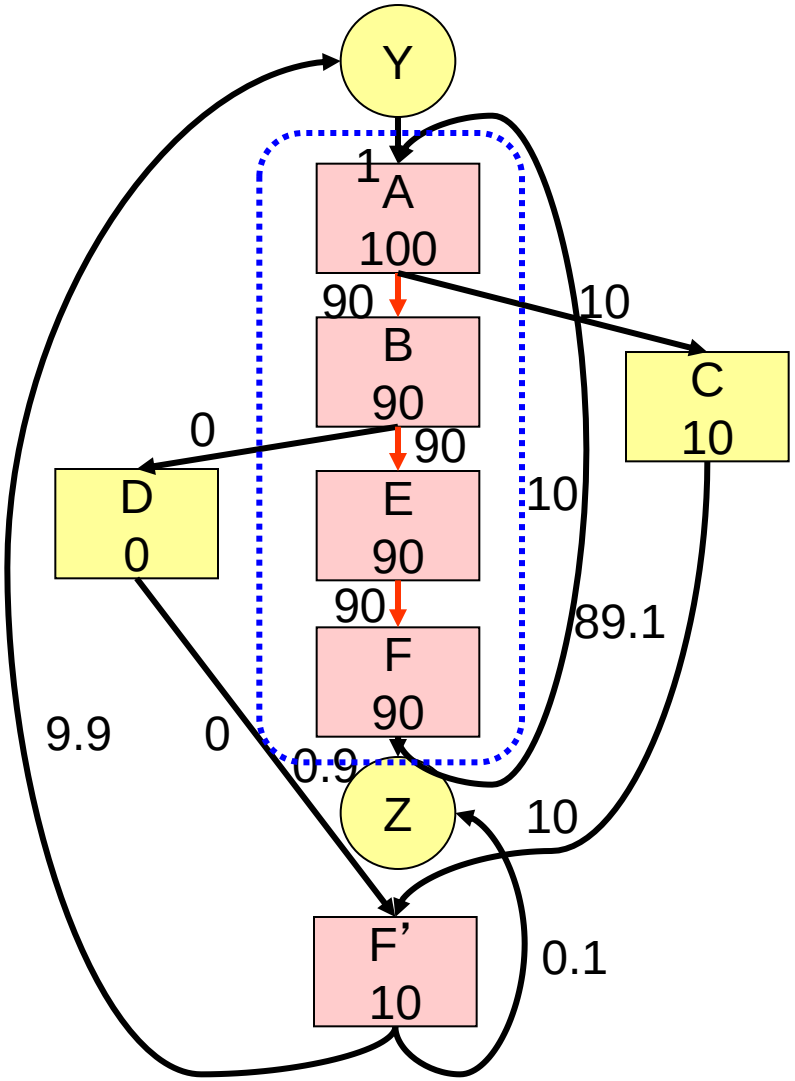


- Hwu et al. "The Superblock: An effective technique for VLIW and superscalar compilation," Journal of Supercomputing, 1993.

Superblock Formation (I)

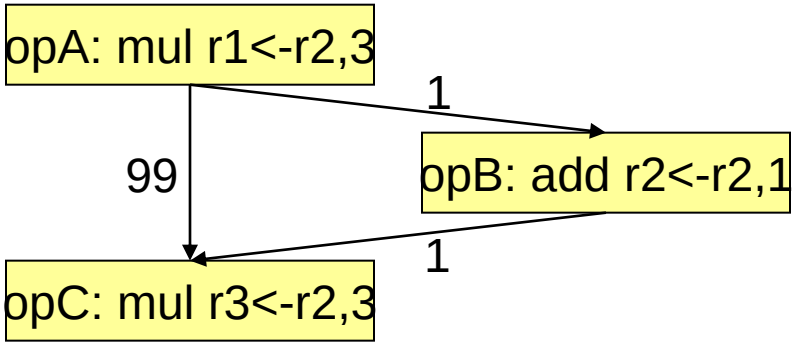


Superblock Formation (II)

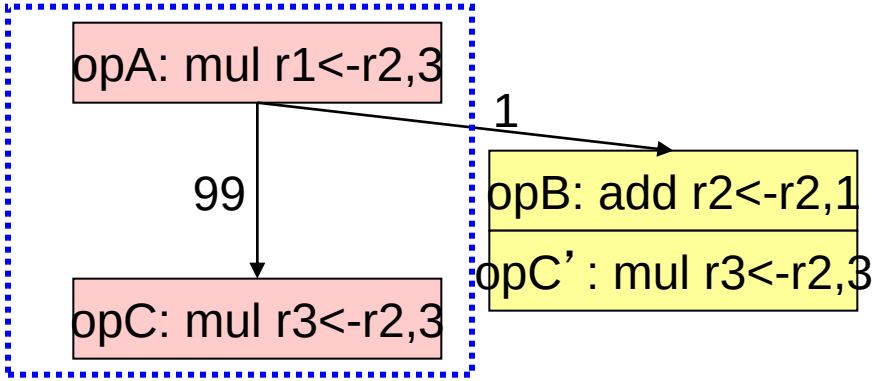


Tail duplication:
duplication of basic blocks after a side entrance to eliminate side entrances
→ transforms
a trace into a superblock.

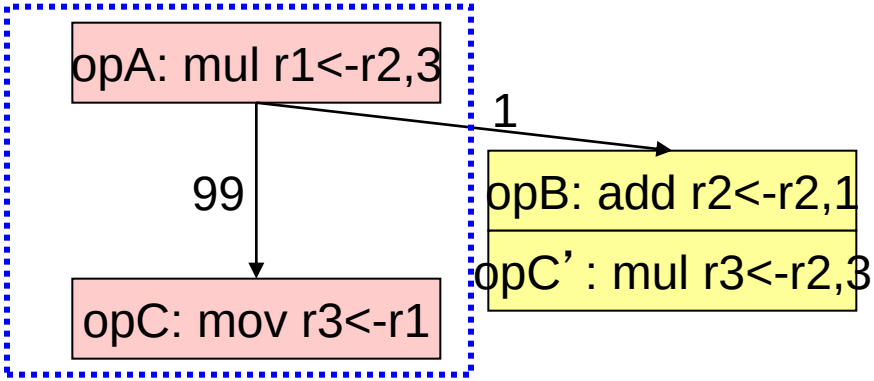
Superblock Code Optimization Example



Original Code



Code After Superblock Formation



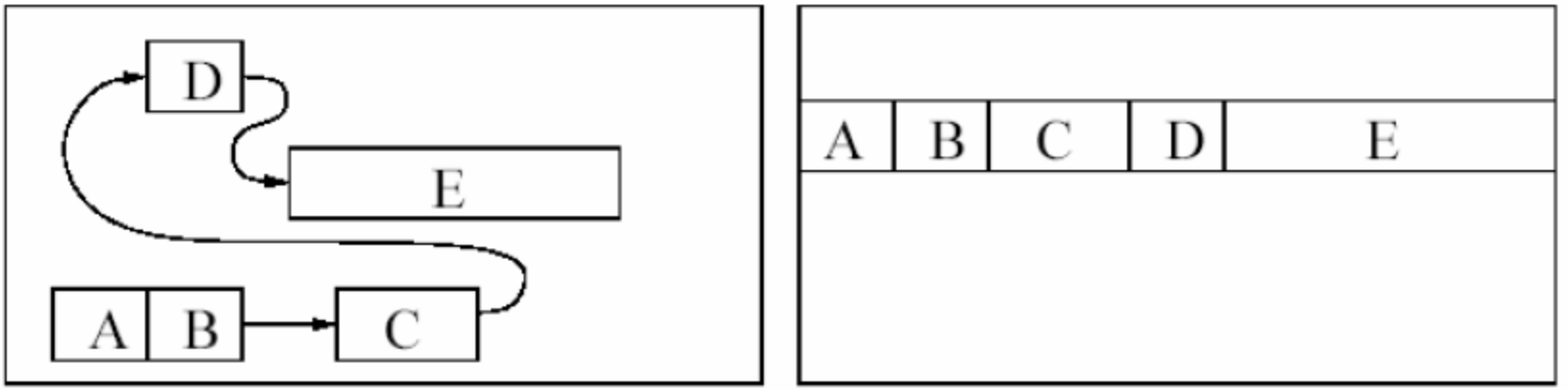
Code After Common Subexpression Elimination

Techniques to Reduce Fetch Breaks

- Compiler
  - Code reordering (basic block reordering)
  - Superblock
- Hardware
  - Trace cache
- Hardware/software cooperative
  - Block structured ISA

Trace Cache: Basic Idea

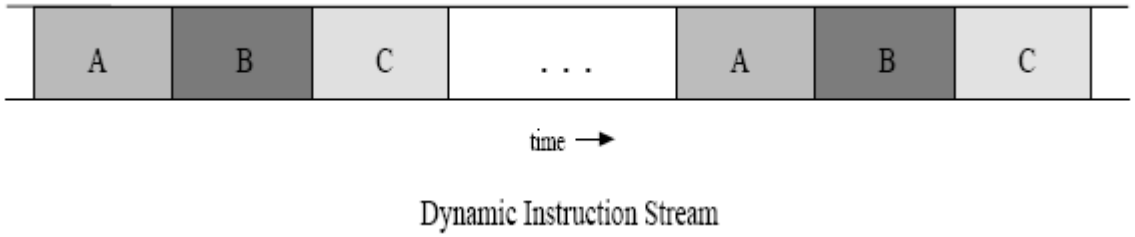
- A trace is a sequence of executed instructions.
- It is specified by a start address and the branch outcomes of control transfer instructions.
- Traces repeat: programs have frequently executed paths
- Trace cache idea: Store the dynamic instruction sequence in the same physical location.



(a) Instruction cache. (b) Trace cache.

Reducing Fetch Breaks: Trace Cache

- Dynamically determine the basic blocks that are executed consecutively
- Trace: Consecutively executed basic blocks
- Idea: Store consecutively-executed basic blocks in physically-contiguous internal storage (called trace cache)

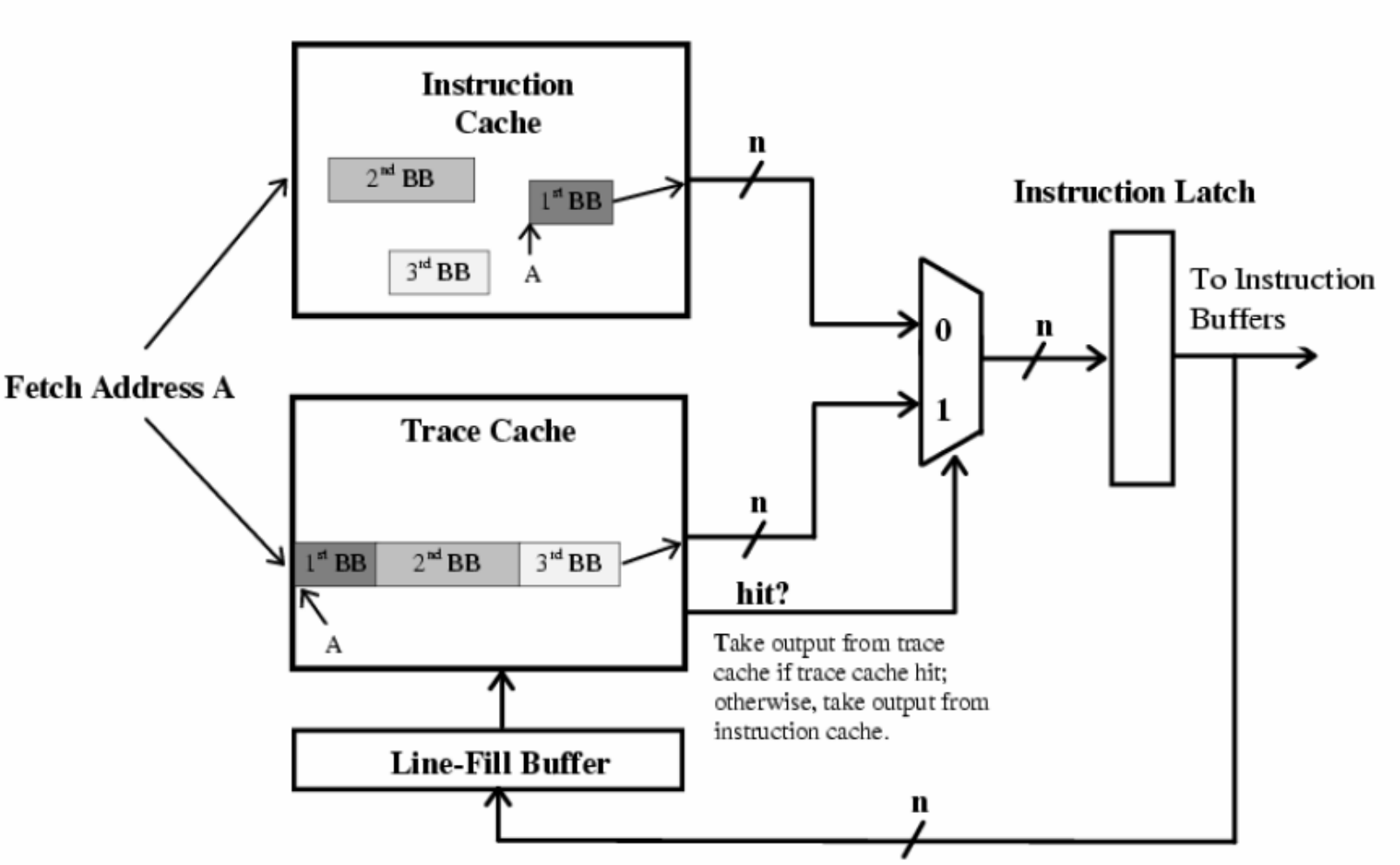


Dynamic Instruction Stream

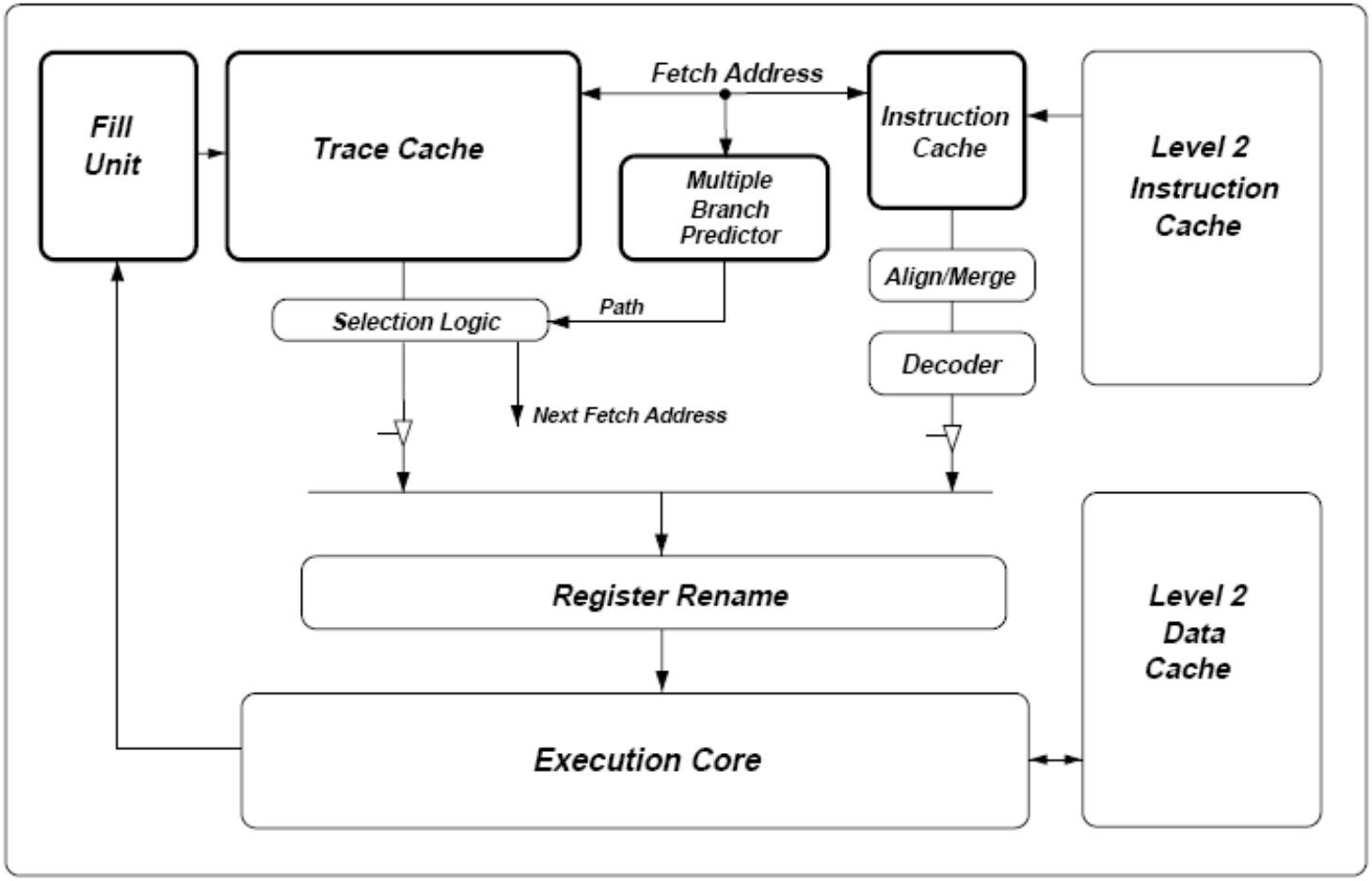
- Basic trace cache operation:
  - Fetch from consecutively-stored basic blocks (predict next trace or branches)
  - Verify the executed branch directions with the stored ones
  - If mismatch, flush the remaining portion of the trace

- Rotenberg et al., "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," MICRO 1996.
- Patel et al., "Critical Issues Regarding the Trace Cache Fetch Mechanism," Umich TR, 1997.

Trace Cache: Example



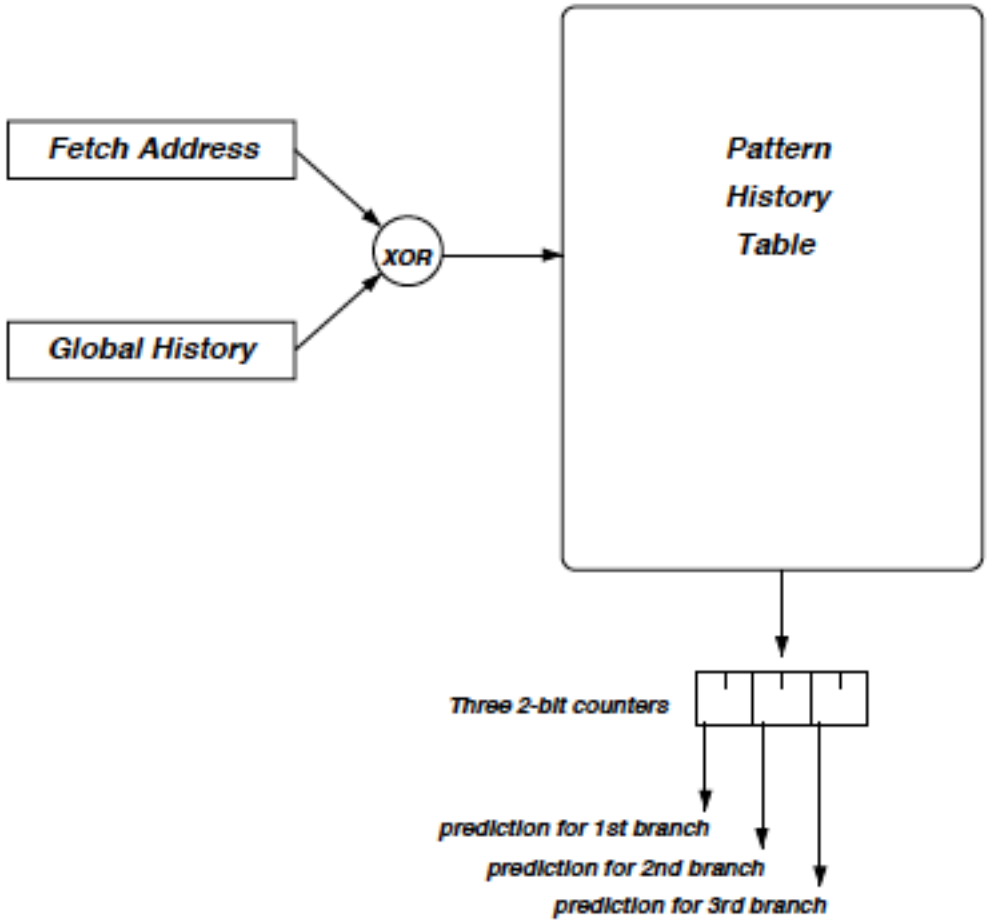
An Example Trace Cache Based Processor



- From Patel's PhD Thesis: "Trace Cache Design for Wide Issue Superscalar Processors," University of Michigan, 1999.

Multiple Branch Predictor

- S. Patel, "Trace Cache Design for Wide Issue Superscalar Processors," PhD Thesis, University of Michigan, 1999.

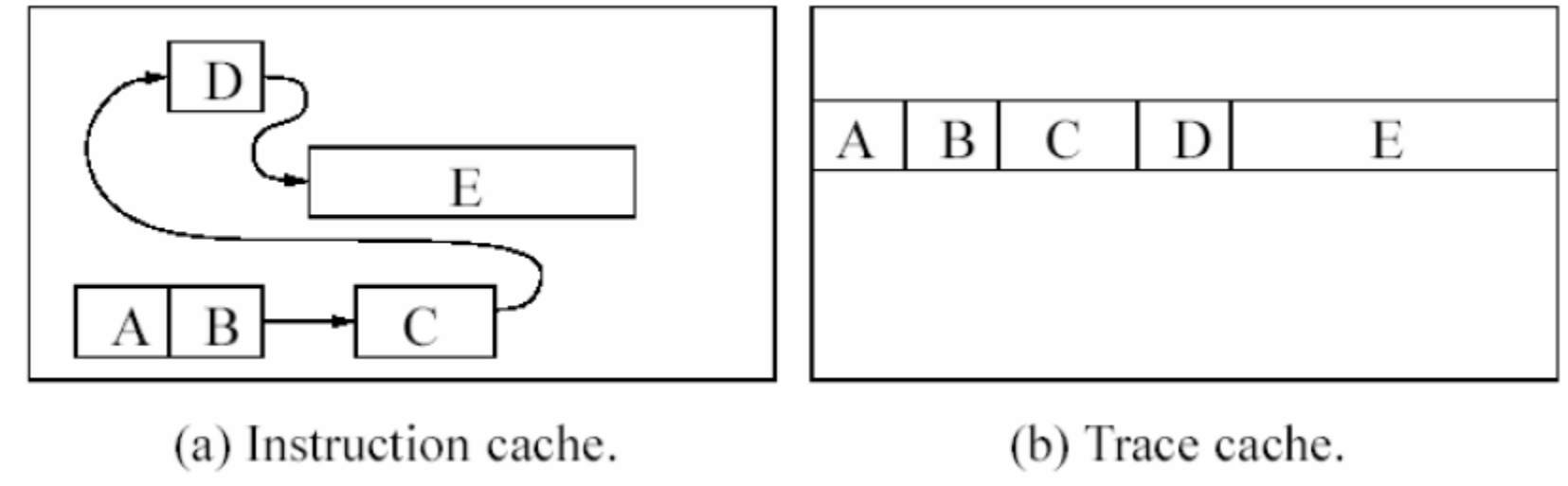


What Does A Trace Cache Line Store?

- 16 slots for instructions. Instructions are stored in decoded form and occupy approximately five bytes for a typical ISA. Up to three branches can be stored per line. Each instruction is marked with a two-bit tag indicating to which block it belongs.
- Four target addresses. With three basic blocks per segment and the ability to fetch partial segments, there are four possible targets to a segment. The four addresses are explicitly stored allowing immediate generation of the next fetch address, even for cases where only a partial segment matches.
- Path information. This field encodes the number and directions of branches in the segment and includes bits to identify whether a segment ends in a branch and whether that branch is a return from subroutine instruction. In the case of a return instruction, the return address stack provides the next fetch address.

- Patel et al., "Critical Issues Regarding the Trace Cache Fetch Mechanism," Umich TR, 1997.

Trace Cache: Advantages/Disadvantages

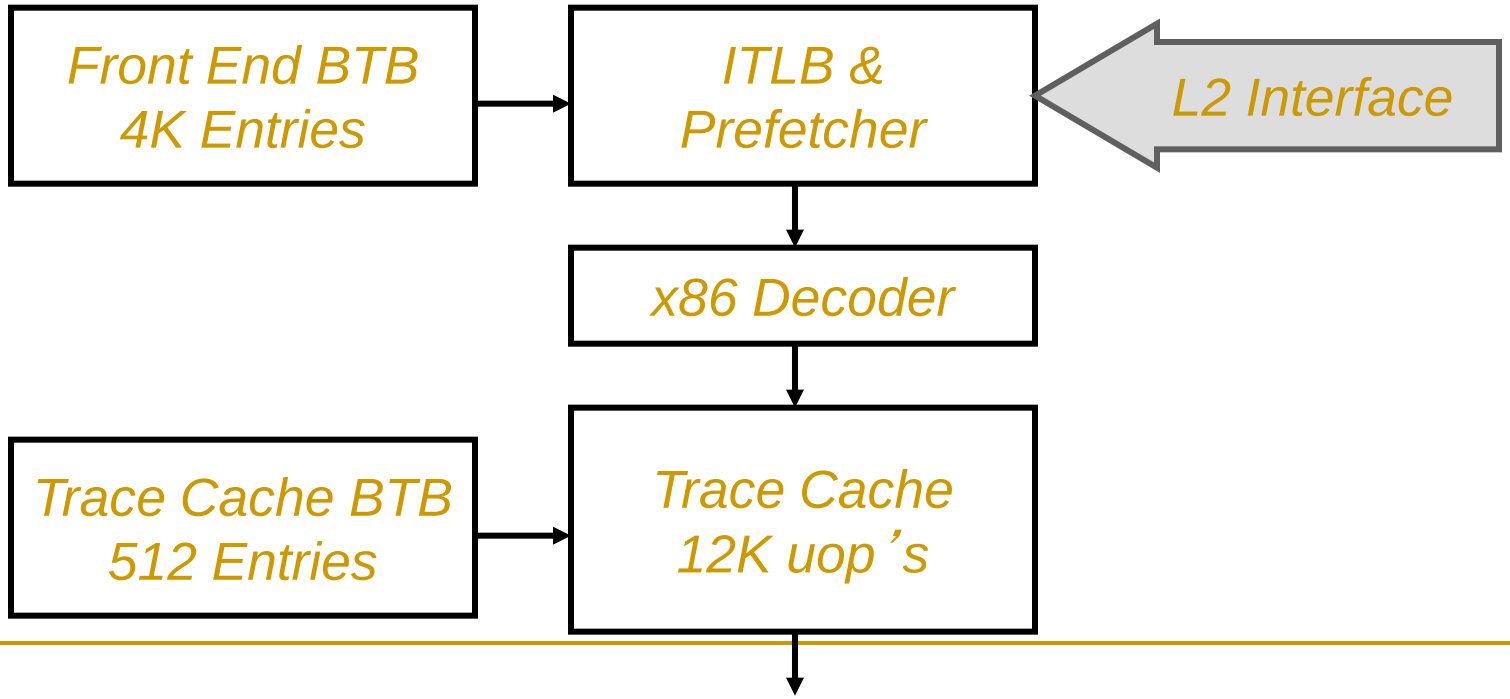


(a) Instruction cache. (b) Trace cache.

+ Reduces fetch breaks (assuming branches are biased)
+ No need for decoding (instructions can be stored in decoded form)
+ Can enable dynamic optimizations within a trace
-- Requires hardware to form traces (more complexity) → called fill unit
-- Results in duplication of the same basic blocks in the cache
-- Can require the prediction of multiple branches per cycle
    -- If multiple cached traces have the same start address
    -- What if XYZ and XYT are both likely traces?

Intel Pentium 4 Trace Cache

- A 12K-uop trace cache replaces the L1 I-cache
- Trace cache stores decoded and cracked instructions
  - Micro-operations (uops): returns 6 uops every other cycle
- x86 decoder can be simpler and slower
- A. Peleg, U. Weiser; "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line", United States Patent No. 5,381,533, Jan 10, 1995



Required Readings for Next Lecture

- Required Reading Assignment:
  - **Chapter 5 and Chapter 9 of Shen and Lipasti (SnL).**
- Recommended References:
  - Robert Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of Research and Development, 11(1):25-33, January 1967.
  - Bo Zhang, "A Tomasulo's Algorithm Emulator," February 1, 2013.

18-740/640
Computer Architecture
Lecture 5: Advanced Branch Prediction

Prof. Onur Mutlu
Carnegie Mellon University
Fall 2015, 9/16/2015