Computer Architecture

Lecture 10: Branch Prediction

Prof. Onur Mutlu
ETH Zürich

Fall 2017

25 October 2017

Mid-Semester Exam

- November 30
- In class
- Questions similar to homework questions

High-Level Summary of Last Week

- SIMD Processing
  - Array Processors
  - Vector Processors
  - SIMD Extensions

- Graphics Processing Units
  - GPU Architecture
  - GPU Programming

Agenda for Today & Tomorrow

- Control Dependence Handling
  - Problem
  - Six solutions
- Branch Prediction
- Other Methods of Control Dependence Handling

Required Readings

- McFarling, "Combining Branch Predictors," DEC WRL Technical Report, 1993. ***Required***

- 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)**
  - ***Required***

Recommended Readings

- 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***

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

Control Dependence Handling

Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction

- 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?

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

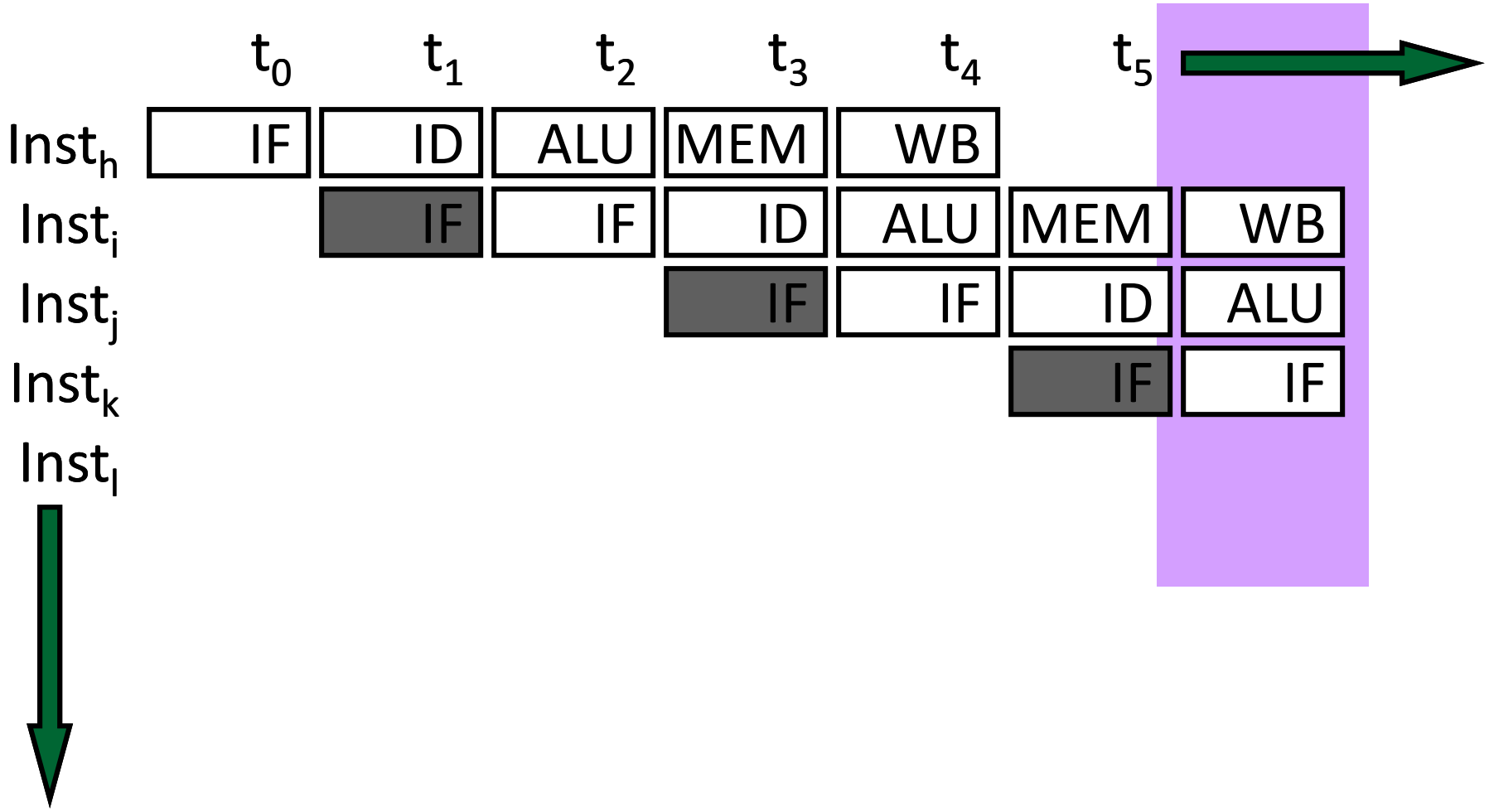
How to Handle Control Dependences

- 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)

Stall Fetch Until Next PC is Known: Good Idea?



This is the case with non-control-flow and unconditional br instructions!

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

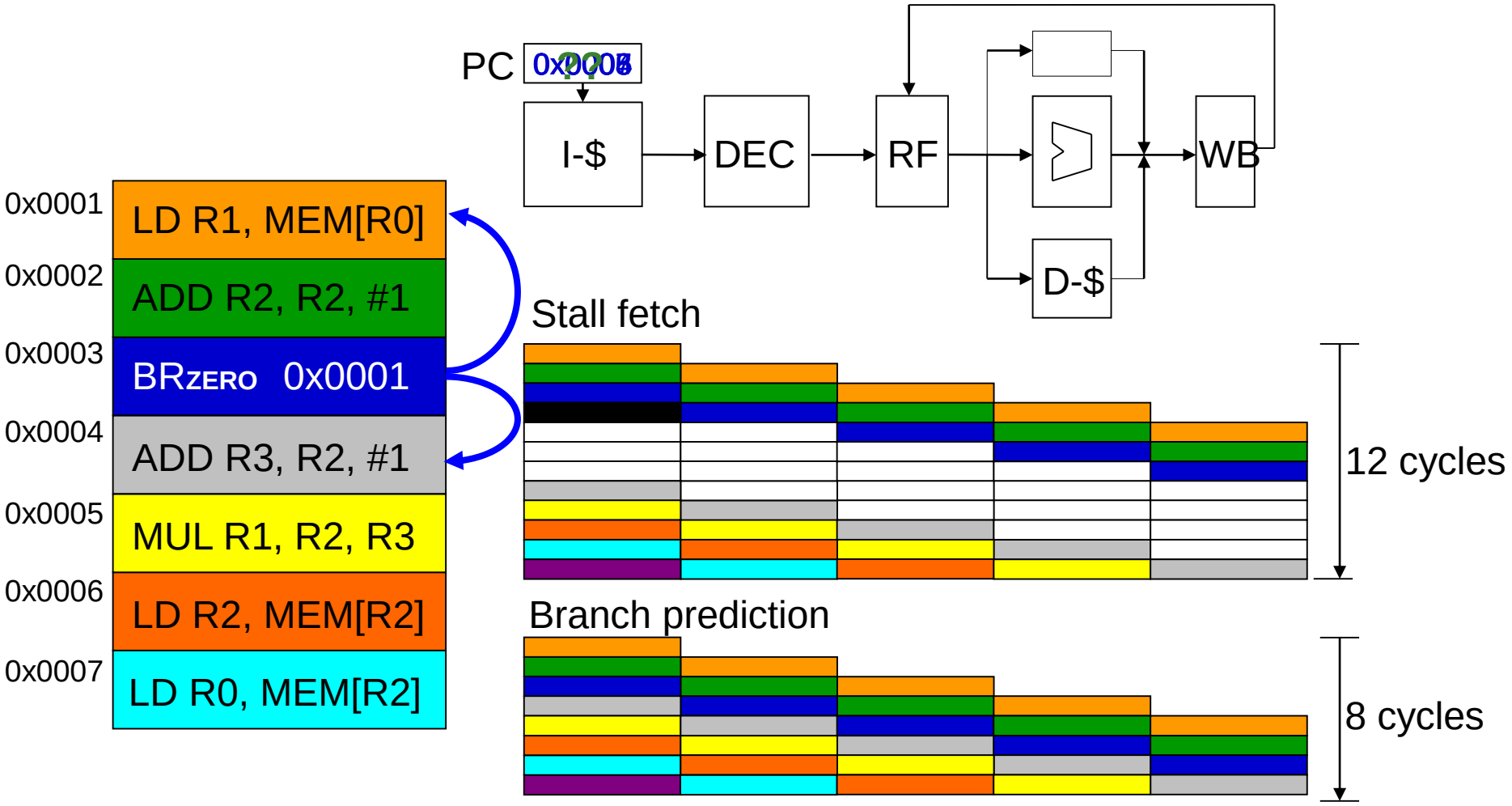
Importance of The Branch Problem

- 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

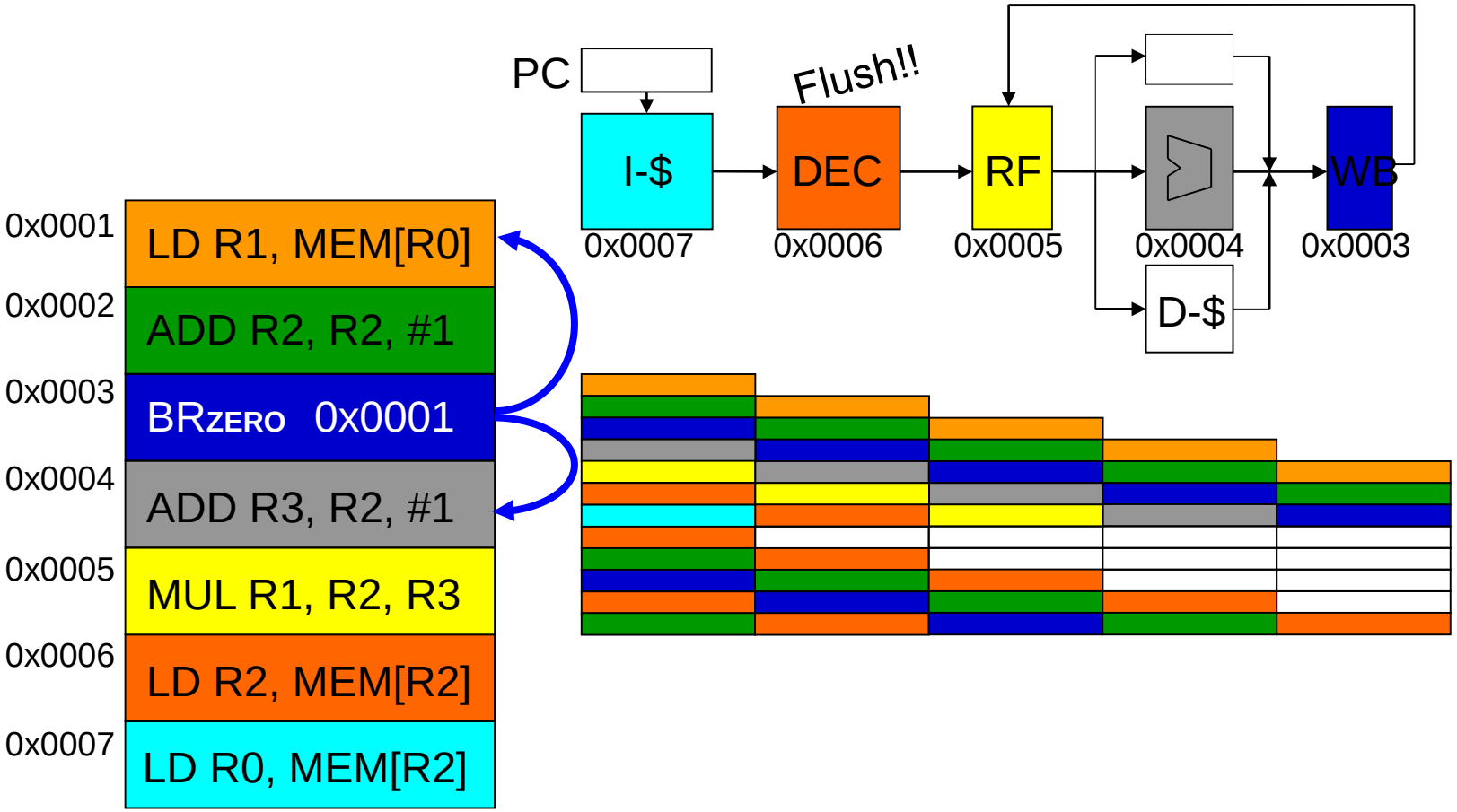
- How long does it take to fetch 500 instructions?
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy
    - 100 (correct path) + 20 (wrong path) = 120 cycles
    - 20% extra instructions fetched
  - 98% accuracy
    - 100 (correct path) + 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

Branch Prediction

Branch Prediction: Guess the Next Instruction to Fetch



Misprediction Penalty



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?

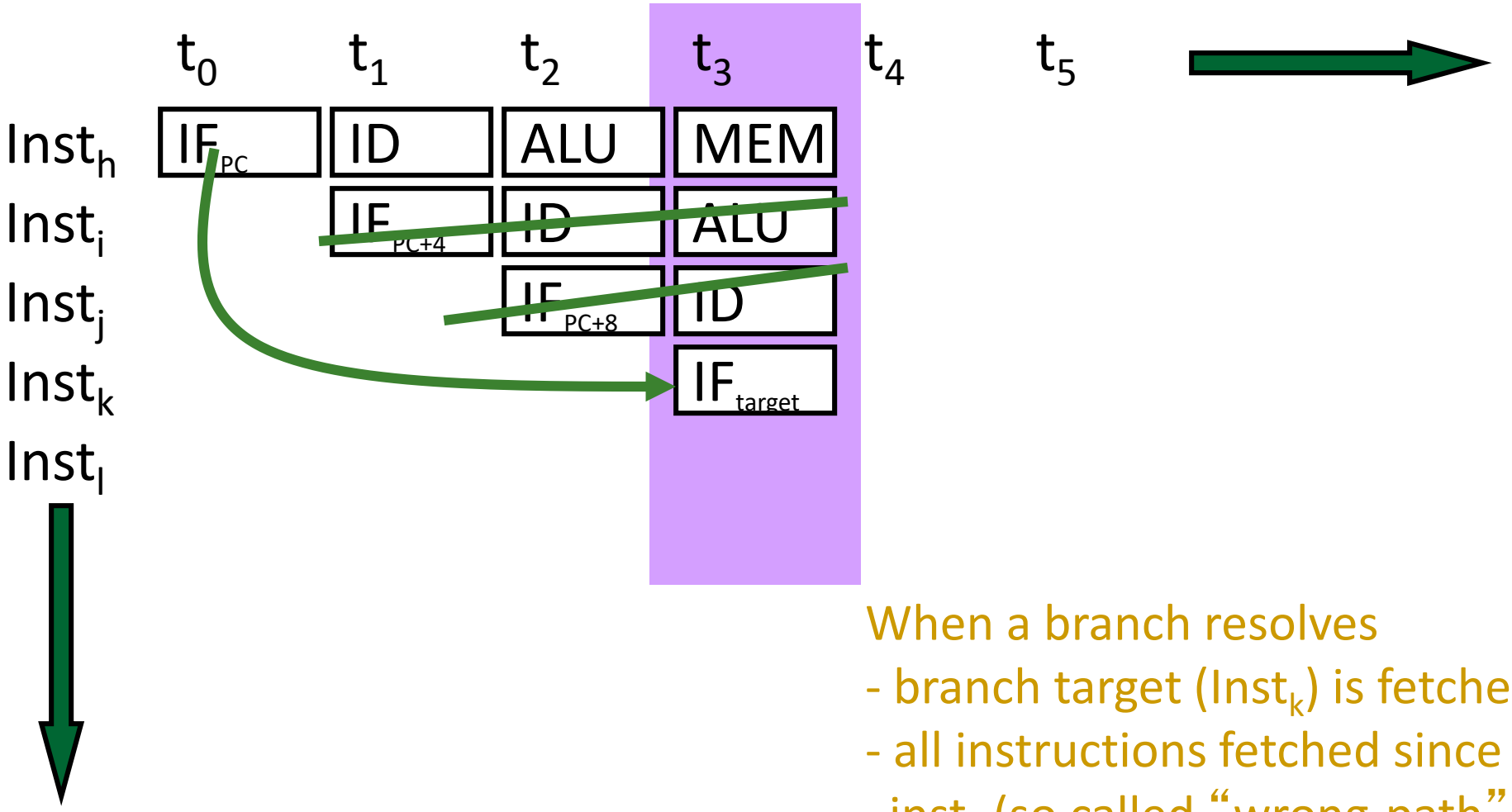
- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed
  - Software: Lay out the control flow graph such that the "likely next instruction" is on the not-taken path of a branch
    - Profile guided code positioning → Pettis & Hansen, PLDI 1990.
  - Hardware: ??? (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

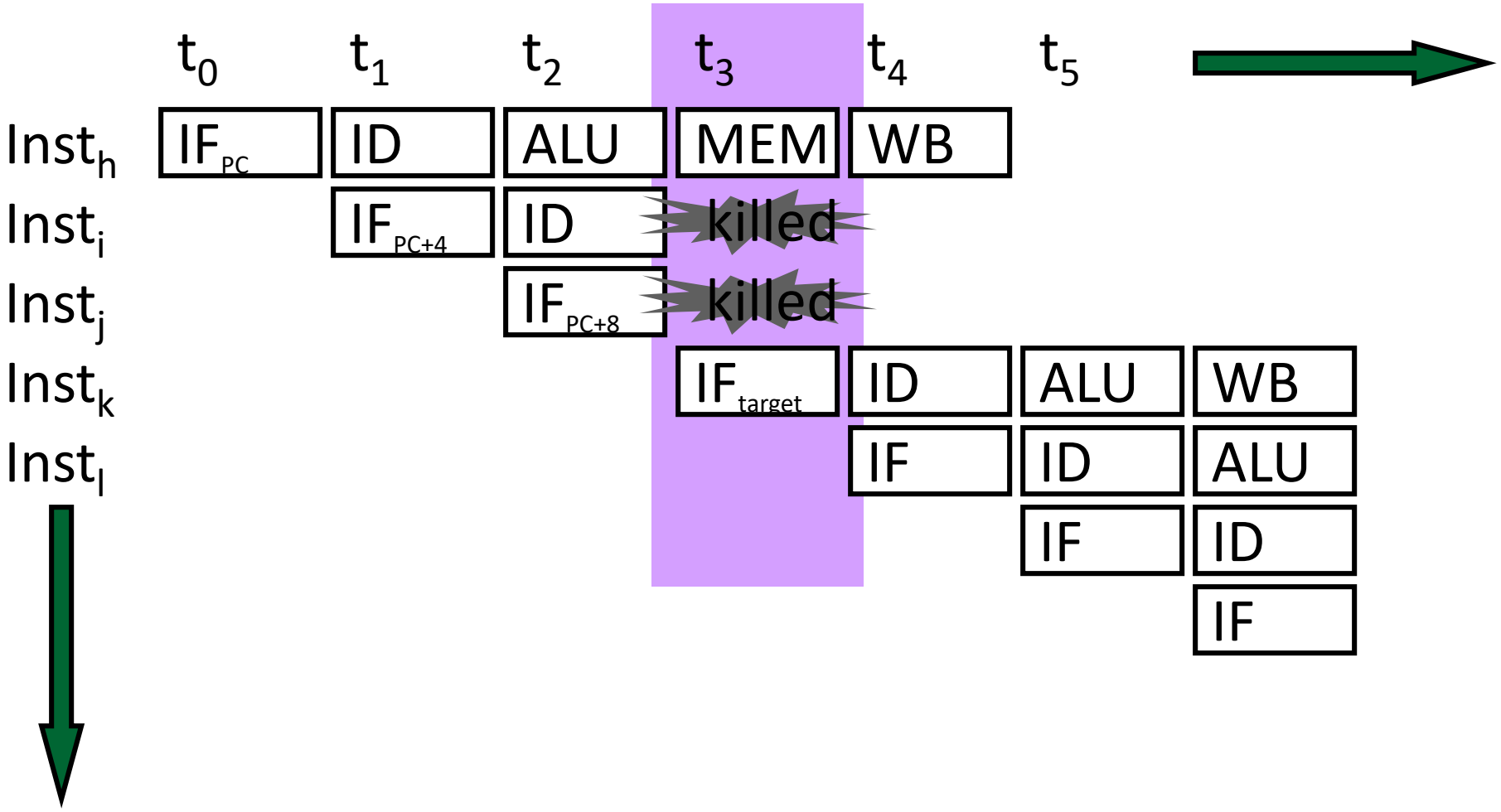
Branch Prediction: Always PC+4



When a branch resolves
- branch target ($Inst_k$) is fetched
- all instructions fetched since $inst_h$ (so called "wrong-path" instructions) must be flushed

$Inst_h$ is a branch

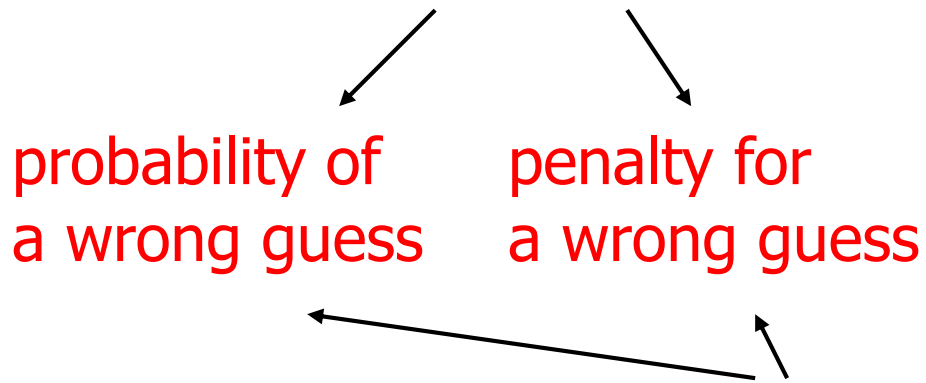
Pipeline Flush on a Misprediction



$Inst_h$ is a branch

Performance Analysis

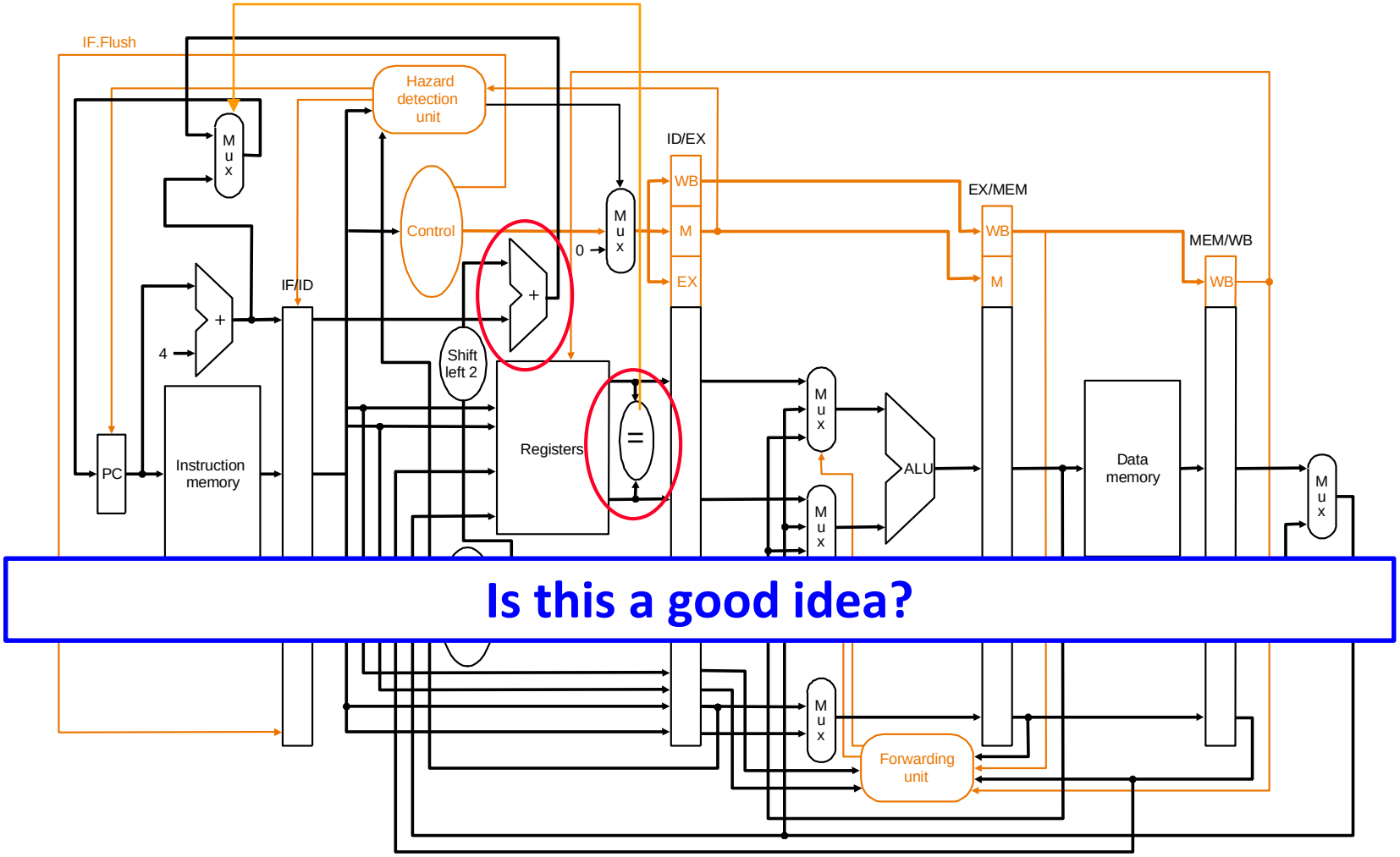
- correct guess ⇒ no penalty ~86% of the time
- incorrect guess ⇒ 2 bubbles
- Assume
  - no data dependency related stalls
  - 20% control flow instructions
  - 70% of control flow instructions are taken
  - CPI = [1 + (0.20*0.7) * 2] =
    = [1 + 0.14 * 2] = 1.28



Can we reduce either of the two penalty terms?

Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

CPI = [1 + (0.2*0.7) * 1] = 1.14

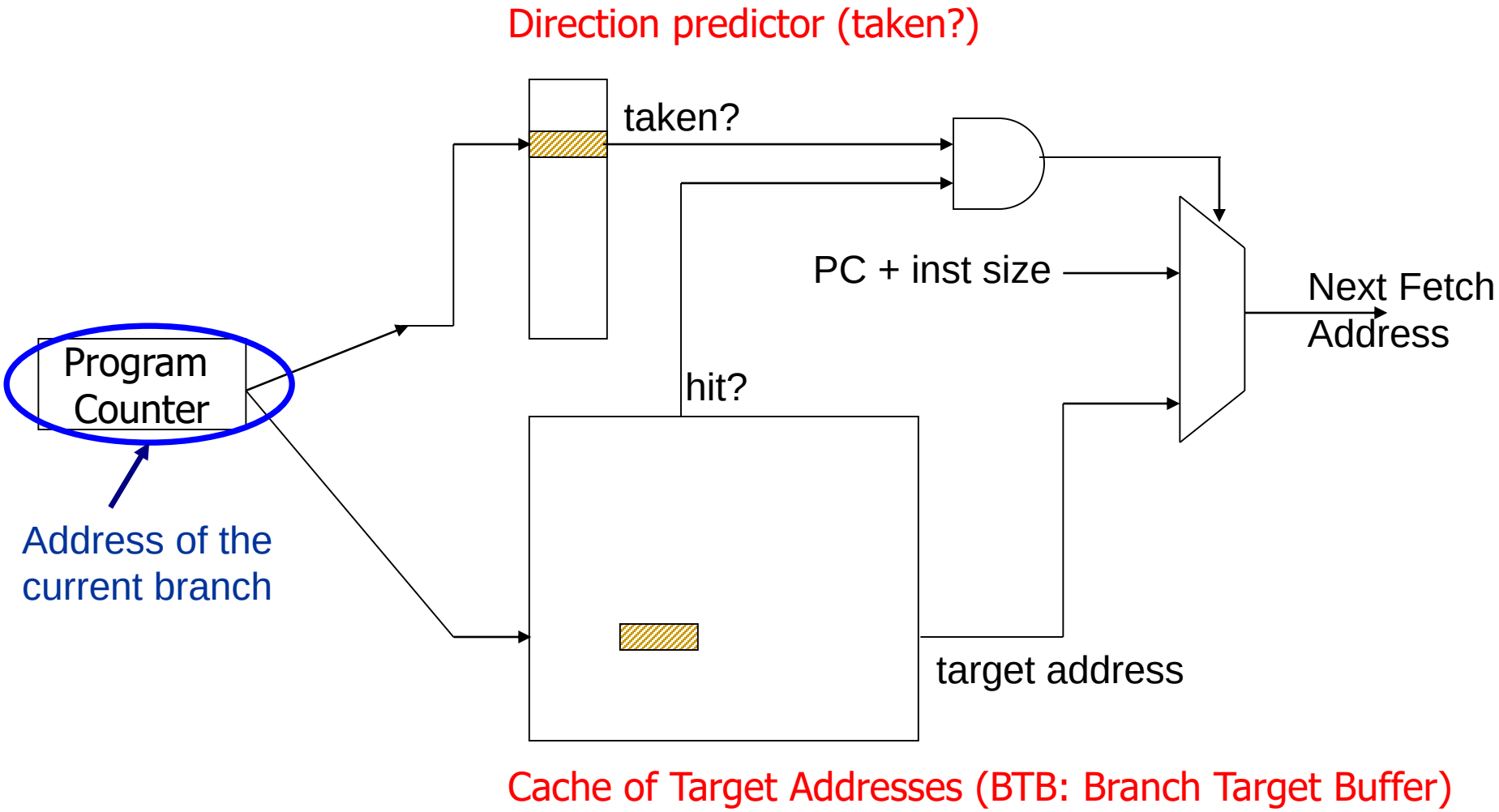
Branch Prediction (A Bit More Enhanced)

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

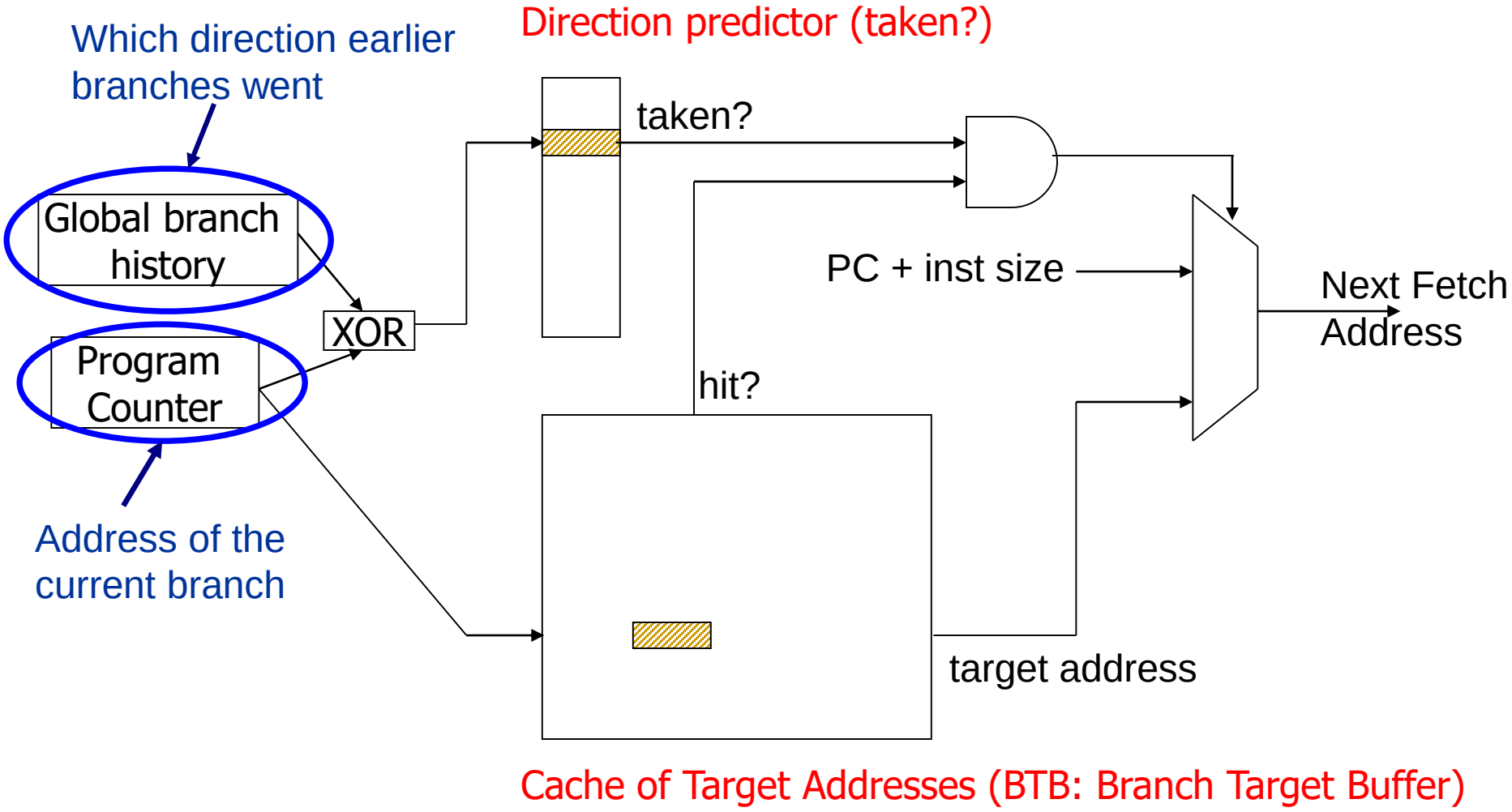
- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)

- 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)

Static Branch Prediction (I)

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

- Always taken
  - No direction prediction
  - Better accuracy: ~60-70% (for conditional branches)
    - 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 a 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
  - Example opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
  - Example 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 (ditto for other static methods)

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

Static Branch Prediction (IV)

- 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?

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 is the common disadvantage 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…)
    - What is a Dynamic Compiler?
      - A compiler that generates code at runtime: Remember Transmeta?
      - Java JIT (just in time) compiler, Microsoft CLR (common lang. runtime)

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)

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

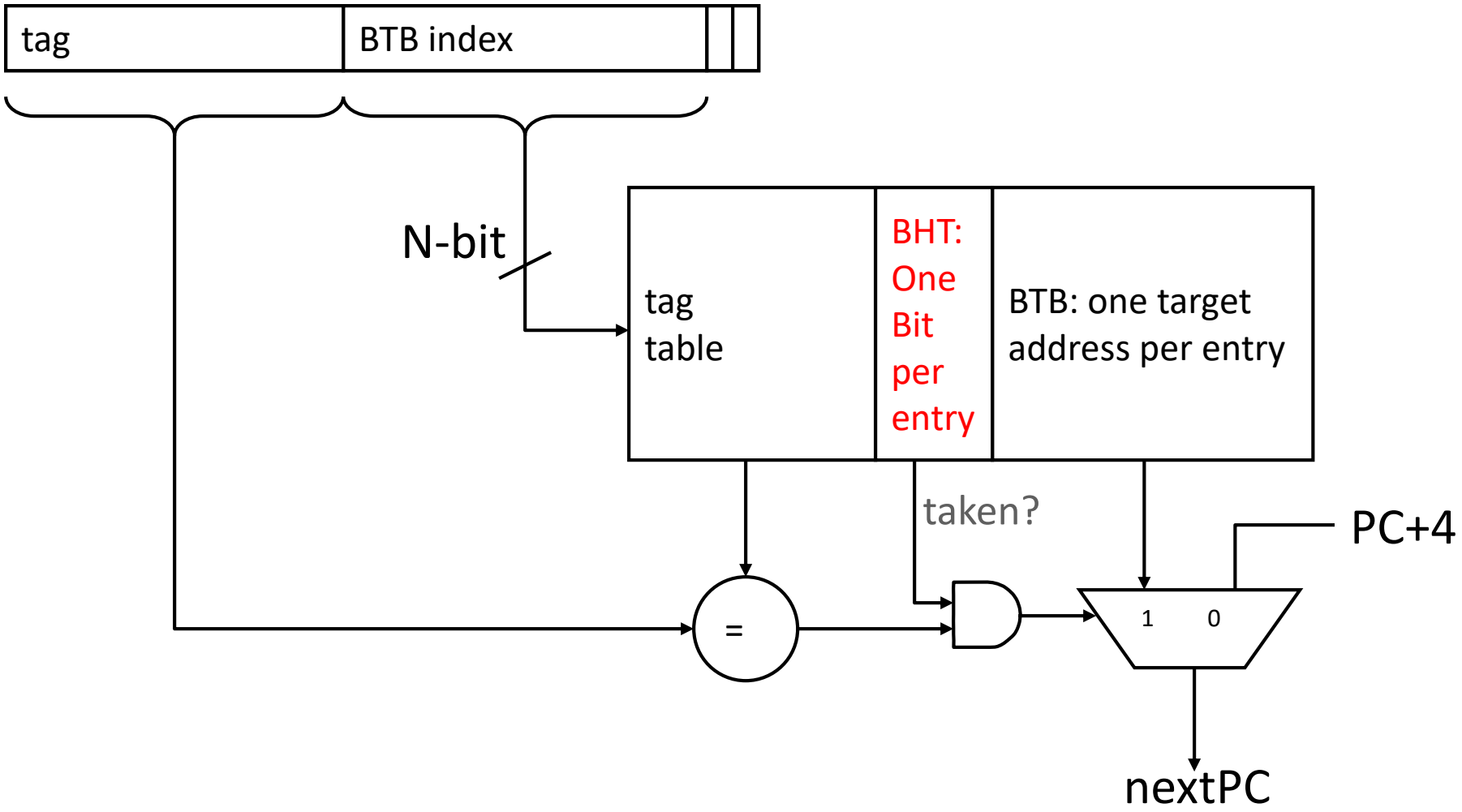
- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
    TTTTTTTTTTNNNNNNNNNN → 90% accuracy

- 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 N (number of iterations)

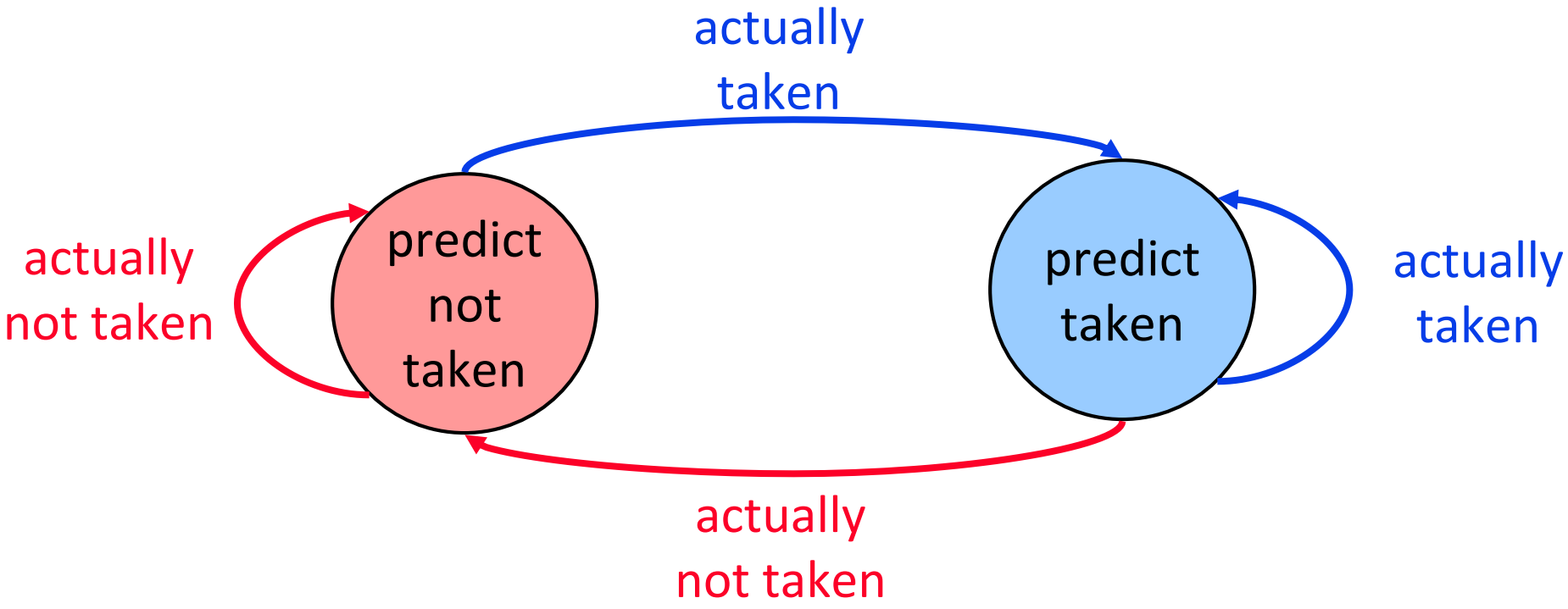
-- Loop branches for loops will small N (number of iterations)
TNTNTNTNTNTNTNTNTNTN → 0% accuracy

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

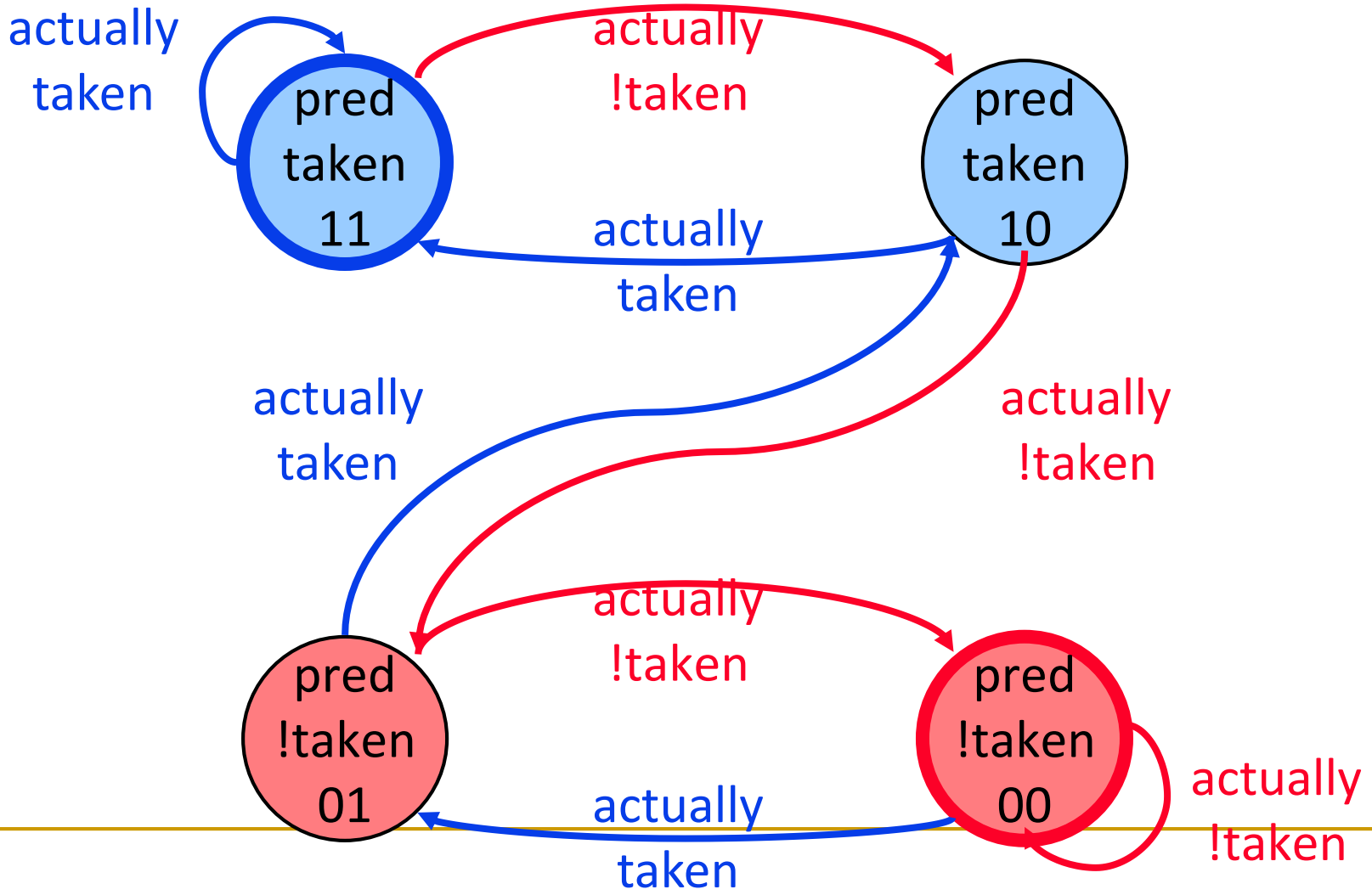
- Accuracy for a loop with N iterations = (N-1)/N
  TNTNTNTNTNTNTNTNTNTNTN → 50% accuracy

(assuming counter initialized to weakly taken)

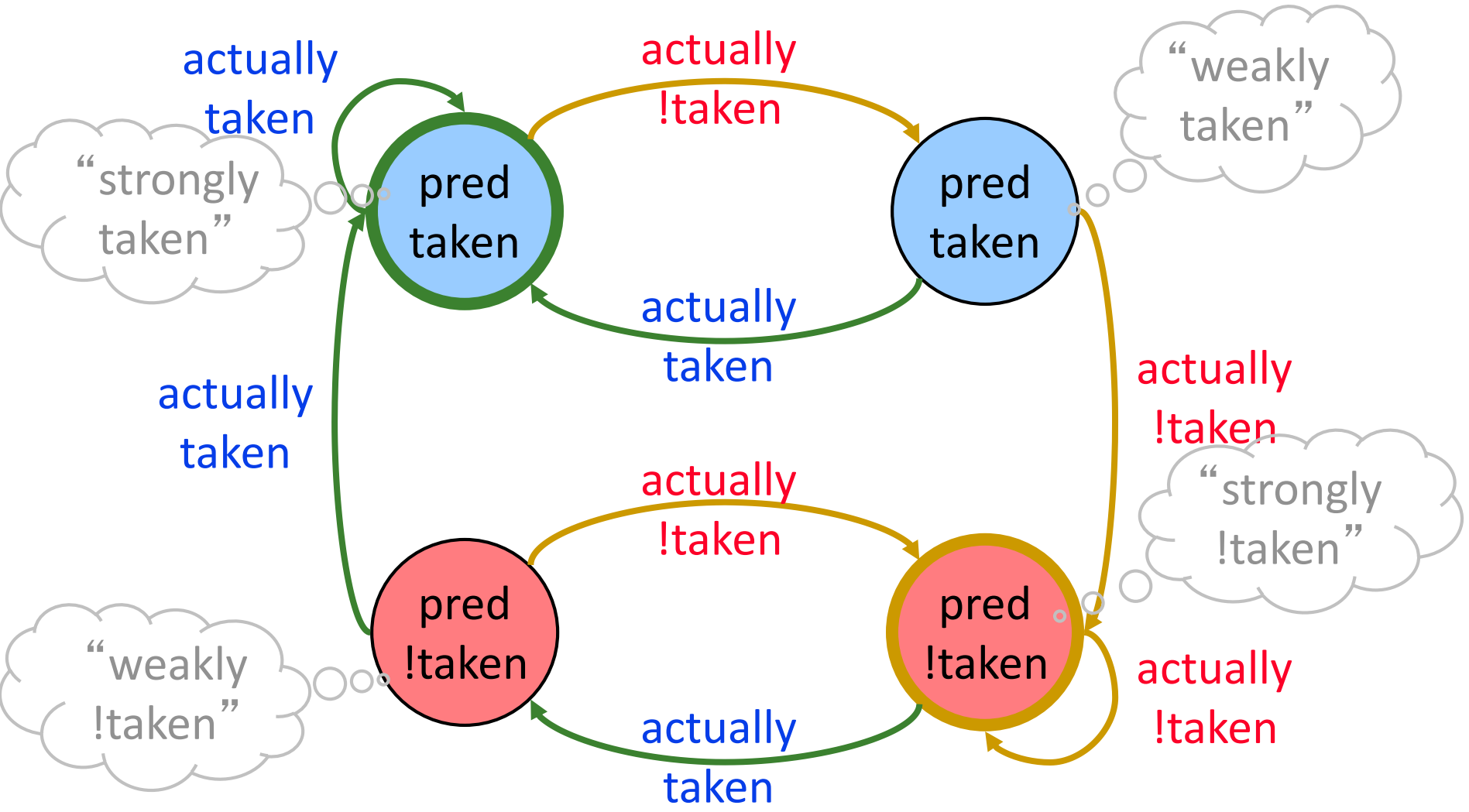
+ Better prediction accuracy
-- More hardware cost (but counter can be part of a BTB entry)

State Machine for 2-bit Saturating Counter

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



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

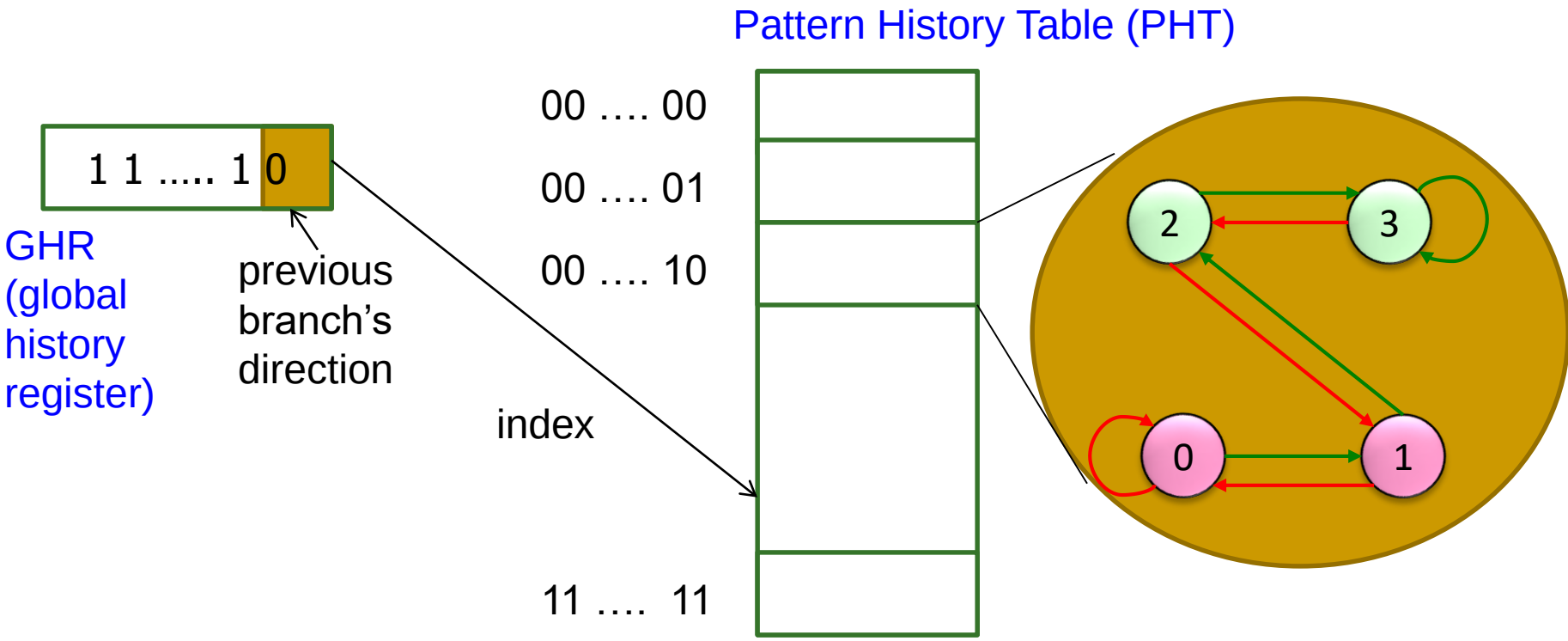
Capturing Global Branch Correlation

- 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

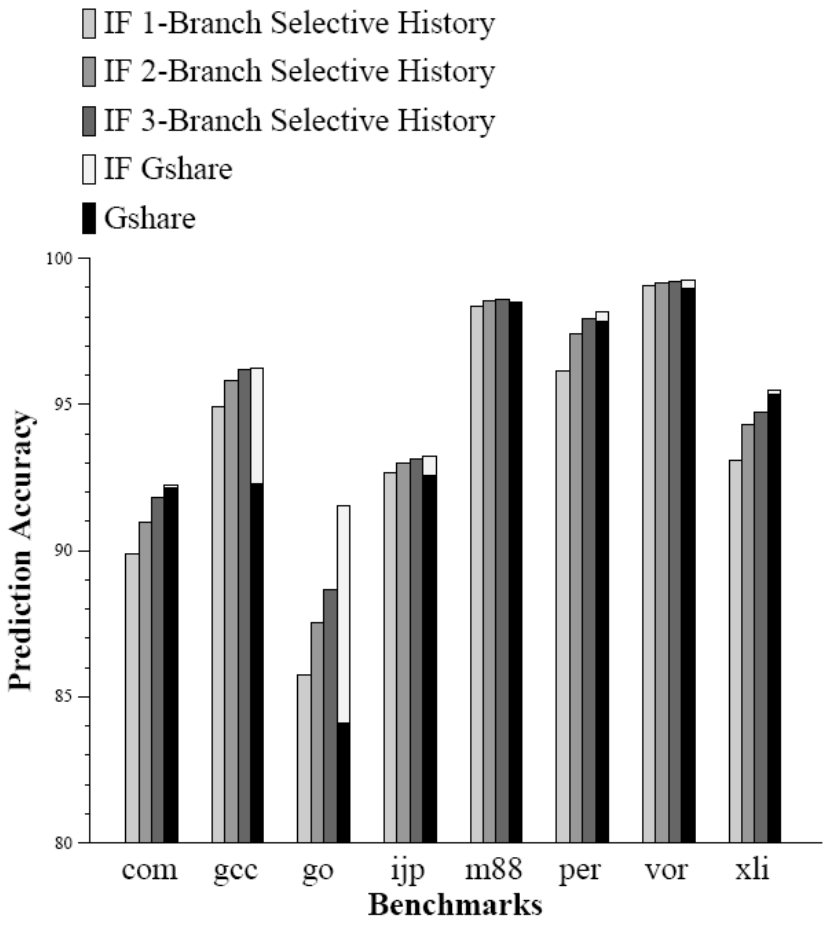
- Two level global branch predictor
- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
  - Which pattern history table to use is determined by lower order bits of the branch address

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.



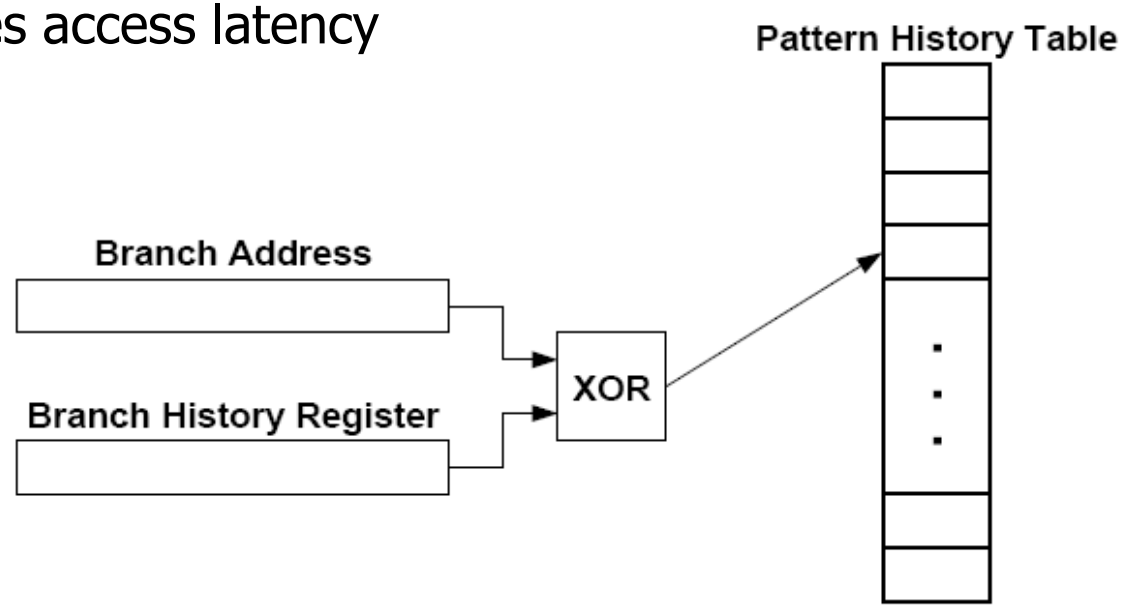
Improving Global Predictor Accuracy

- 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

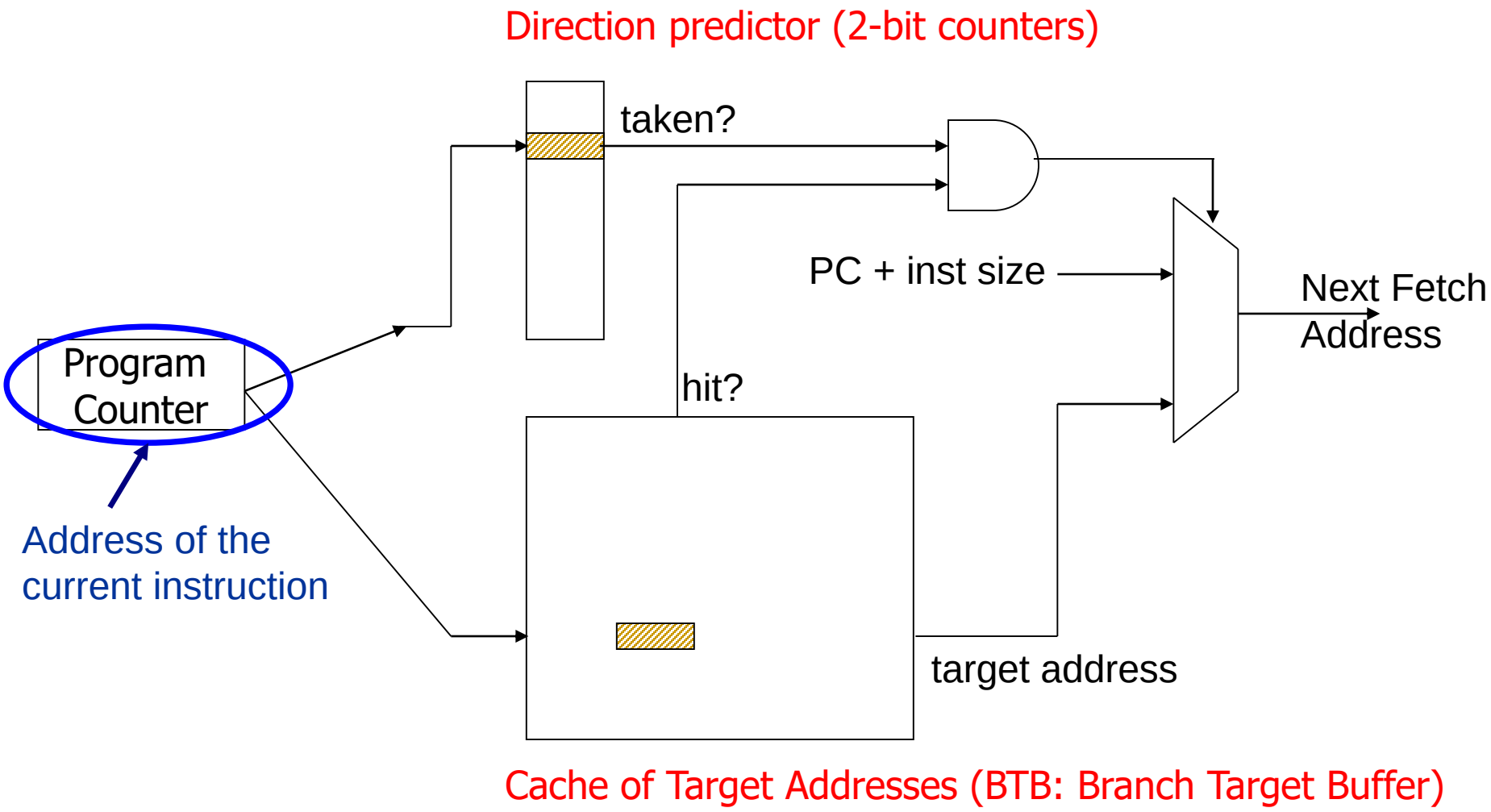
  + Better utilization of PHT

  -- Increases access latency

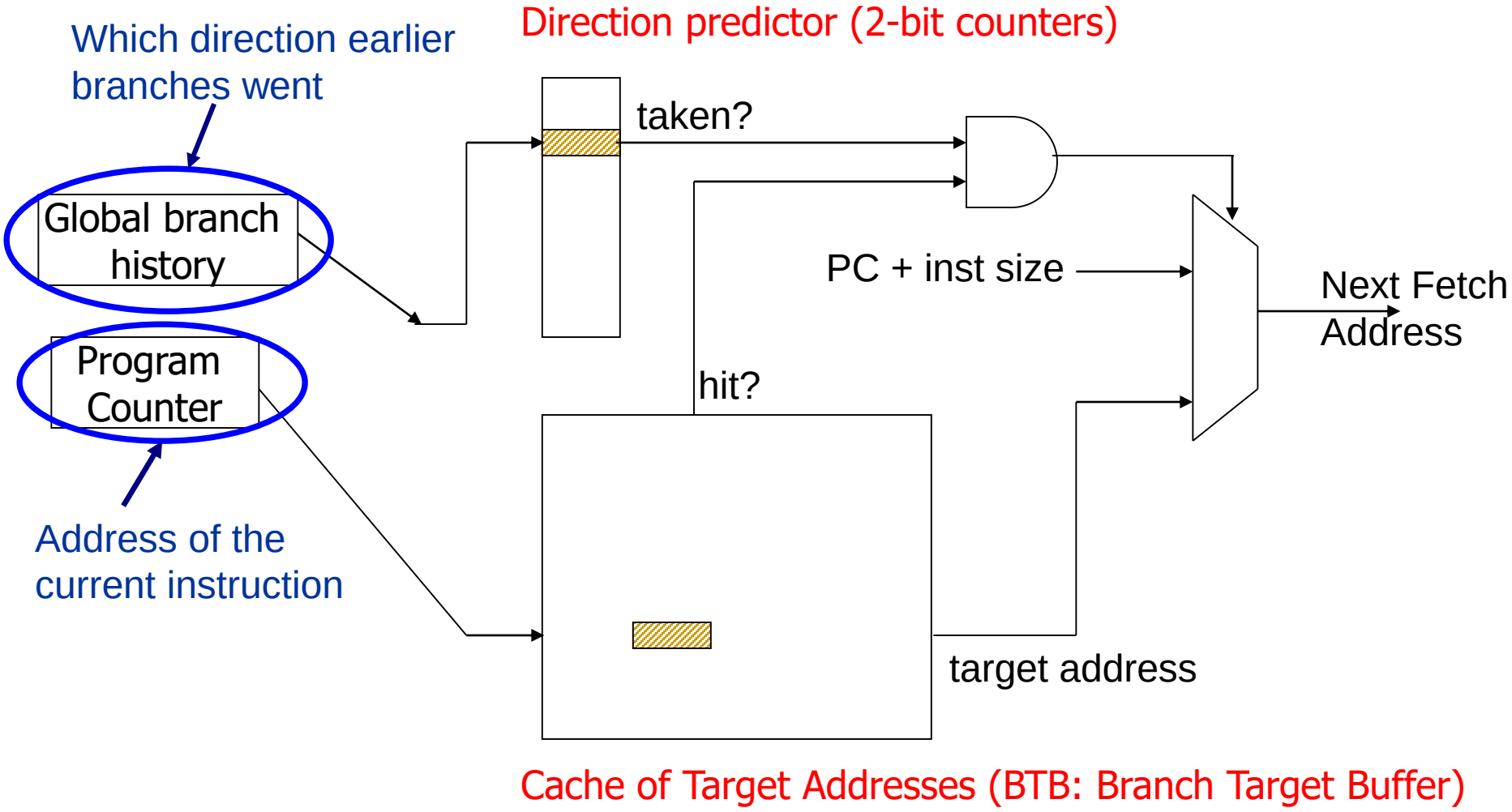


- McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

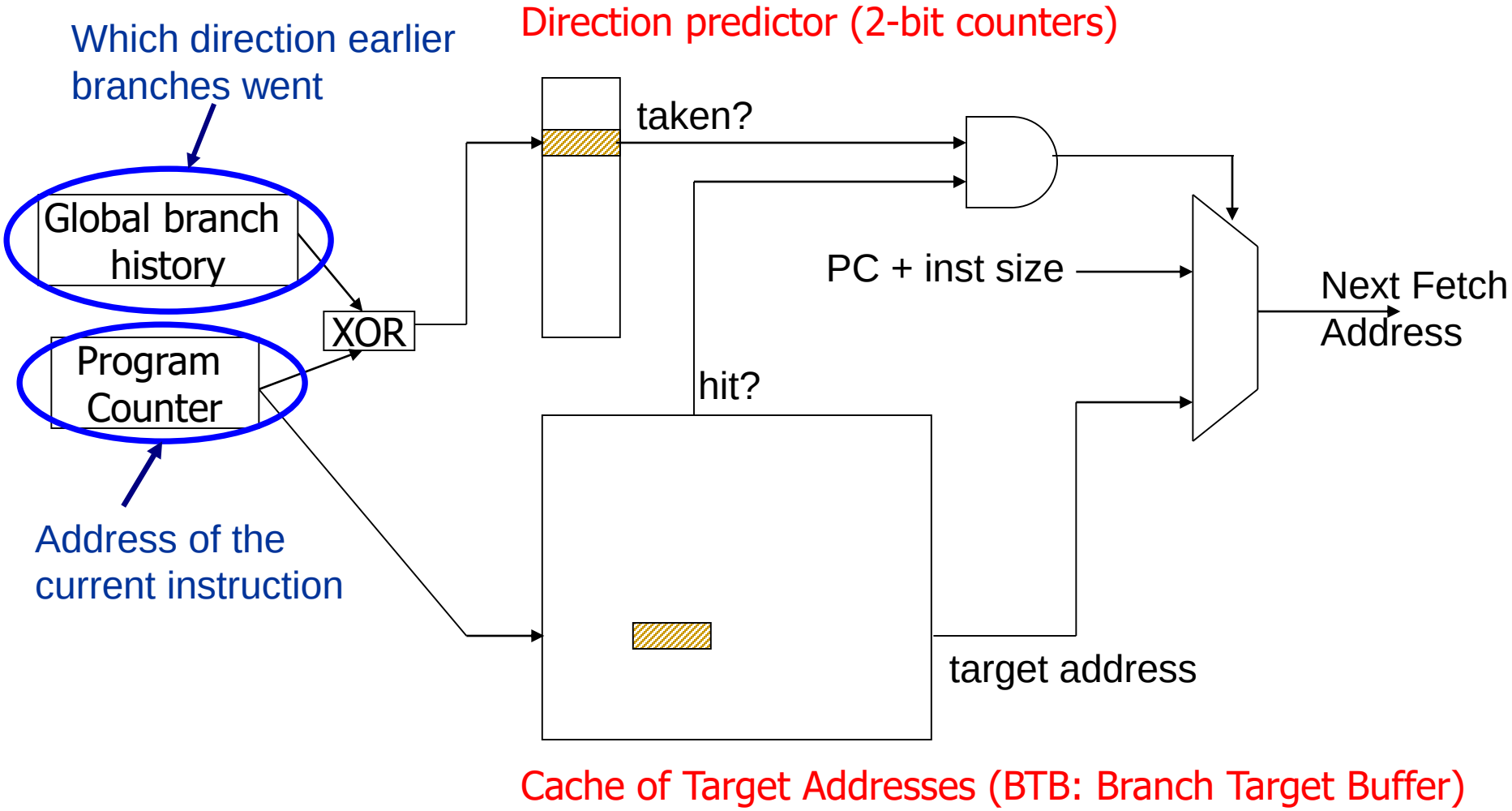
Review: One-Level Branch Predictor



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

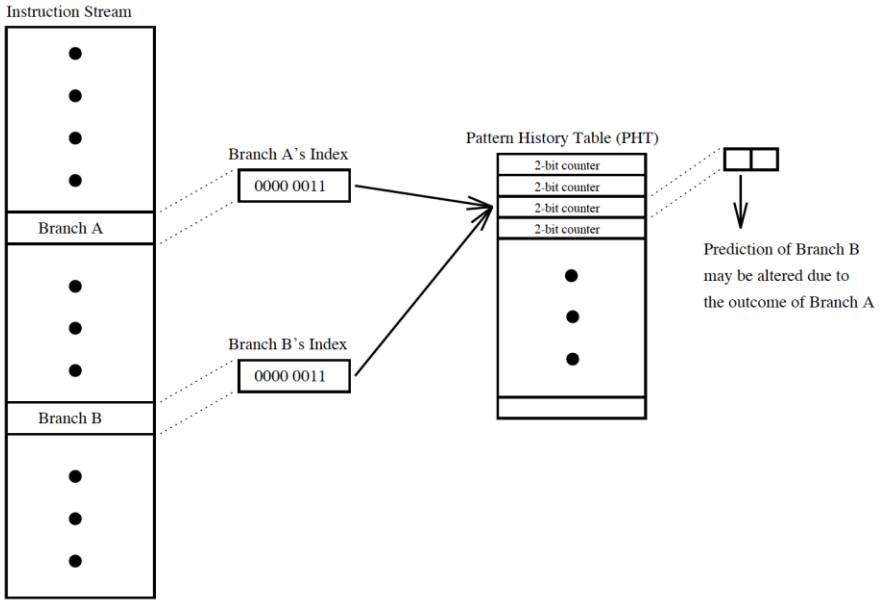


Figure 2: Interference in a two-level predictor.

- 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

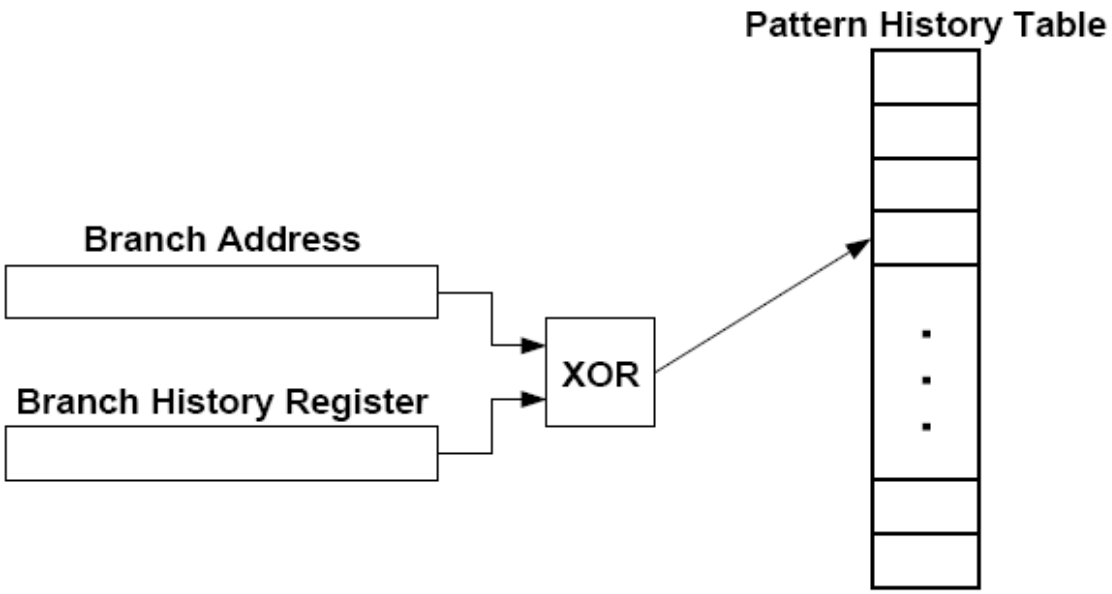
- Observation: Many branches are biased in one direction (e.g., 99% taken)
- Problem: These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing "interference" in branch prediction tables and history registers
- Solution: Detect such biased 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



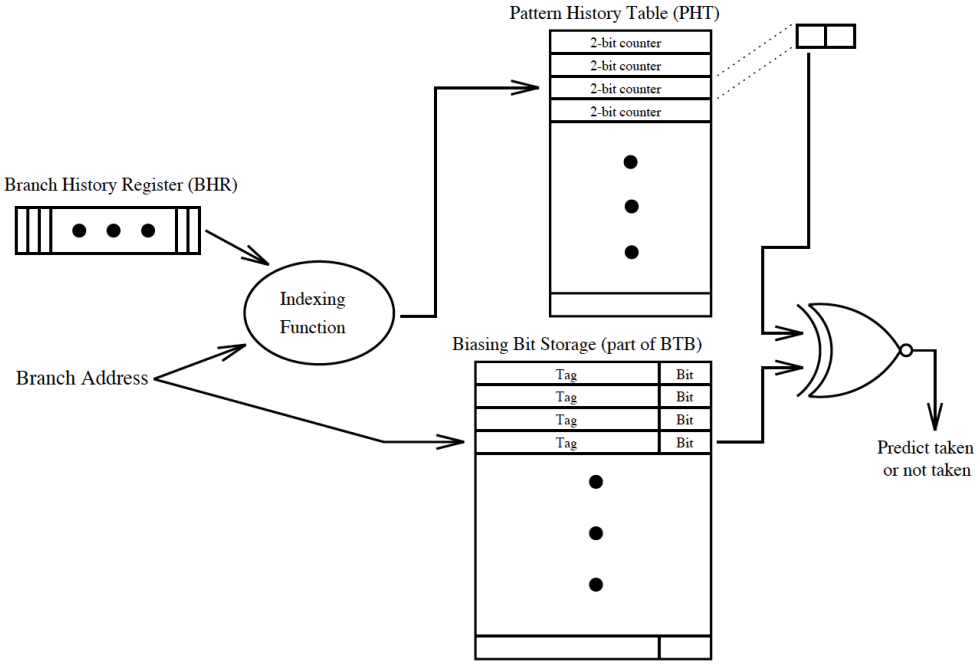
- McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

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%

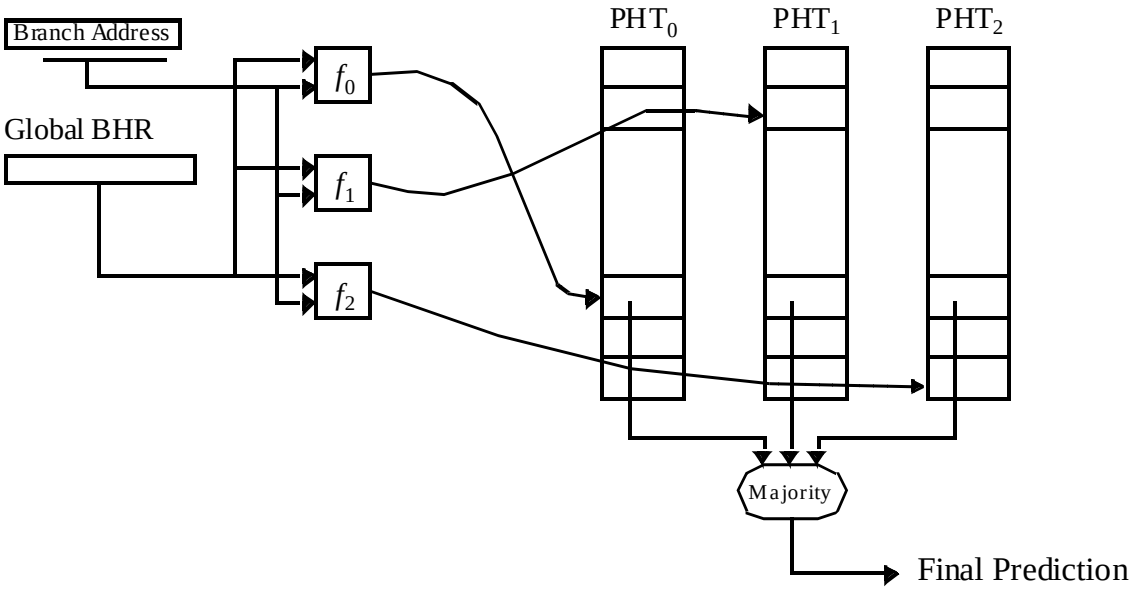
- Works because most branches are biased (not 50% taken)

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 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

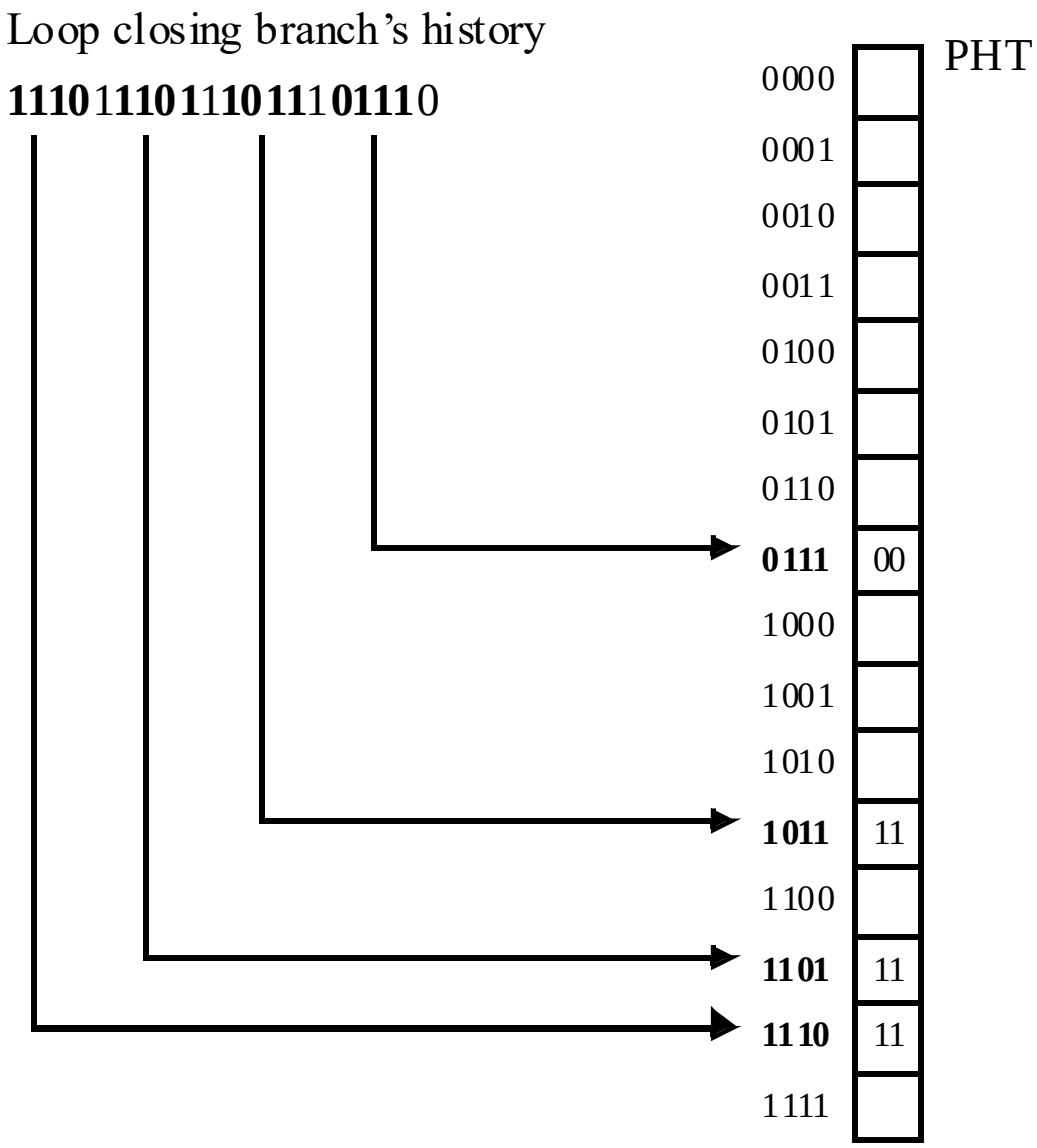
```
for (i=1; i<=4; i++) { }
```

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



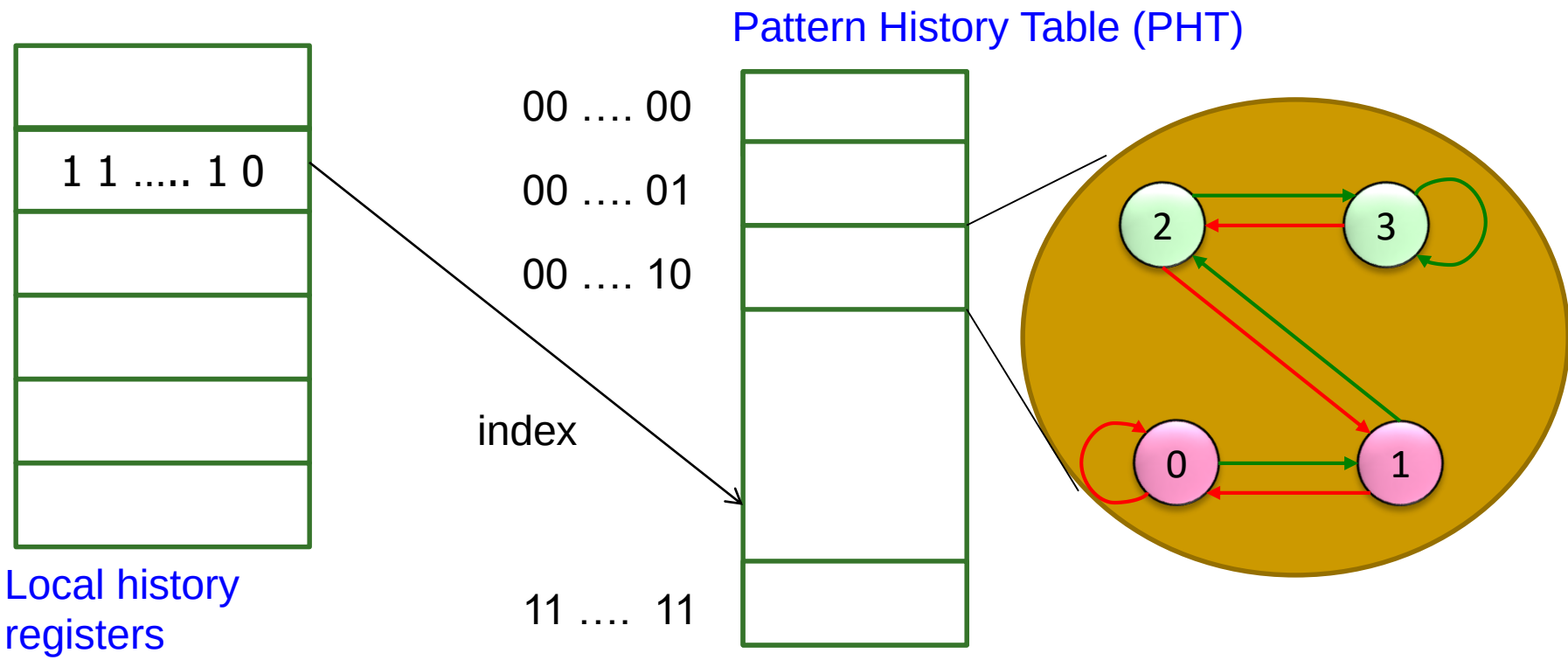
Capturing Local Branch Correlation

- 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)

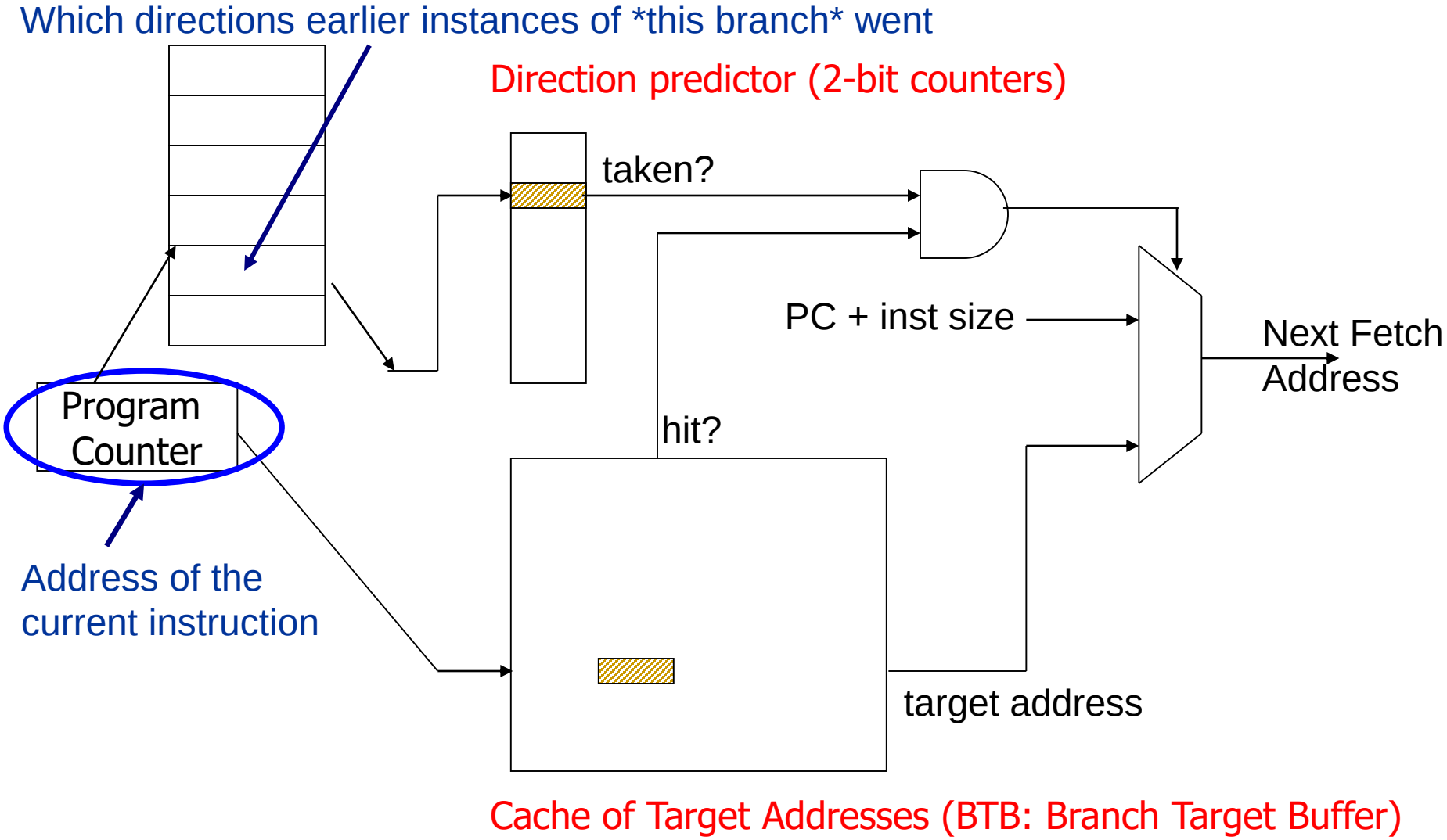
Two Level Local Branch Prediction

- 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



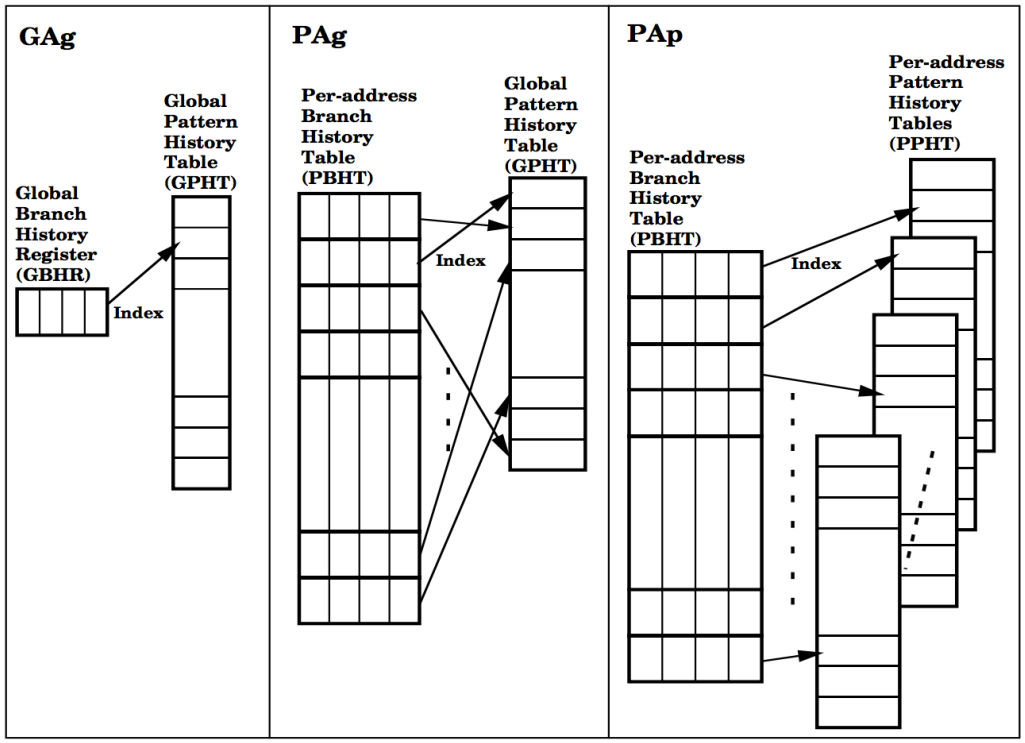
Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

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, "Alternative Implementations of Two-Level Adaptive Branch Prediction," ISCA 1992.

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)

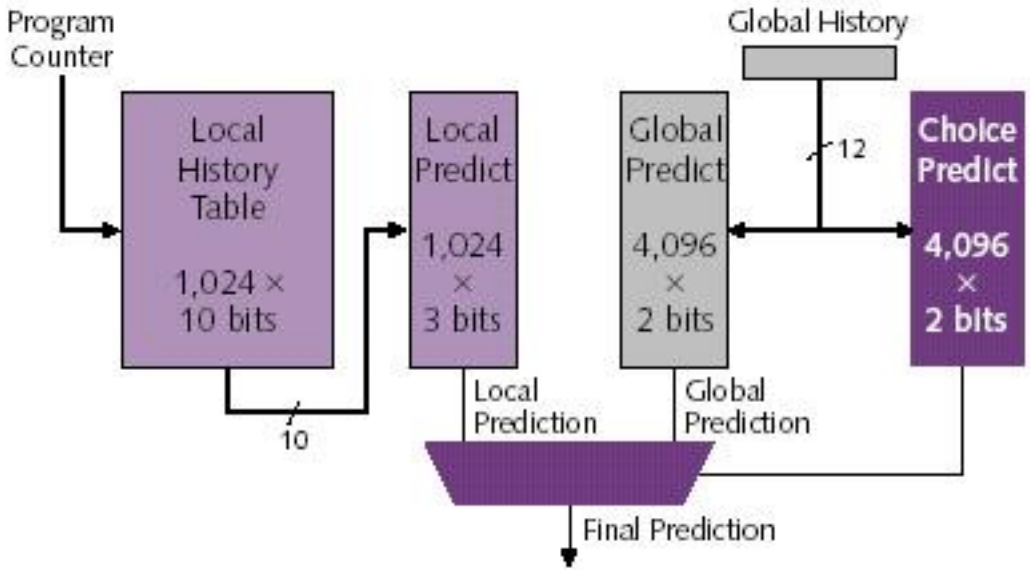
- Disadvantages:

  -- 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

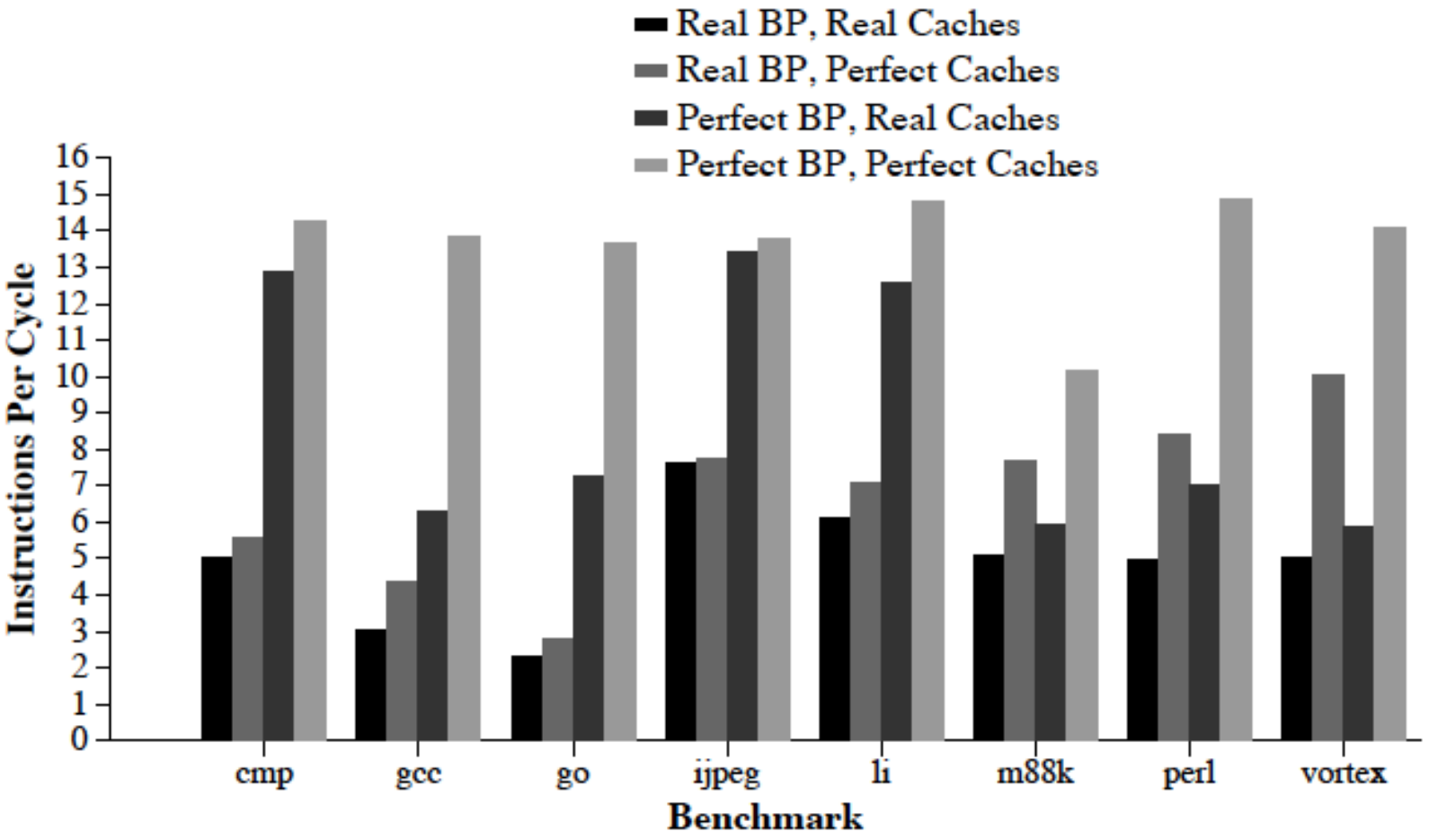
- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

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.

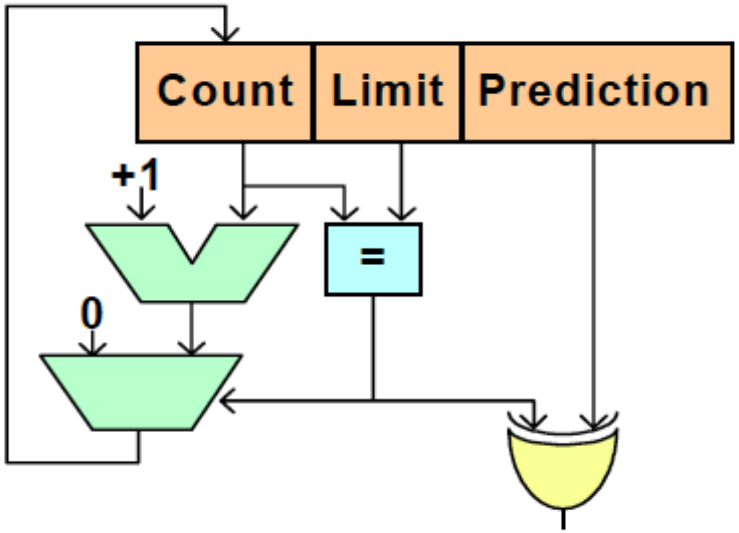


Figure 2: The Loop Detector logic

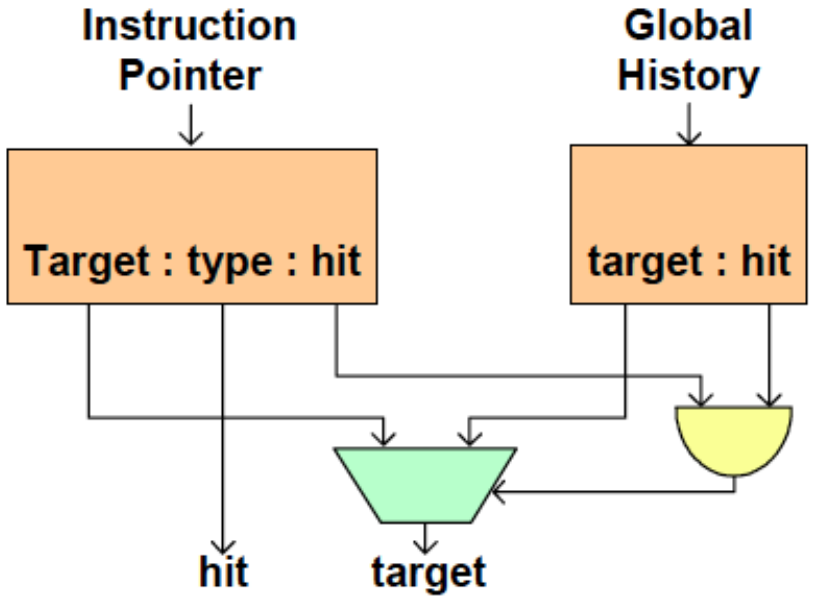
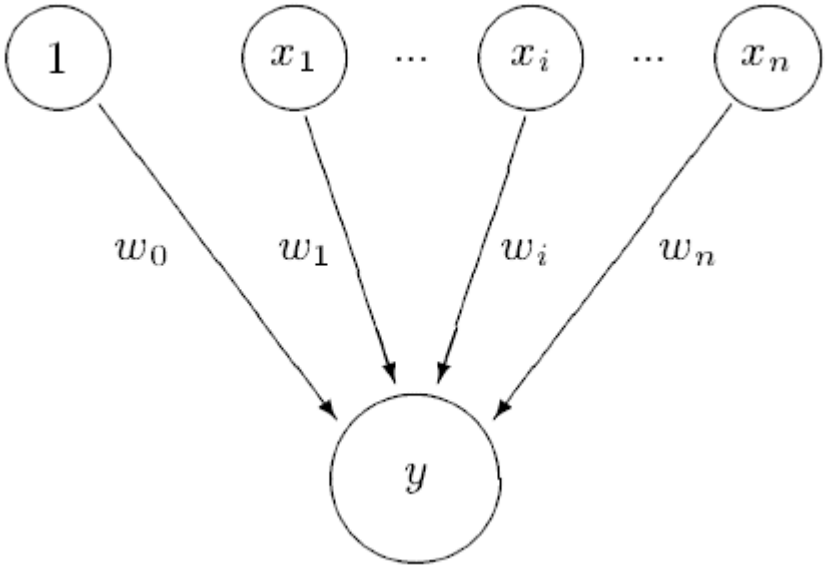


Figure 3: The Indirect Branch Predictor logic

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



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

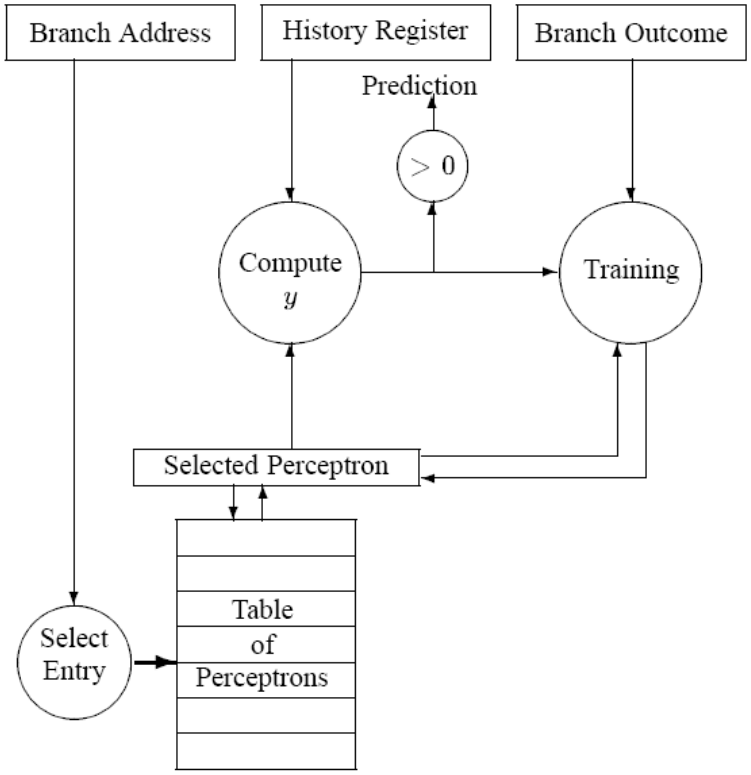
Each branch associated with a perceptron

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

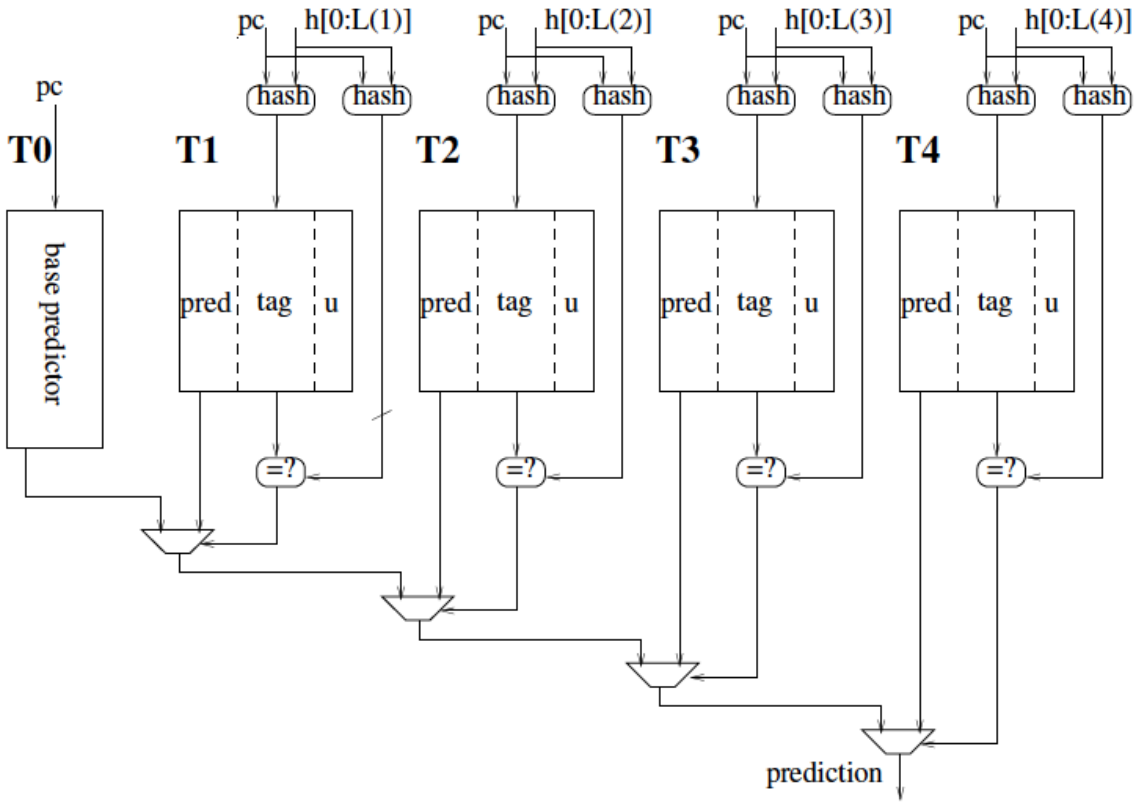
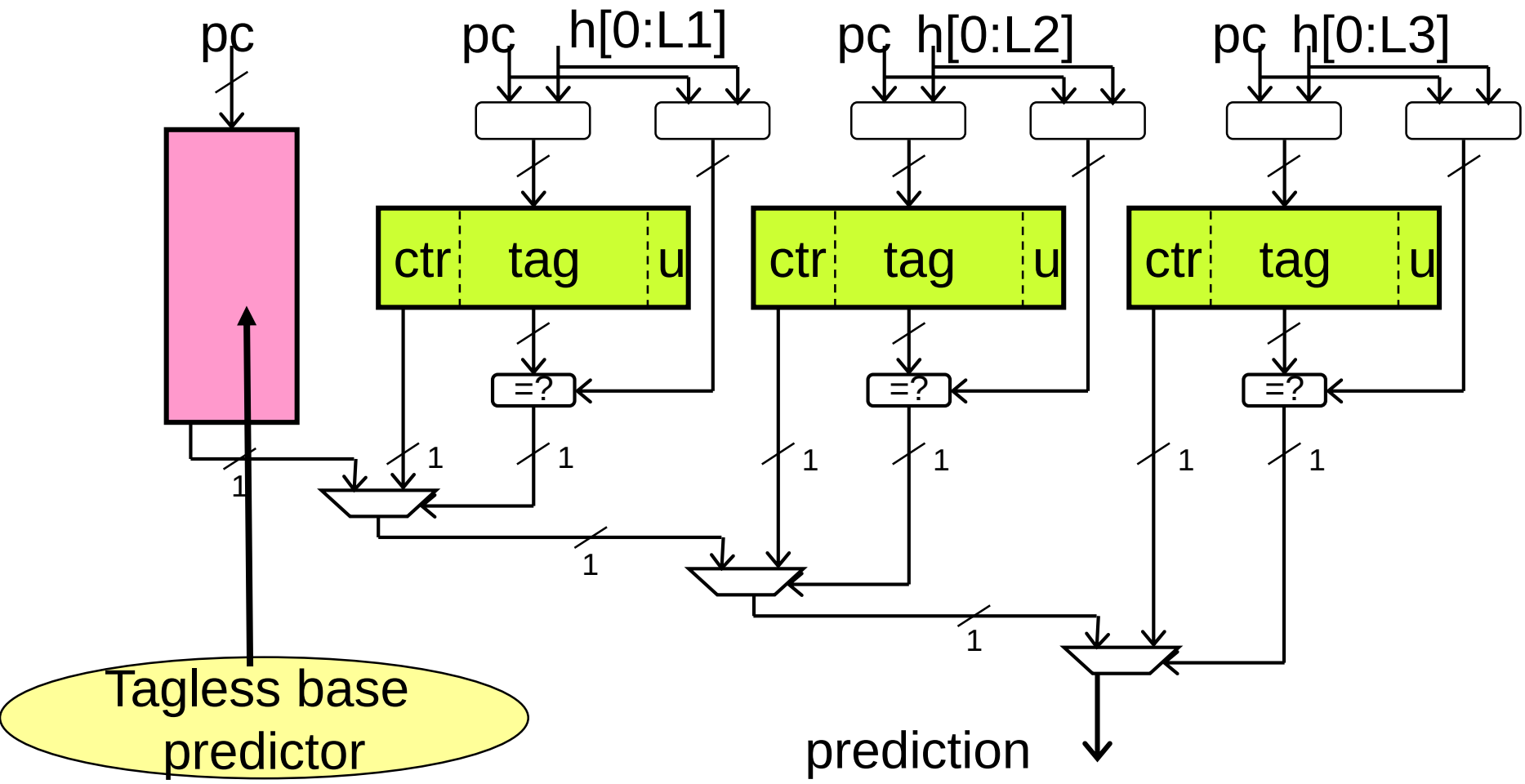


Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

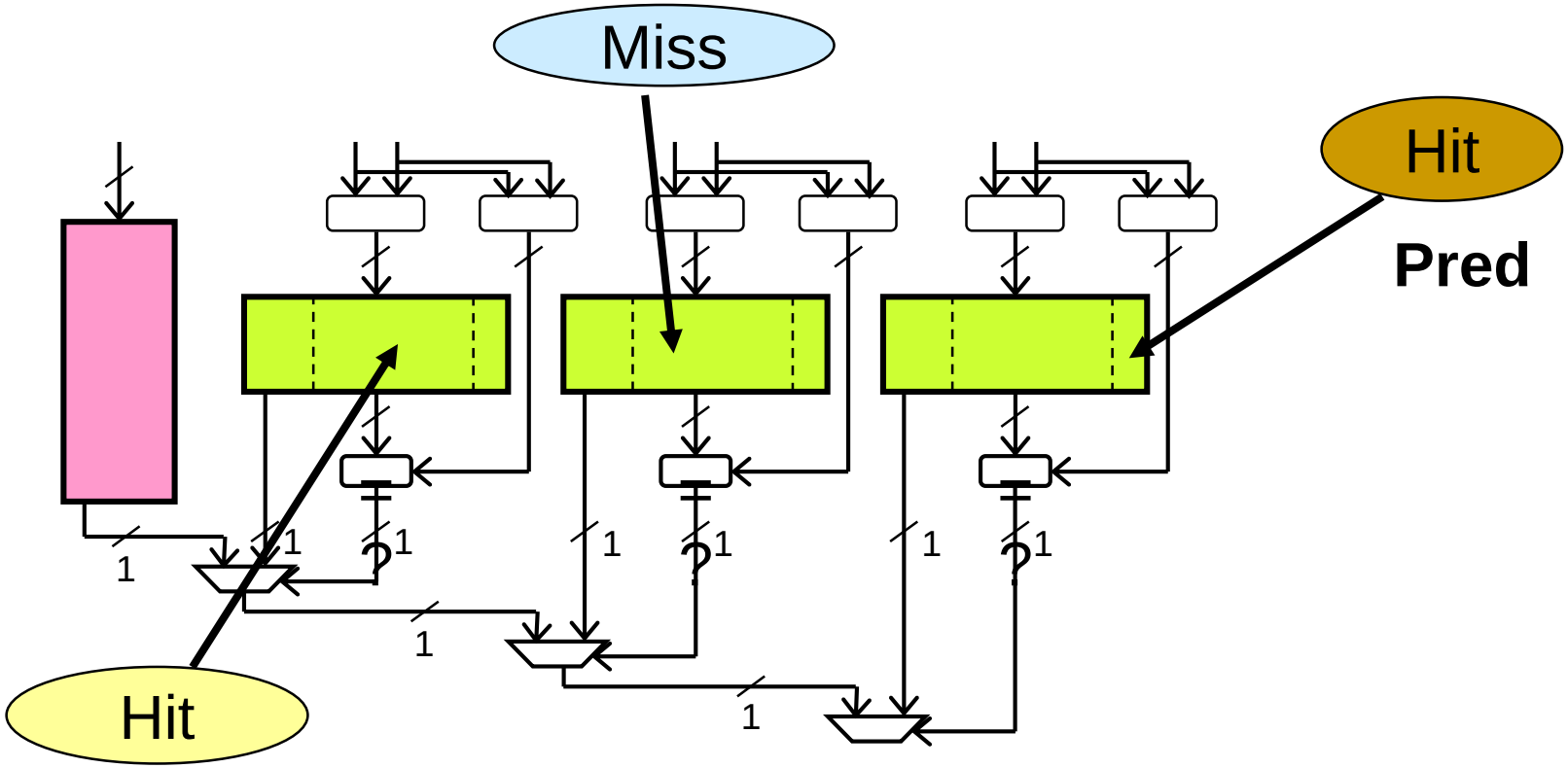
Seznec and Michaud, "A case for (partially) tagged Geometric History Length Branch Prediction," JILP 2006.

TAGE: Tagged & prediction by the longest history matching entry



Andre Seznec, "TAGE-SC-L branch predictors again," CBP 2016.

TAGE: Multiple Tables



Altpred: Alternative prediction

Andre Seznec, "TAGE-SC-L branch predictors again," CBP 2016.

TAGE: Which Table to Use?

- General case:
  - Longest history-matching component provides the prediction
- Special case:
  - Many mispredictions on newly allocated entries: weak Ctr

  On many applications, **Altpred** more accurate than **Pred**
  - Property dynamically monitored through 4-bit counters

Andre Seznec, "TAGE-SC-L branch predictors again," CBP 2016.

A Tagged Table Entry

- Ctr: 3-bit prediction counter
- U: 1 or 2-bit counters
  - Was the entry recently useful?
- Tag: partial tag

Andre Seznec, “TAGE-SC-L branch predictors again,” CBP 2016.

State of the Art in Branch Prediction

- See the Branch Prediction Championship
  - https://www.jilp.org/cbp2016/program.html

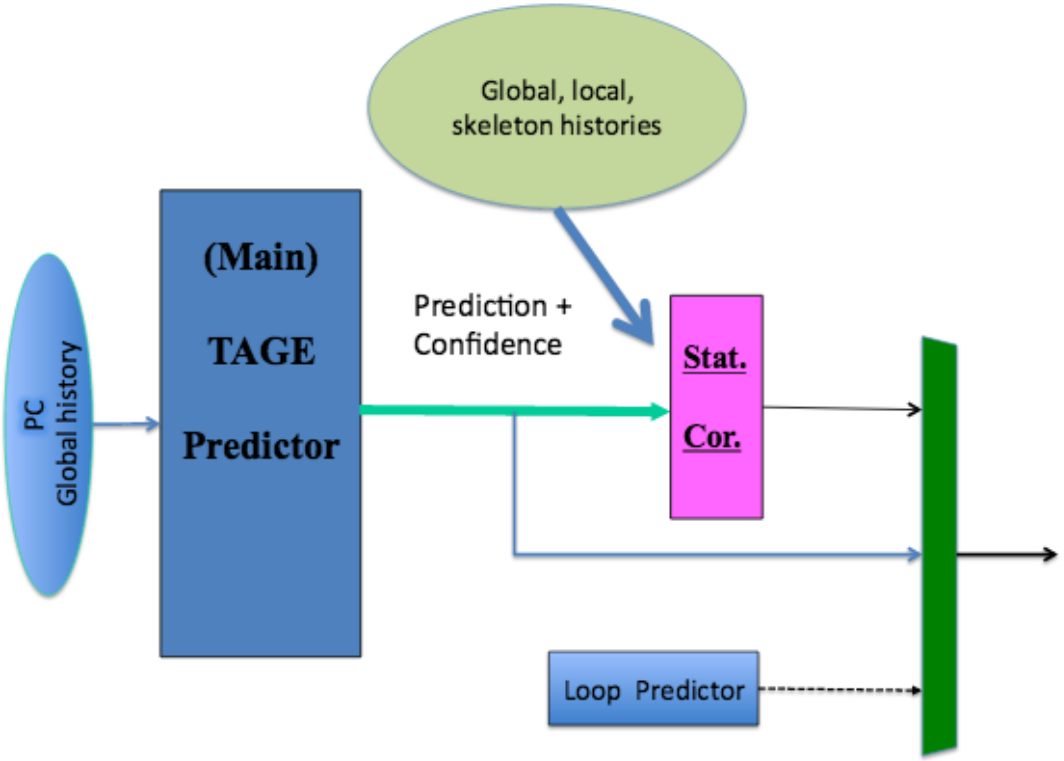


Figure 1. The TAGE-SC-L predictor: a TAGE predictor backed with a Statistical Corrector predictor and a loop predictor

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

Andre Seznec, "TAGE-SC-L branch predictors again," CBP 2016.

Another Direction: Helper Threading

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

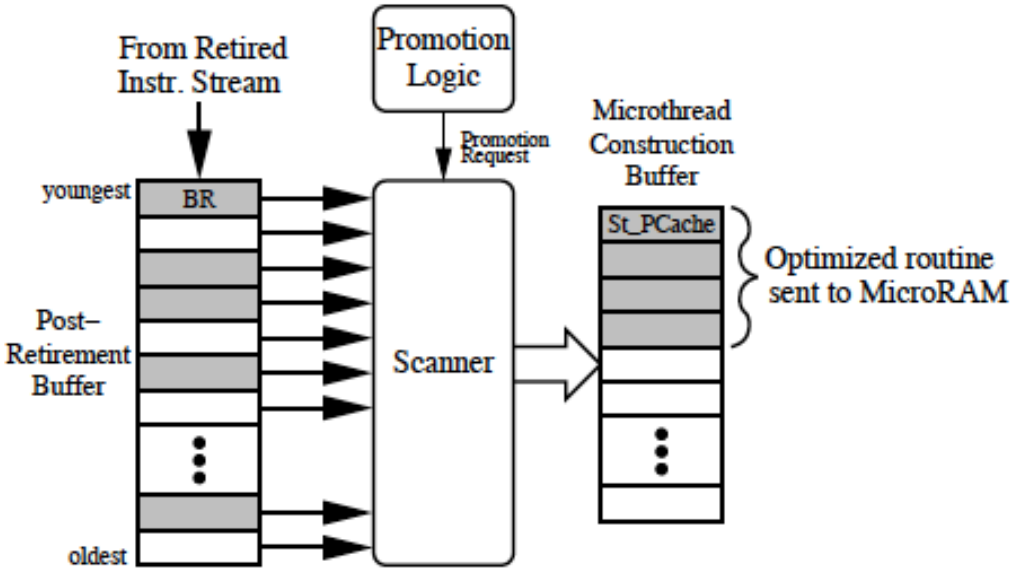


Figure 3. The Microthread Builder

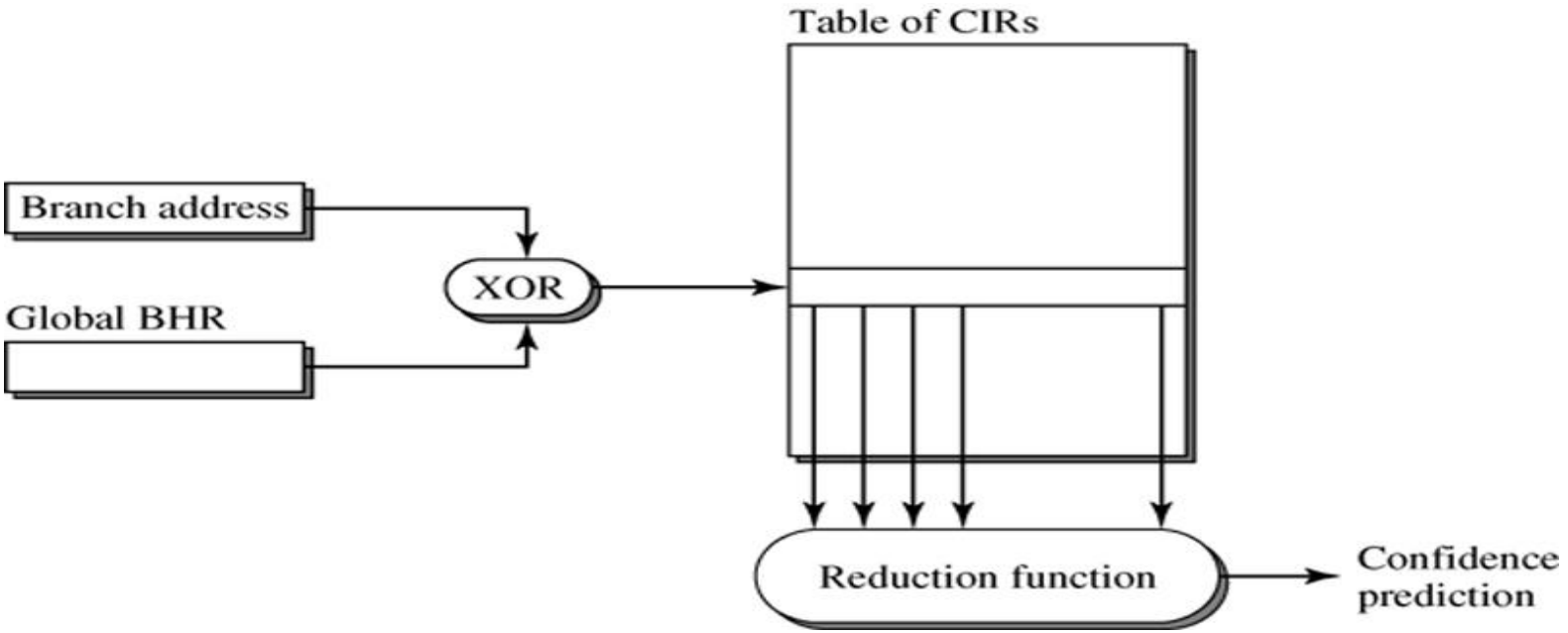
- Chappell et al., "Difficult-Path Branch Prediction Using Subordinate Microthreads," ISCA 2002.
- Chappell et al., "Simultaneous Subordinate Microthreading," ISCA 1999.

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

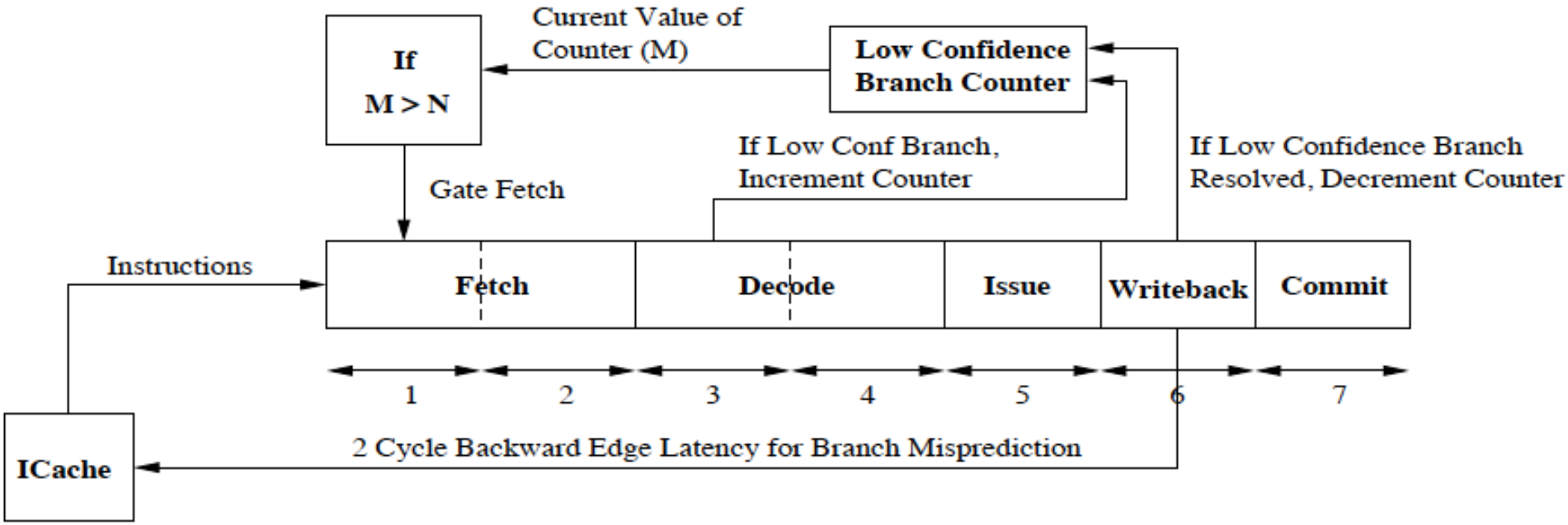
- An example estimator:
  - Keep a record of correct/incorrect outcomes for the past N instances of the "branch"
  - Based on the correct/incorrect patterns, guess if the curent prediction will likely be correct/incorrect



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.

What to Do With Confidence Estimation?

- Another application: Statistical Correction of Prediction

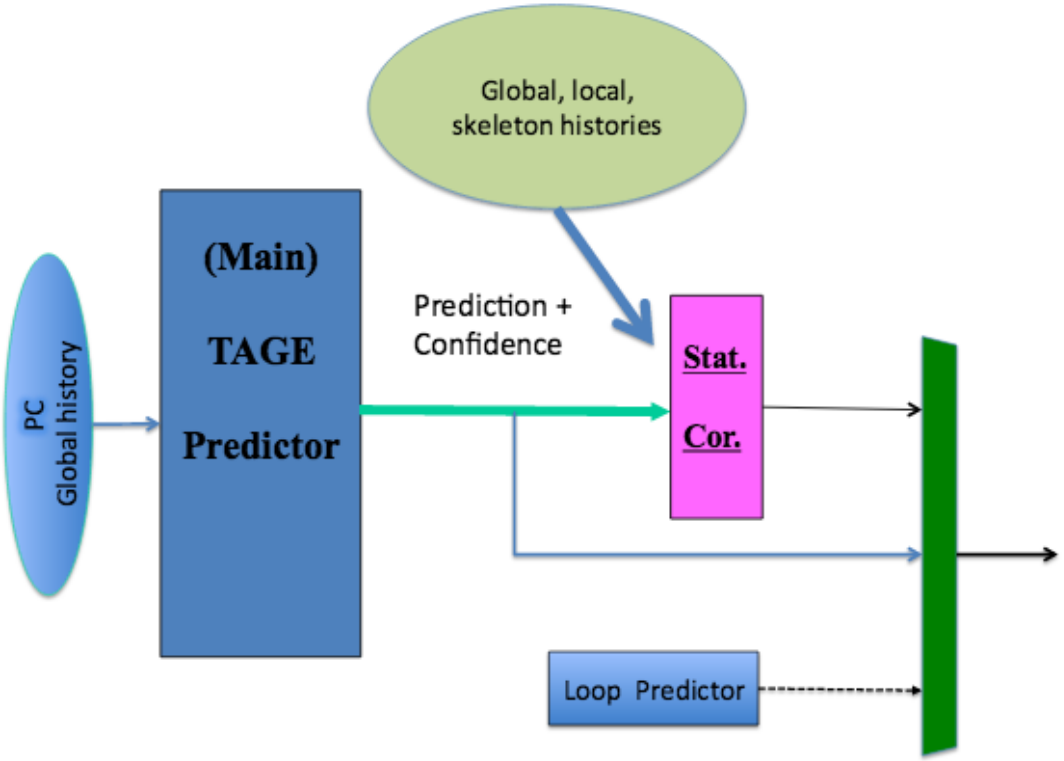


Figure 1. The TAGE-SC-L predictor: a TAGE predictor backed with a Statistical Corrector predictor and a loop predictor

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

Andre Seznec, "TAGE-SC-L branch predictors again," CBP 2016.

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: Predicted the 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

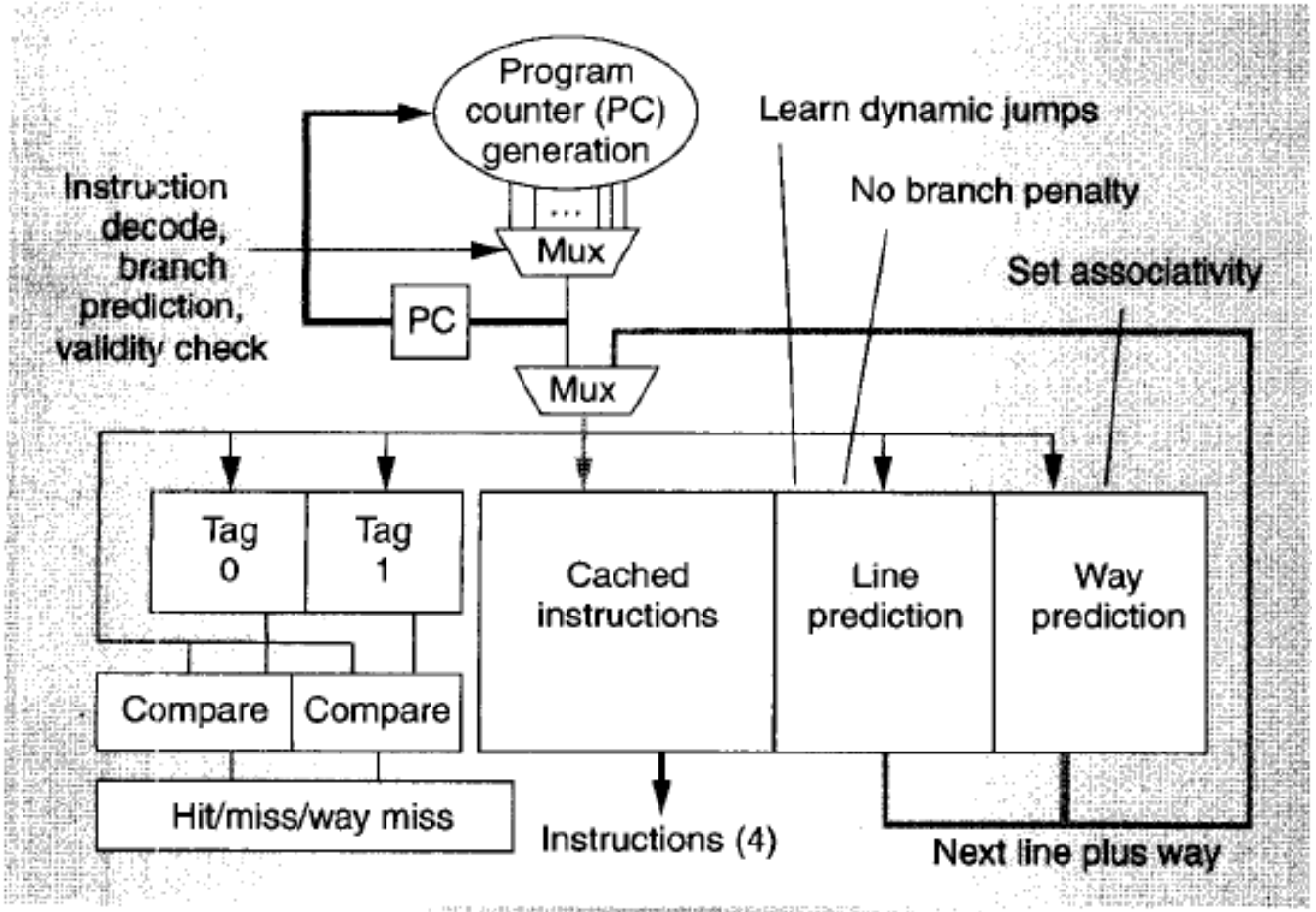
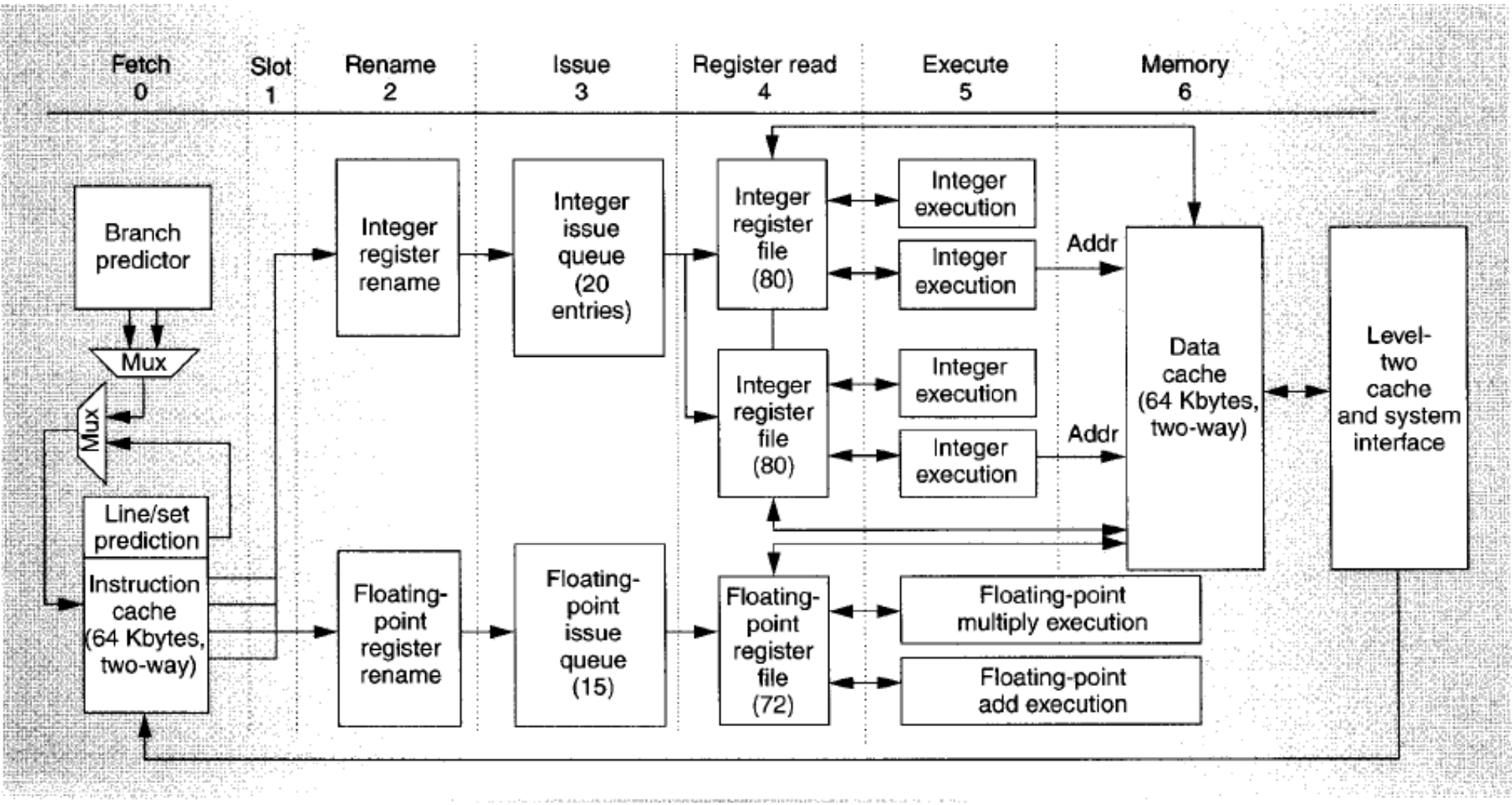


Figure 3. Alpha 21264 instruction fetch. The line and way prediction (wrap-around path on the right side) provides a fast instruction fetch path that avoids common fetch stalls when the predictions are correct.

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

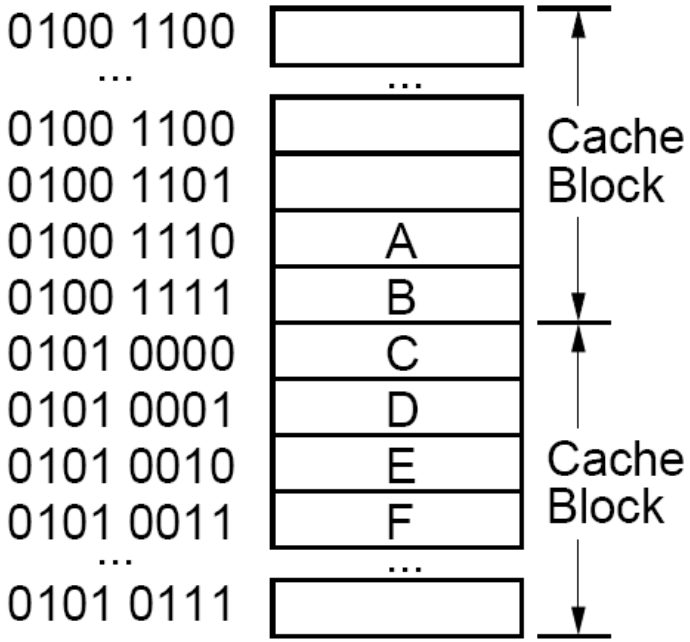
- Large cache blocks: Hope N instructions are contained in the block

- 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

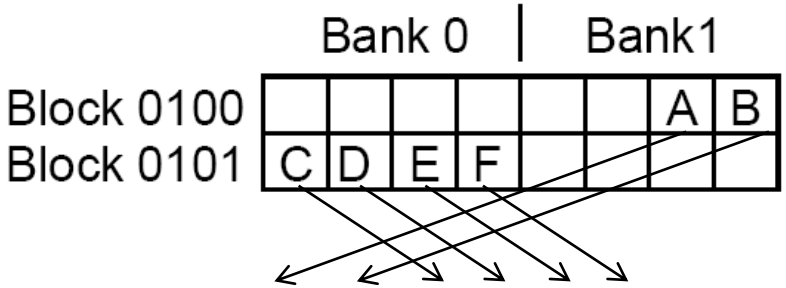
Split Line Fetch

Cache Banking

Memory Map

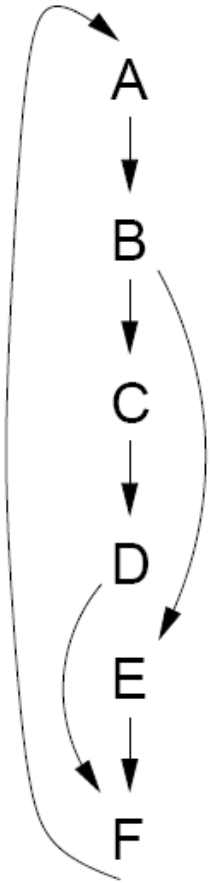


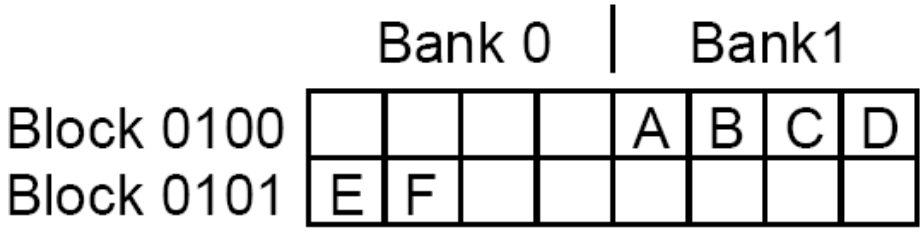
Cache



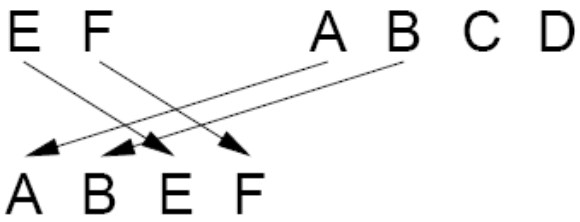
Need alignment logic:

Short Distance Predicted-Taken Branches

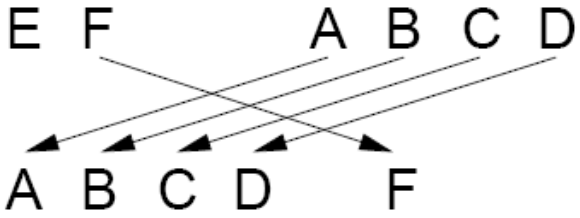




First Iteration (Branch B taken to E)



Second Iteration (Branch B fall through to C)

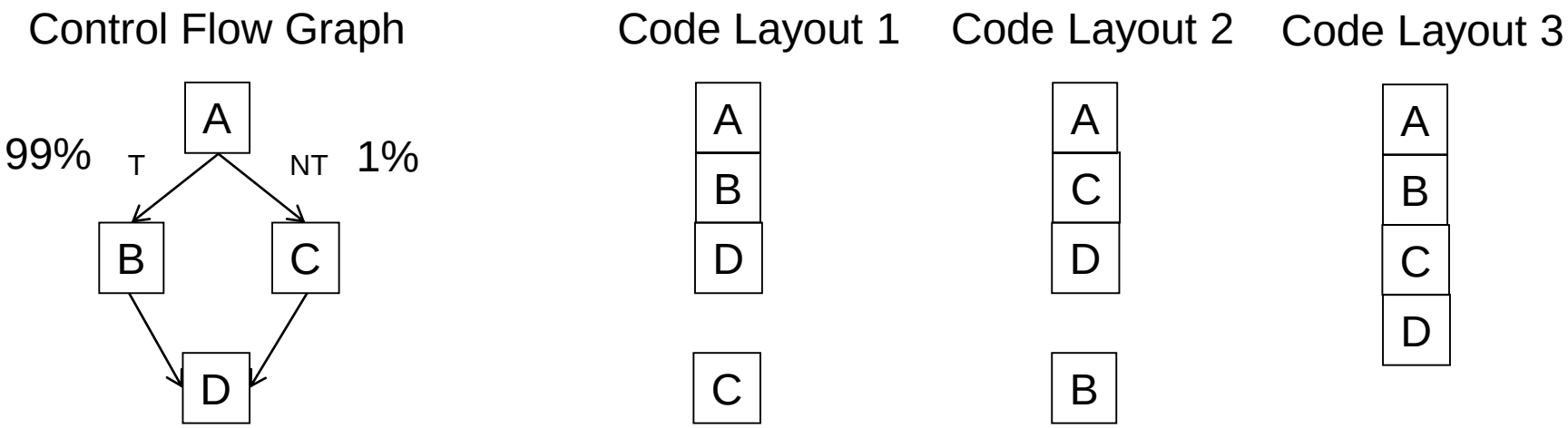


Techniques to Reduce Fetch Breaks

- 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

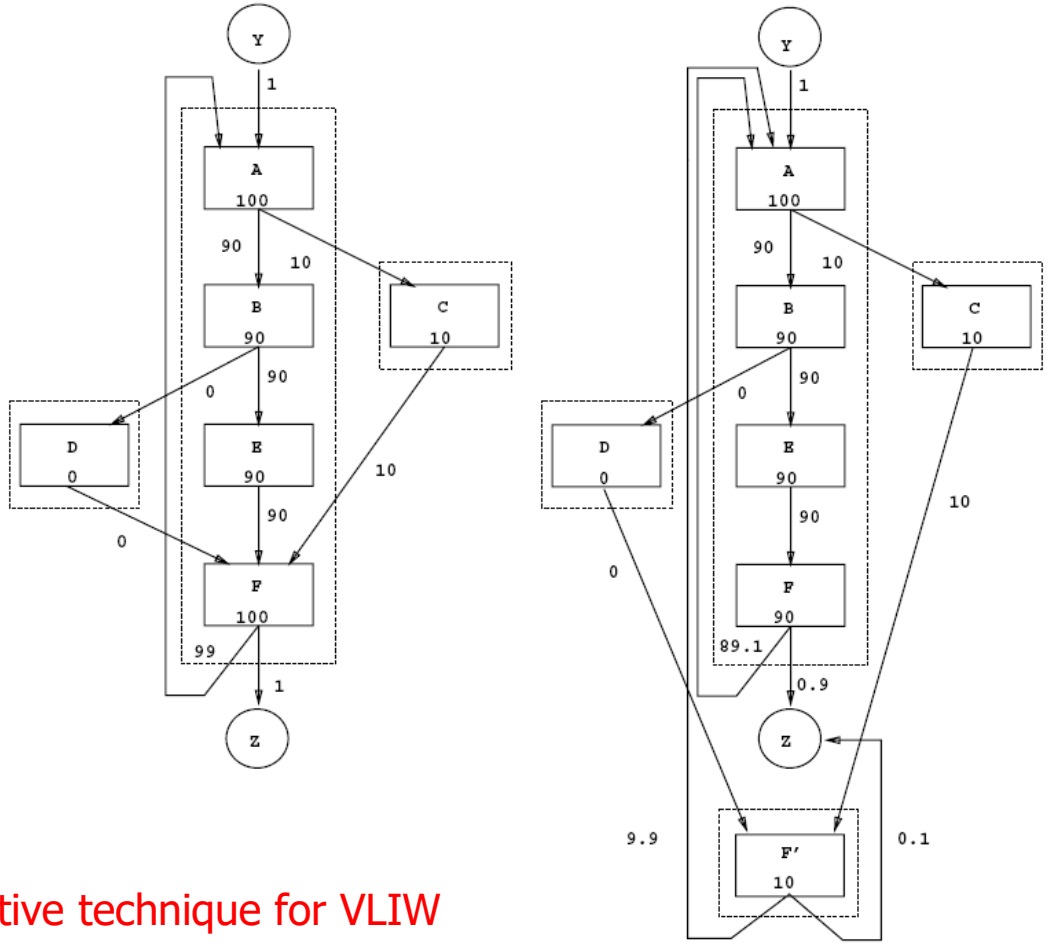
- Disadvantages:
  -- Dependent on compile-time profiling
  -- Does not help if branches are not biased
  -- Requires recompilation

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

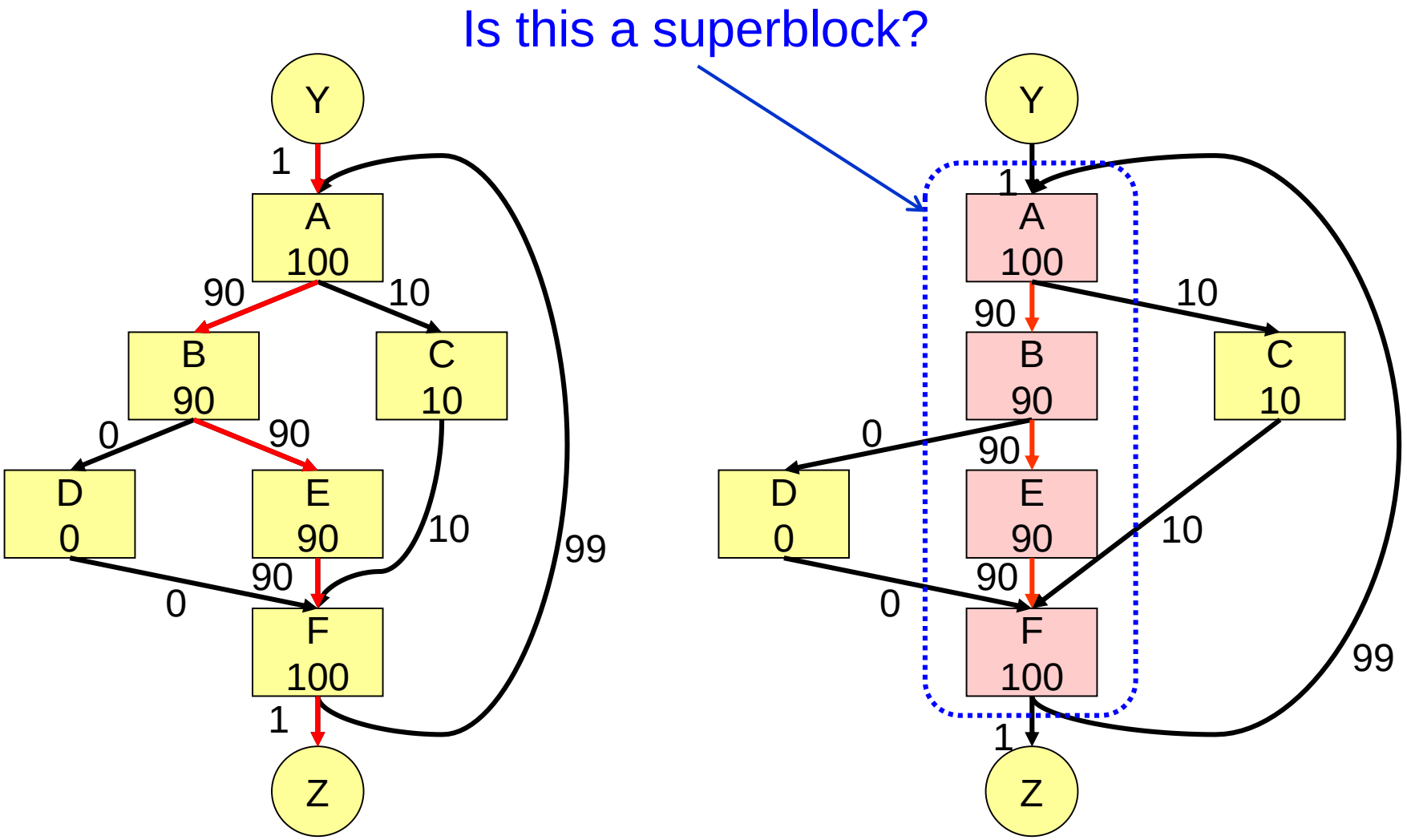
+ Helps wide fetch
+ Enables aggressive compiler optimizations and code reordering within the superblock

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

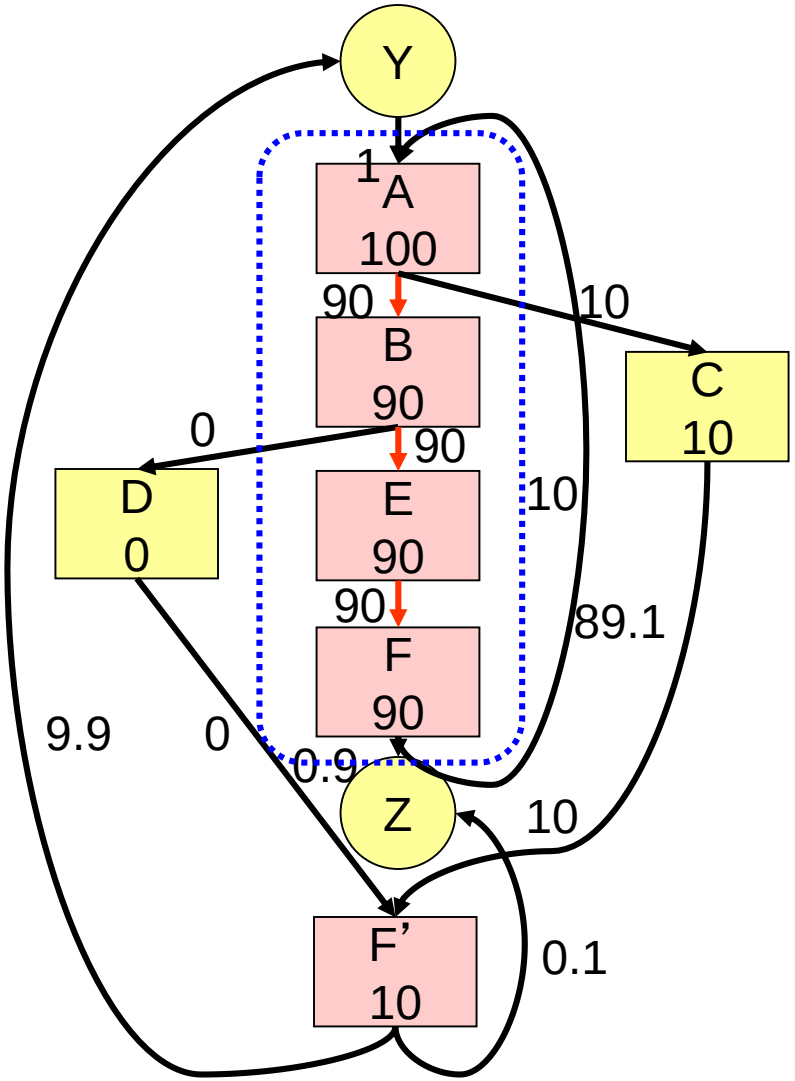


- Hwu et al. "The Superblock: An effective technique for VLIW and superscalar compilation," Journal of Supercomputing, 1993.

Superblock Formation (I)



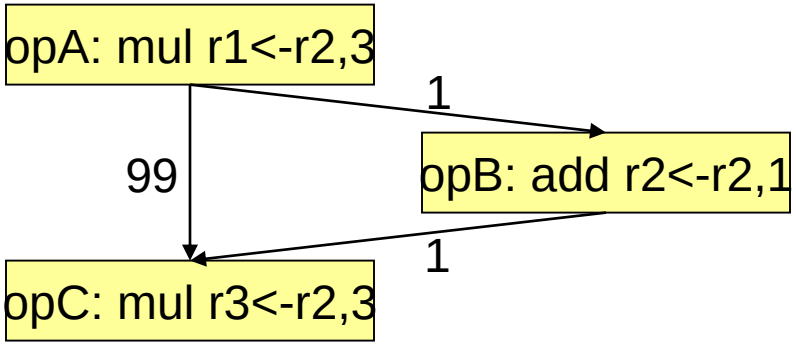
Superblock Formation (II)



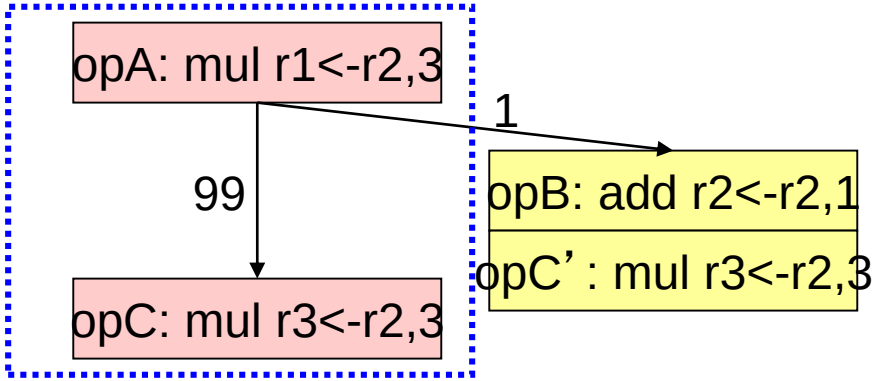
Tail duplication:
duplication of basic blocks after a side entrance to eliminate side entrances

→ transforms
a trace into a superblock.

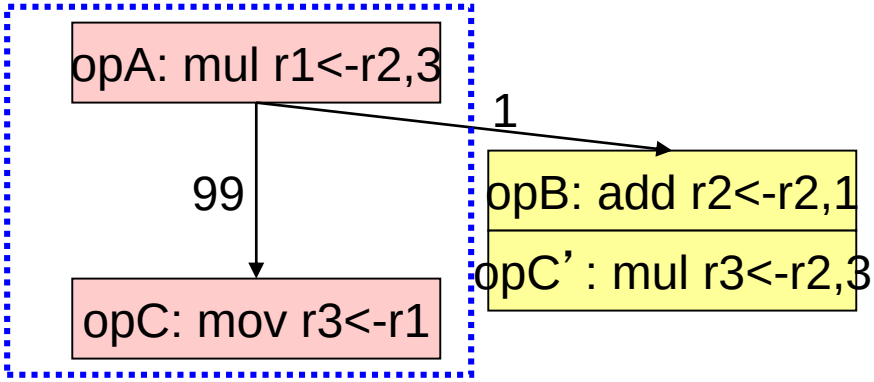
Superblock Code Optimization Example



Original Code



Code After Superblock Formation



Code After Common Subexpression Elimination

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Computer Architecture

Lecture 10: Branch Prediction

Prof. Onur Mutlu
ETH Zürich

Fall 2017

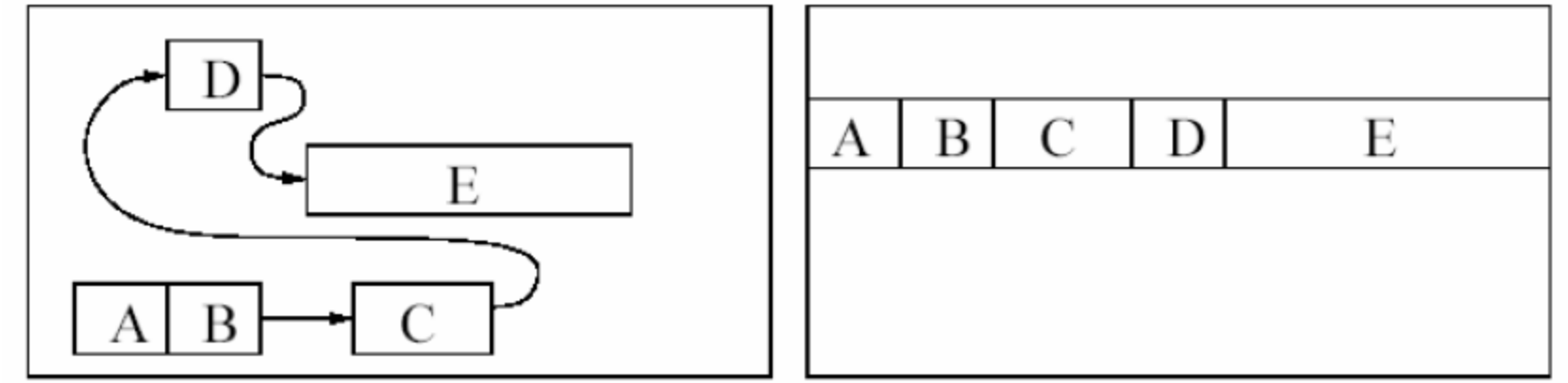
25 October 2017

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

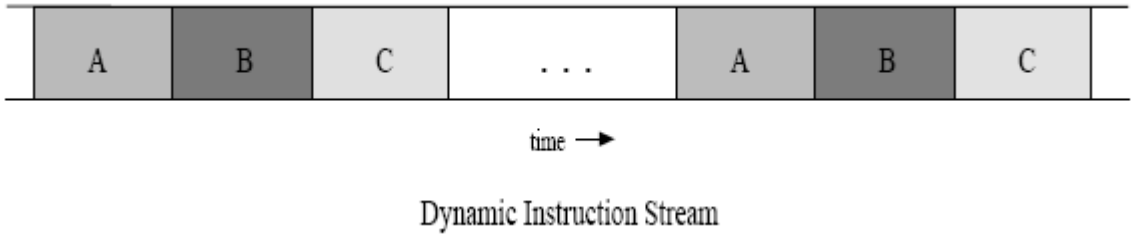
- A trace is a sequence of executed instructions.
- It is specified by a start address and the outcomes of control transfer instructions within the trace.
- Traces repeat: programs have frequently executed paths
- Trace cache idea: Store a dynamic instruction sequence in the same physical location so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



(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)

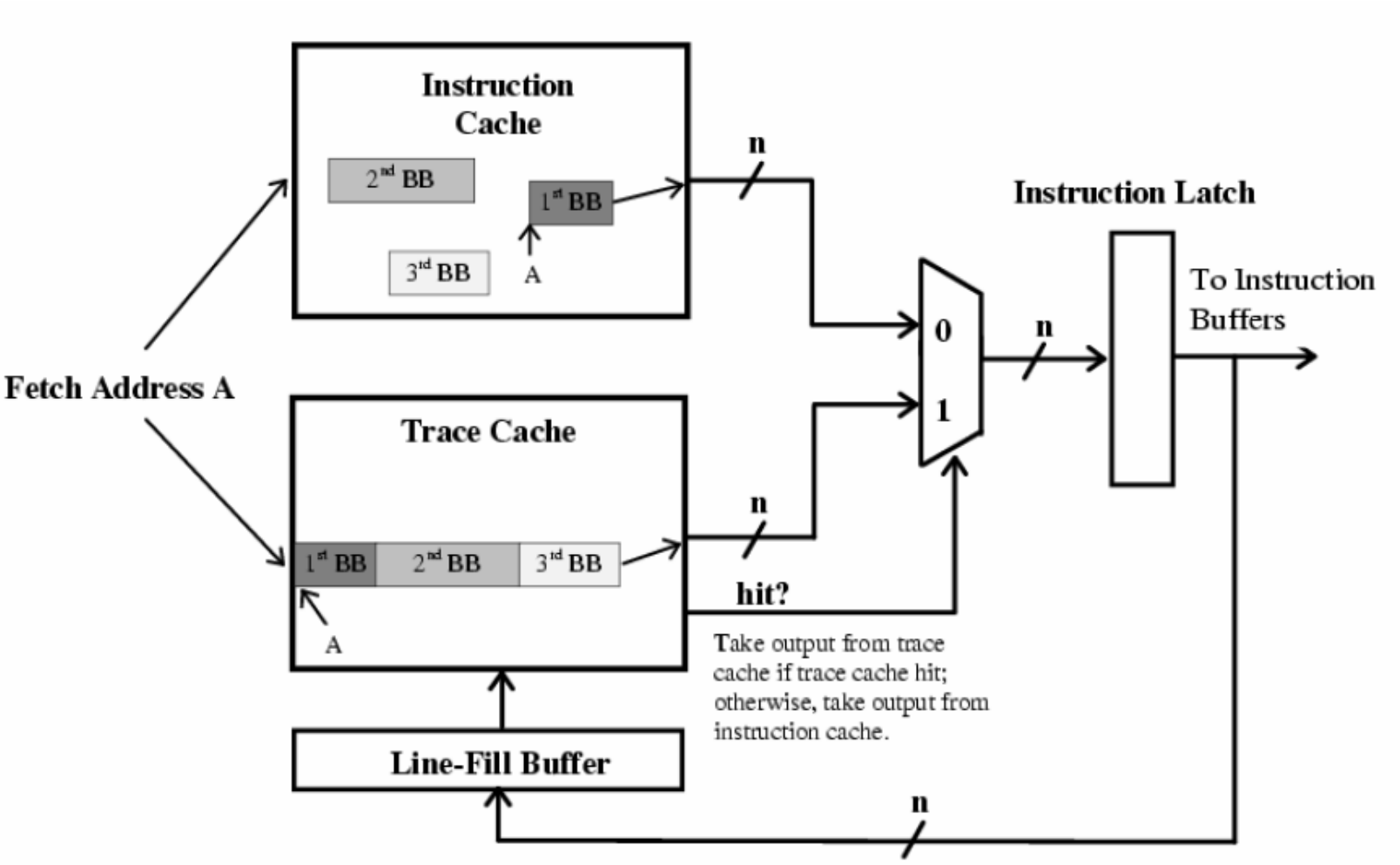


Dynamic Instruction Stream

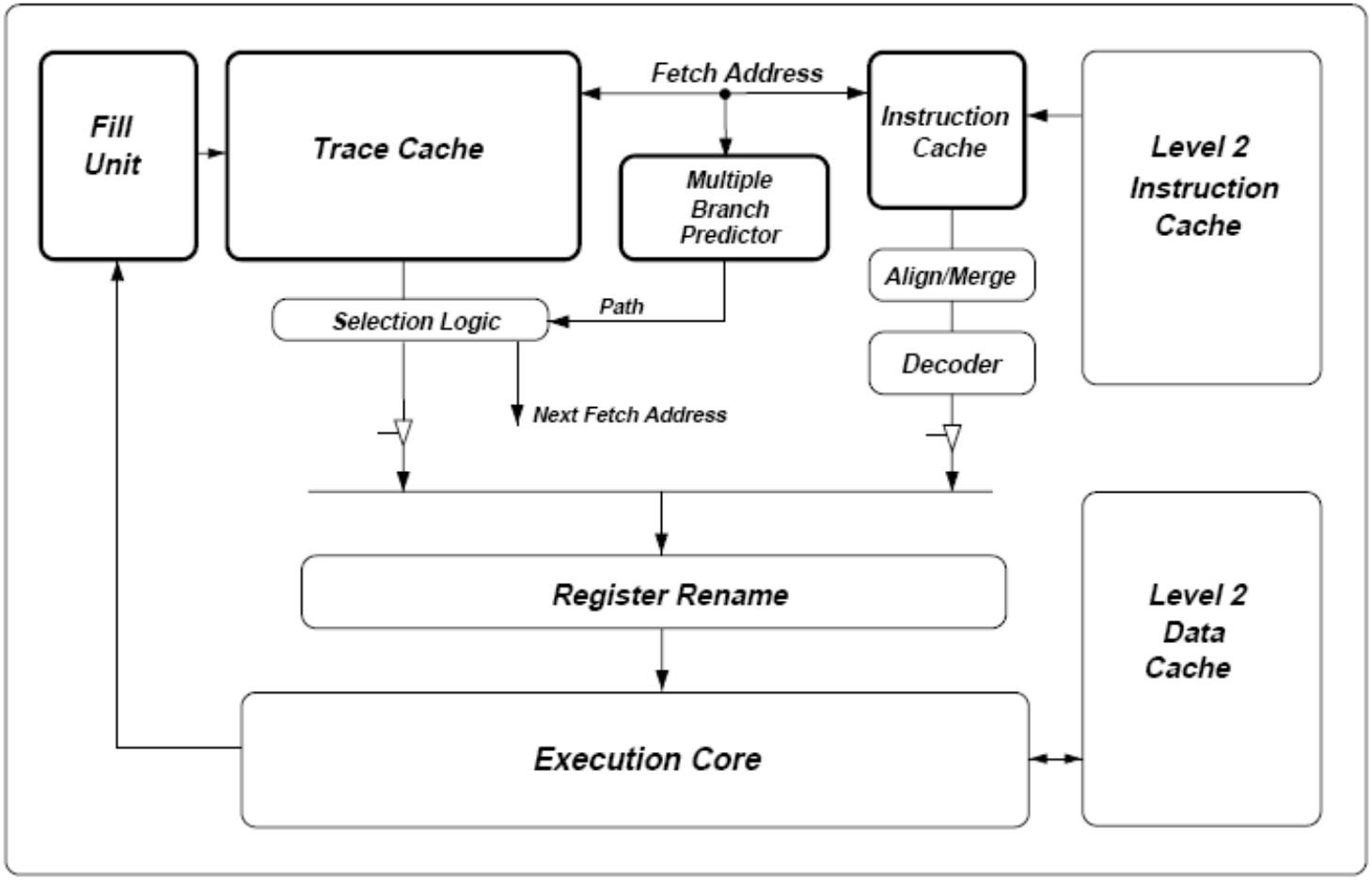
- Basic trace cache operation:
  - Fetch from consecutively-stored basic blocks (predict next trace or branches)
  - Verify the executed branch directions with the stored ones
  - If mismatch, flush the remaining portion of the trace

- Rotenberg et al., "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," MICRO 1996. **Received the MICRO Test of Time Award 20 years later**
- Patel et al., "Critical Issues Regarding the Trace Cache Fetch Mechanism," Umich TR, 1997.

Trace Cache: Example



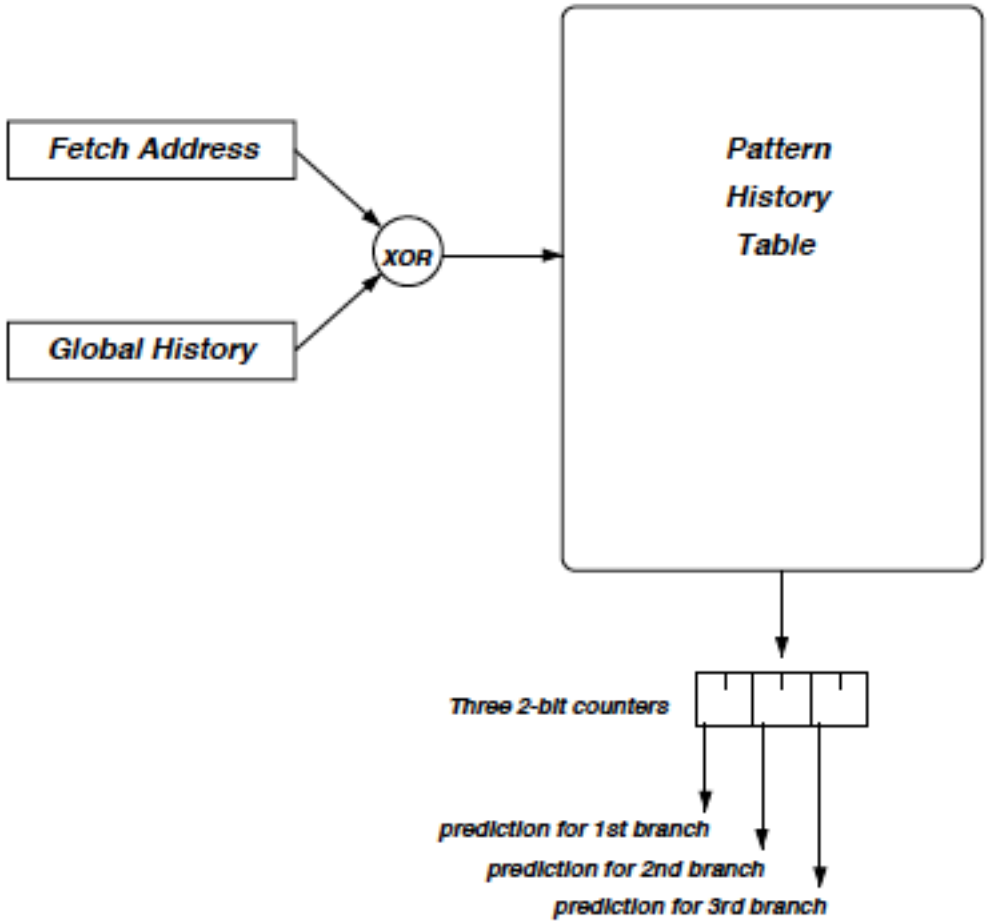
An Example Trace Cache Based Processor



- 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.

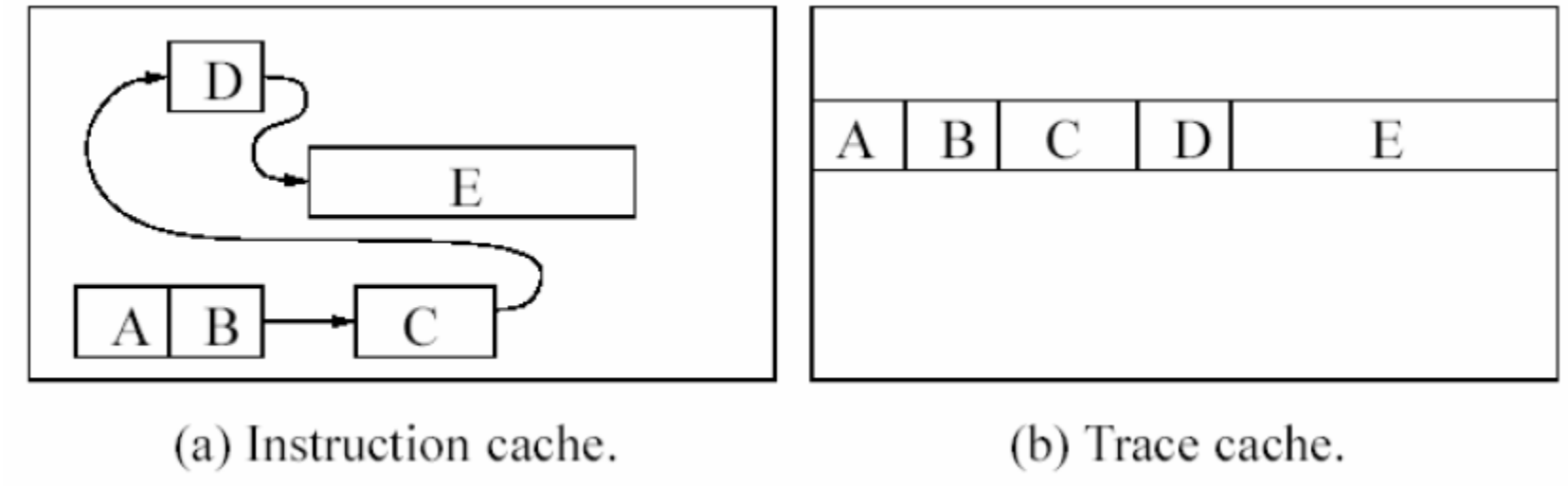


What Does A Trace Cache Line Store?

- 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

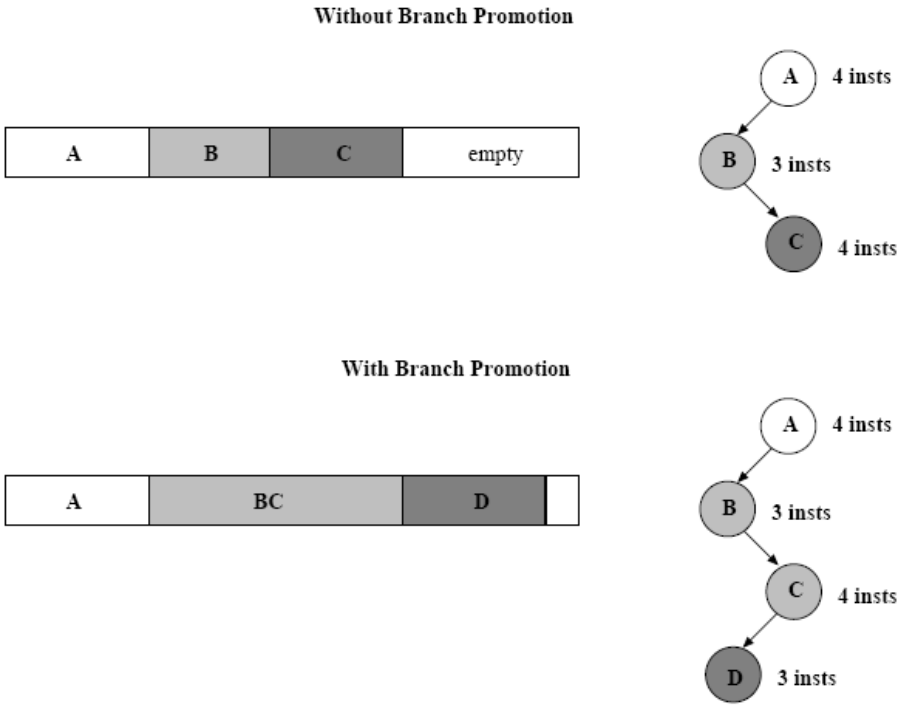


(a) Instruction cache. (b) Trace cache.

+ 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?

Trace Cache Design Issues: Example

- Branch promotion: promote highly-biased branches to branches with static prediction
  - + Larger traces
  - + No need for consuming branch predictor BW
  - + Can enable optimizations within trace
  - -- Requires hardware to determine highly-biased branches



How to Determine Biased Branches

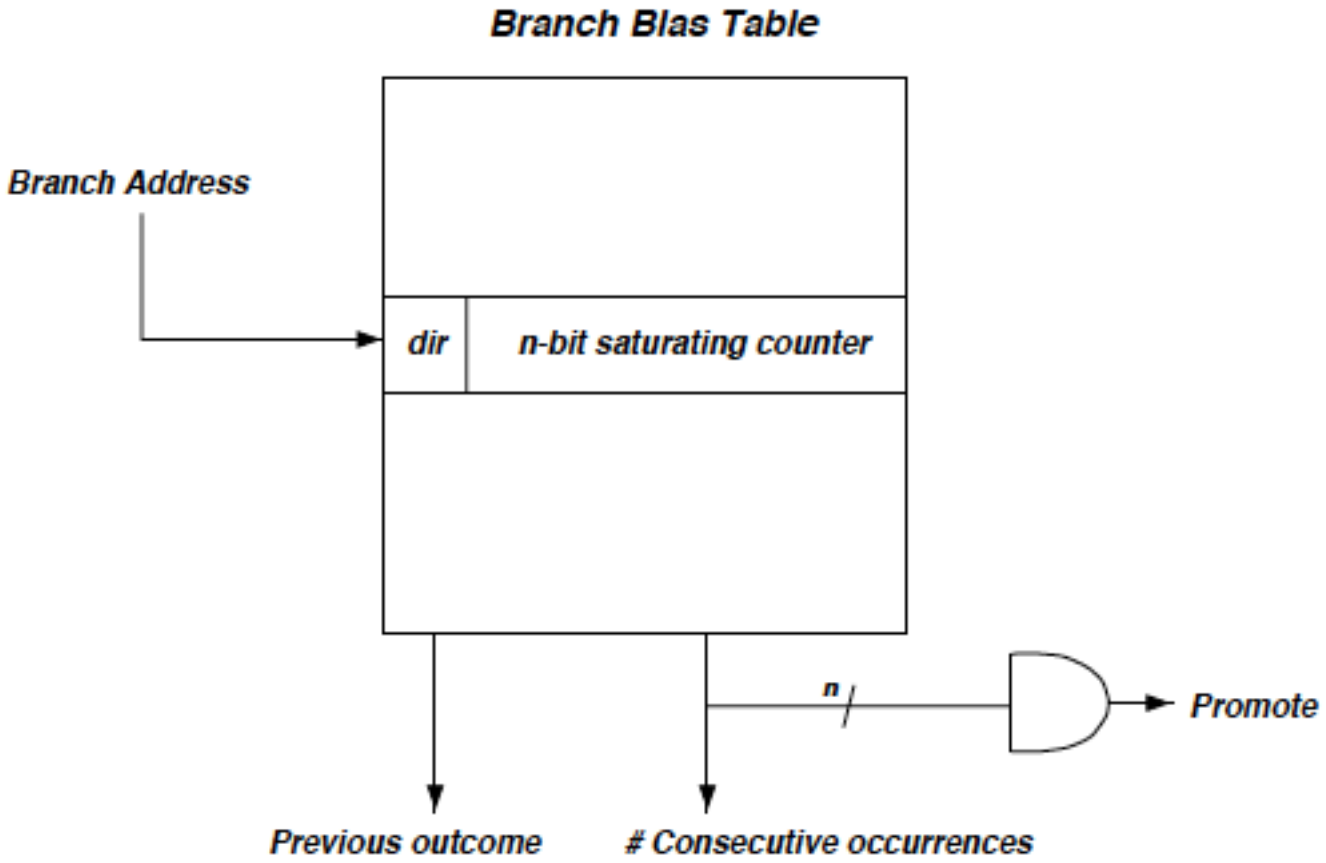
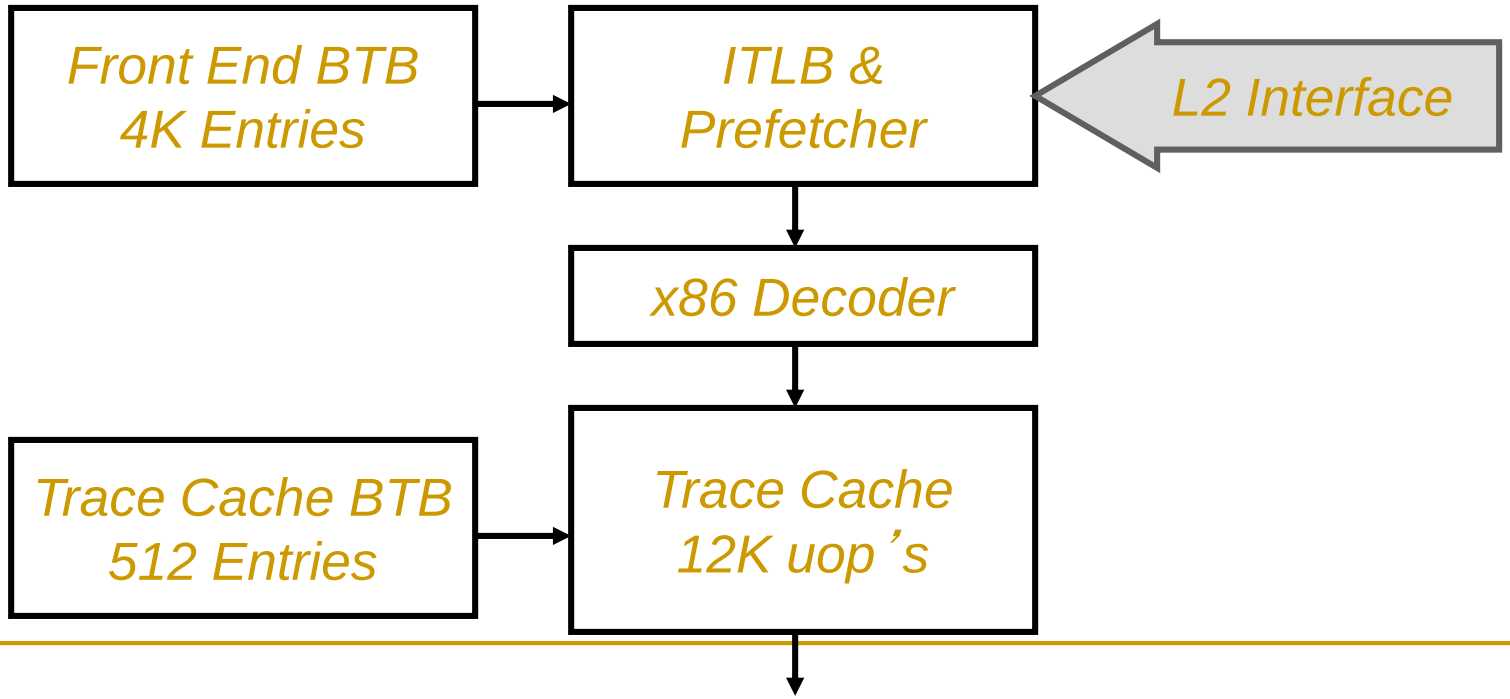


Figure 6.19: Diagram of the branch bias table.

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



Other Ways of Handling Branches

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)

Delayed Branching (I)

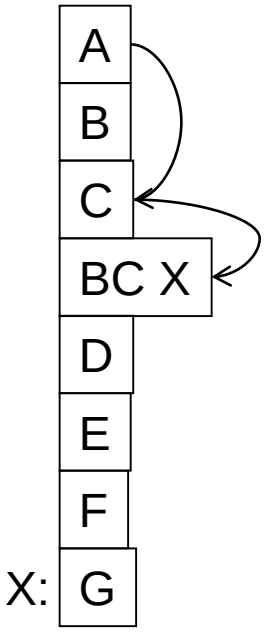
- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles
- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are **always** executed regardless of branch direction.

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

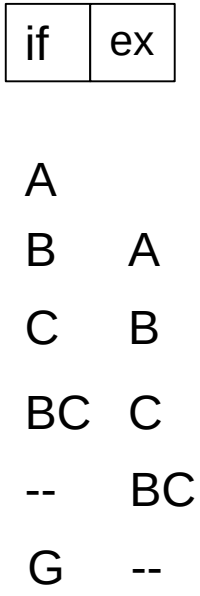
- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot

Delayed Branching (II)

Normal code:

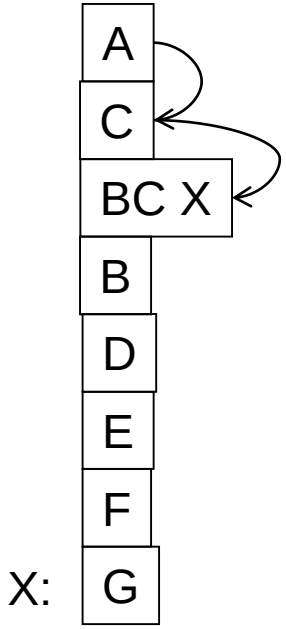


Timeline:

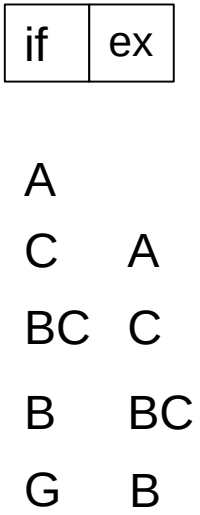


6 cycles

Delayed branch code:



Timeline:

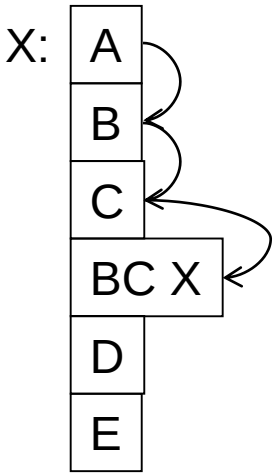


5 cycles

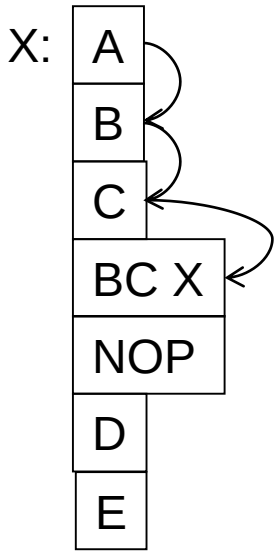
Fancy Delayed Branching (III)

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

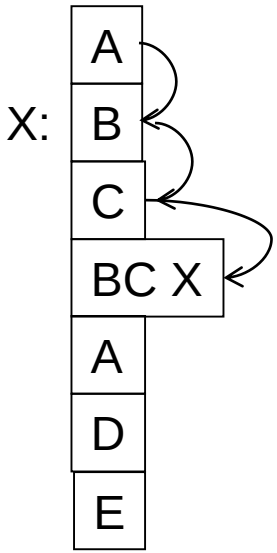
Normal code:



Delayed branch code:



Delayed branch w/ squashing:



Delayed Branching (IV)

- Advantages:

  + Keeps the pipeline full with useful instructions in a simple way 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, superscalar execution width

    2. Number of delay slots should be variable with variable latency operations. Why?

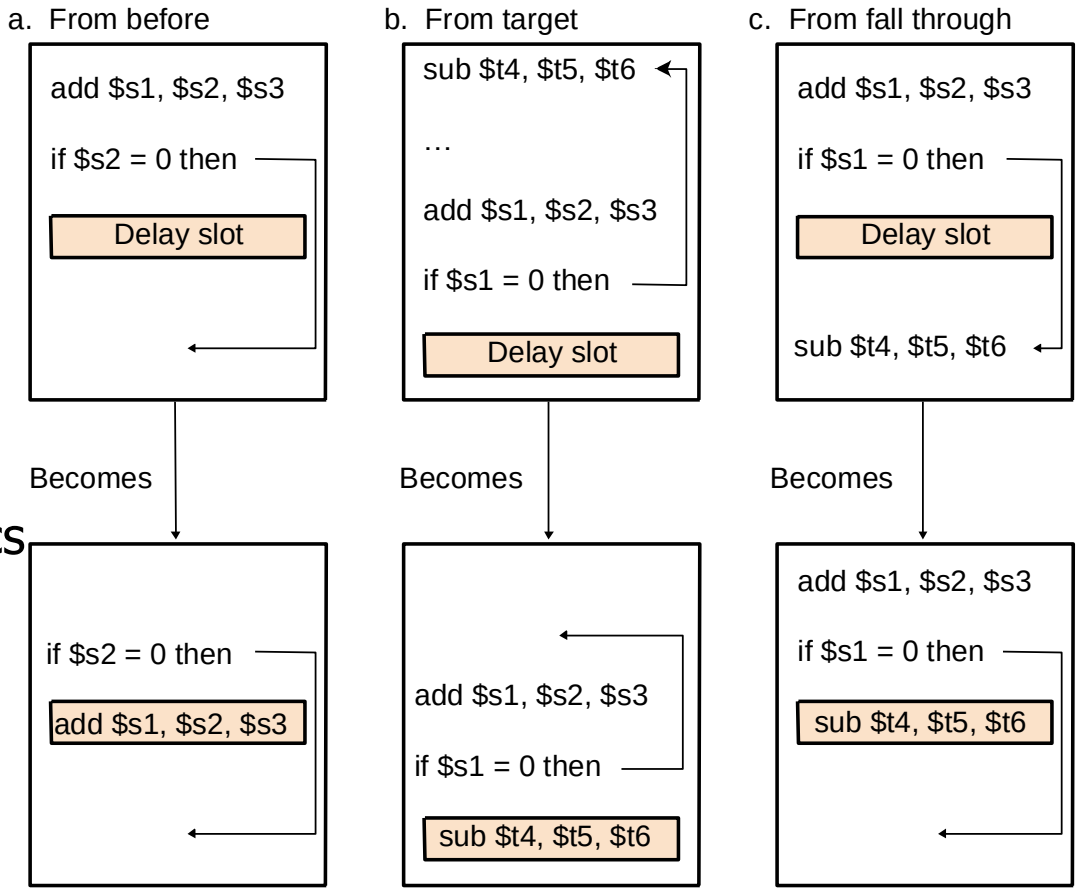
  -- Ties ISA semantics to hardware implementation

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

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

An Aside: Filling the Delay Slot

reordering data independent (RAW, WAW, WAR) instructions does not change program semantics



within same basic block

For correctness: add a new instruction to the not-taken path?

For correctness: add a new instruction to the taken path?

Safe?

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)

Fine-Grained Multithreading

Fine-Grained Multithreading

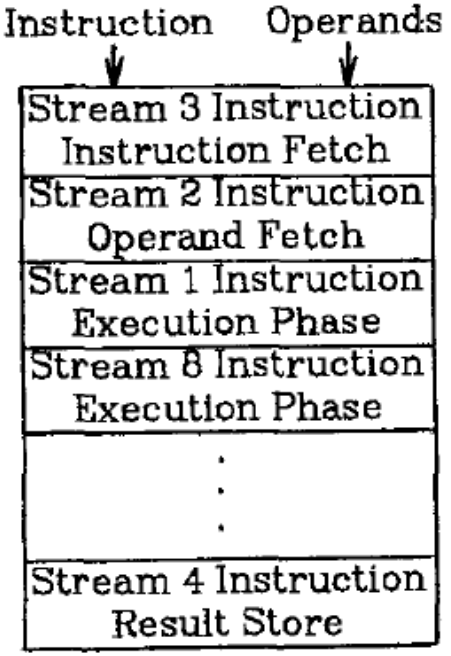
- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads' instructions

+ No logic needed for handling control and data dependences within a thread

-- Single thread performance suffers

-- Extra logic for keeping thread contexts

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



Fine-Grained Multithreading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently

- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads

- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

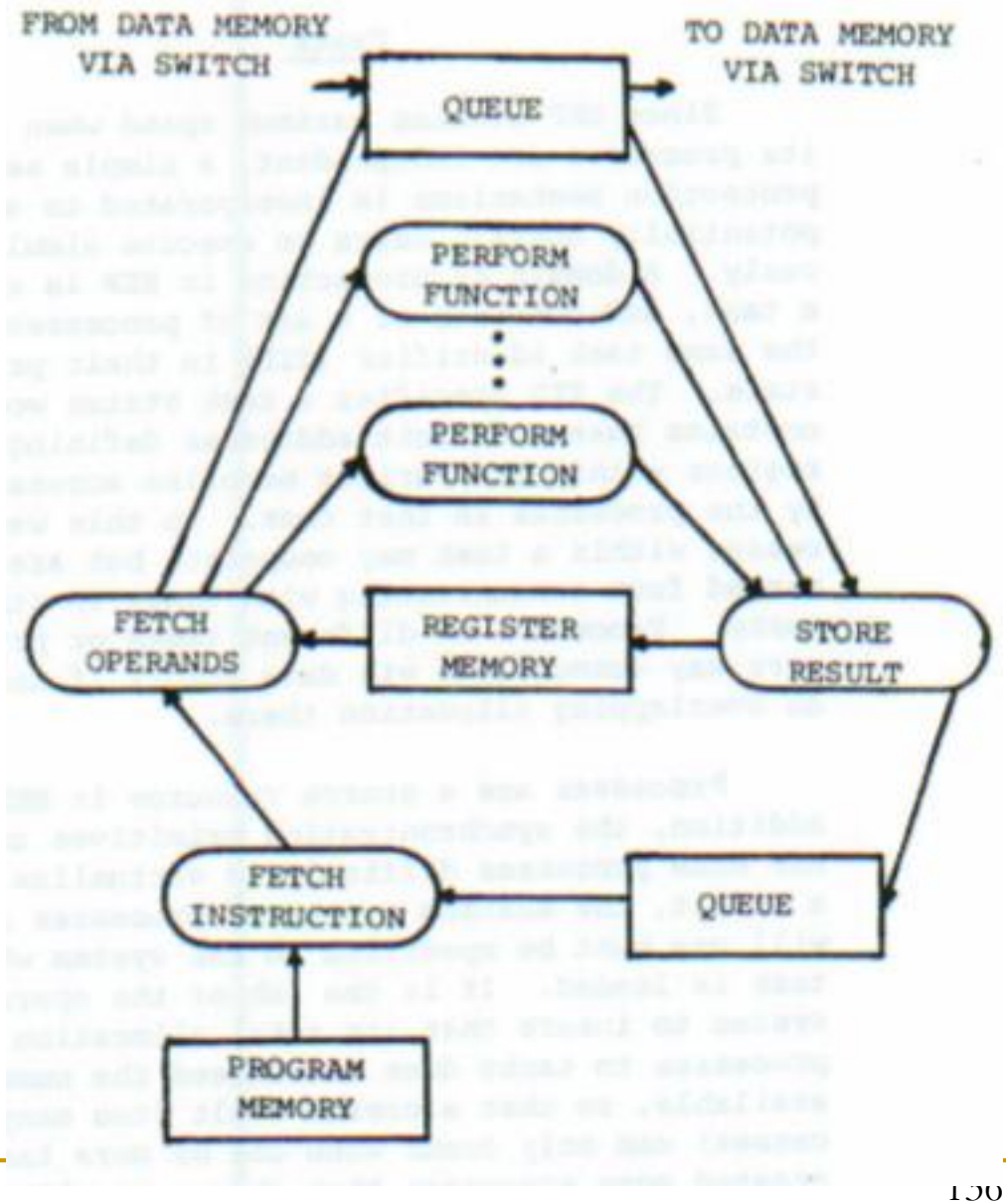
Fine-Grained Multithreading: History

- CDC 6600's peripheral processing unit is fine-grained multithreaded
  - Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
  - Processor executes a different I/O thread every cycle
  - An operation from the same thread is executed every 10 cycles

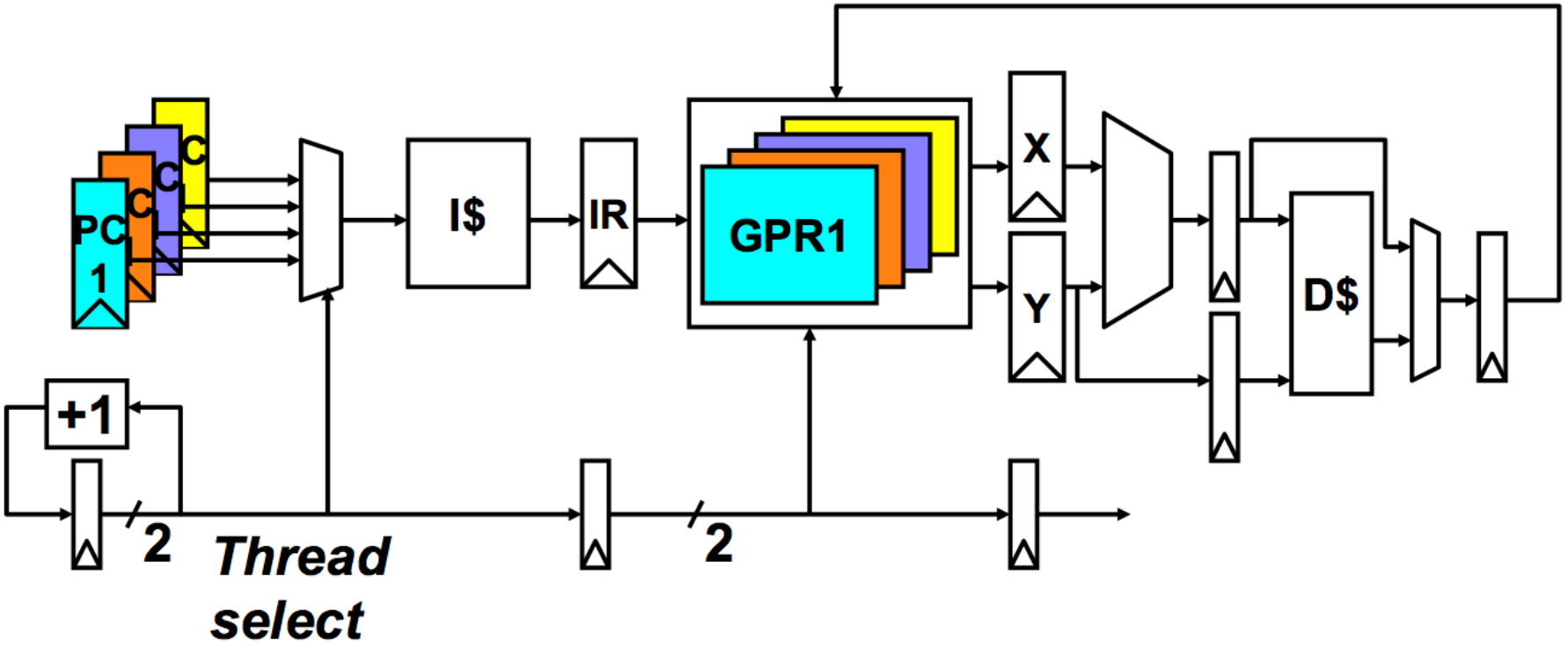
- Denelcor HEP (Heterogeneous Element Processor)
  - Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
  - 120 threads/processor
  - available queue vs. unavailable (waiting) queue for threads
  - each thread can have only 1 instruction in the processor pipeline; each thread independent
  - to each thread, processor looks like a non-pipelined machine
  - system throughput vs. single thread performance tradeoff

Fine-Grained Multithreading in HEP

- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
  - assuming no memory access
- No control and data dependency checking

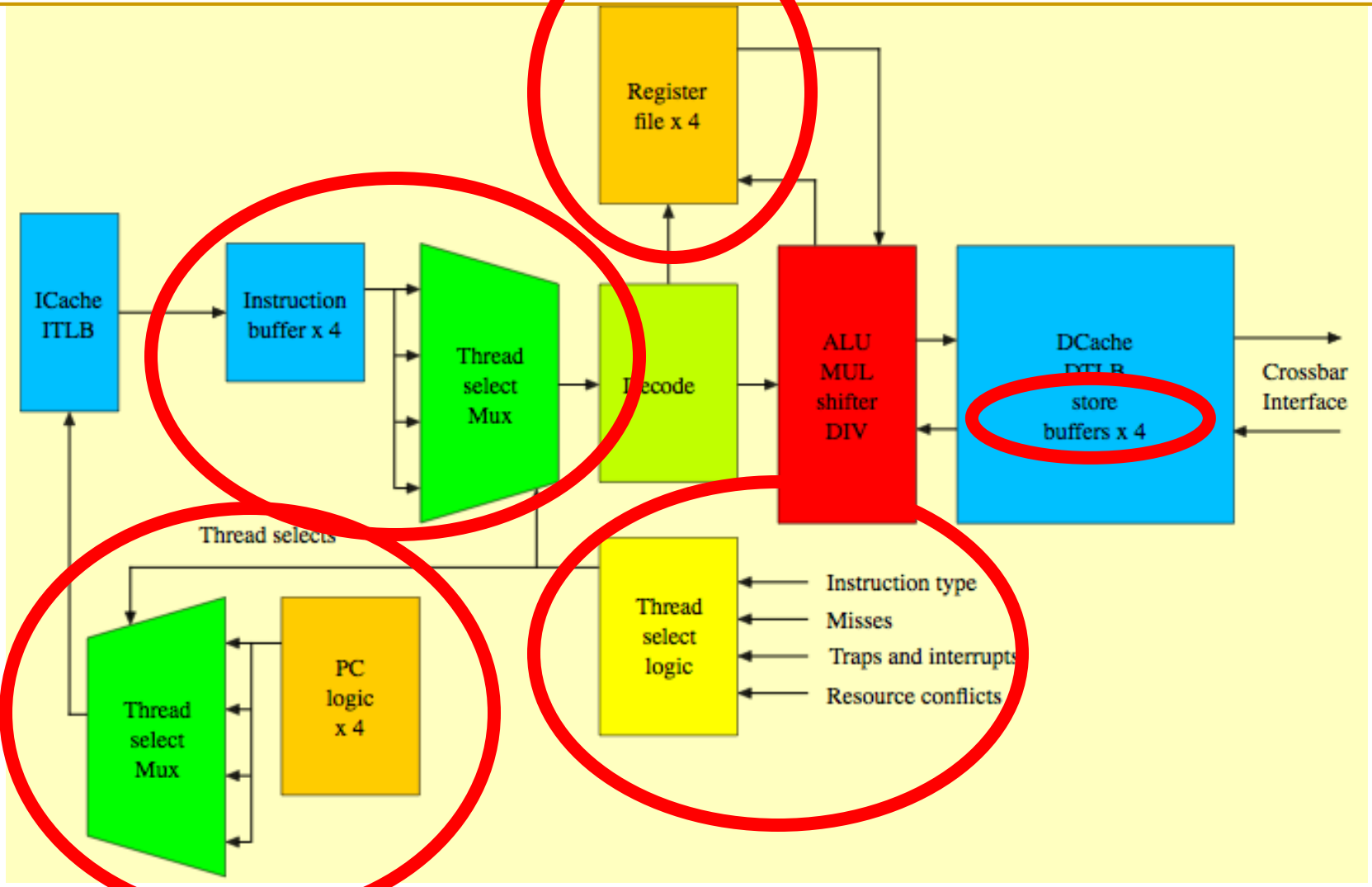


Multithreaded Pipeline Example



Slide credit: Joel Emer

Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

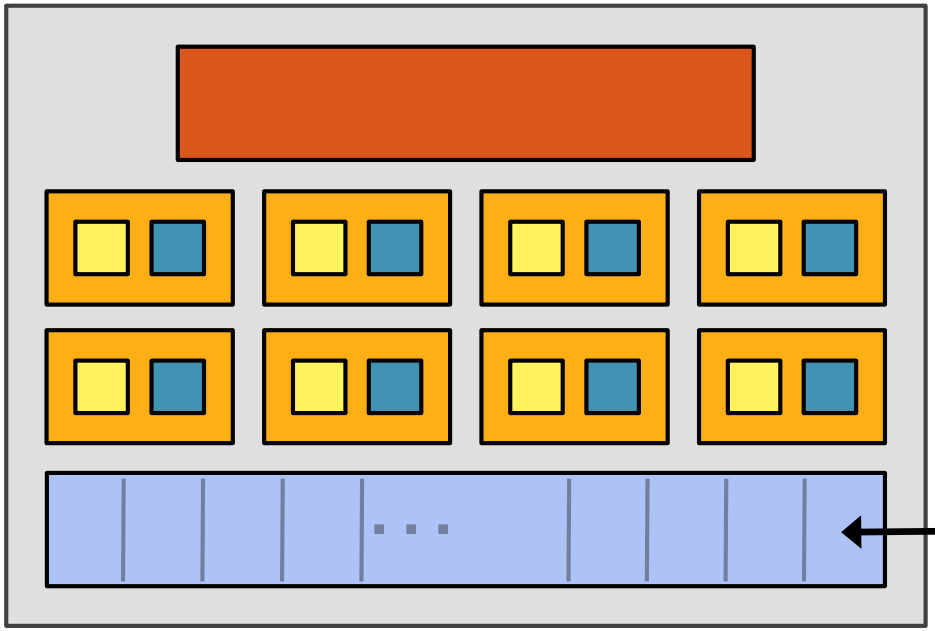
Fine-grained Multithreading

- Advantages
  - + No need for dependency checking between instructions (only one instruction in pipeline from a single thread)
  - + No need for branch prediction logic
  - + Otherwise-bubble cycles used for executing useful instructions from different threads
  - + Improved system throughput, latency tolerance, utilization

- Disadvantages
  - - Extra hardware complexity: multiple hardware contexts (PCs, register files, …), thread selection logic
  - - Reduced single thread performance (one instruction fetched every N cycles from the same thread)
  - - Resource contention between threads in caches and memory
  - - Some dependency checking logic *between* threads remains (load/store)

Modern GPUs Are FGMT Machines

NVIDIA GeForce GTX 285 "core"



64 KB of storage for thread contexts (registers)

= data-parallel (SIMD) func. unit, control shared across 8 units

= multiply-add

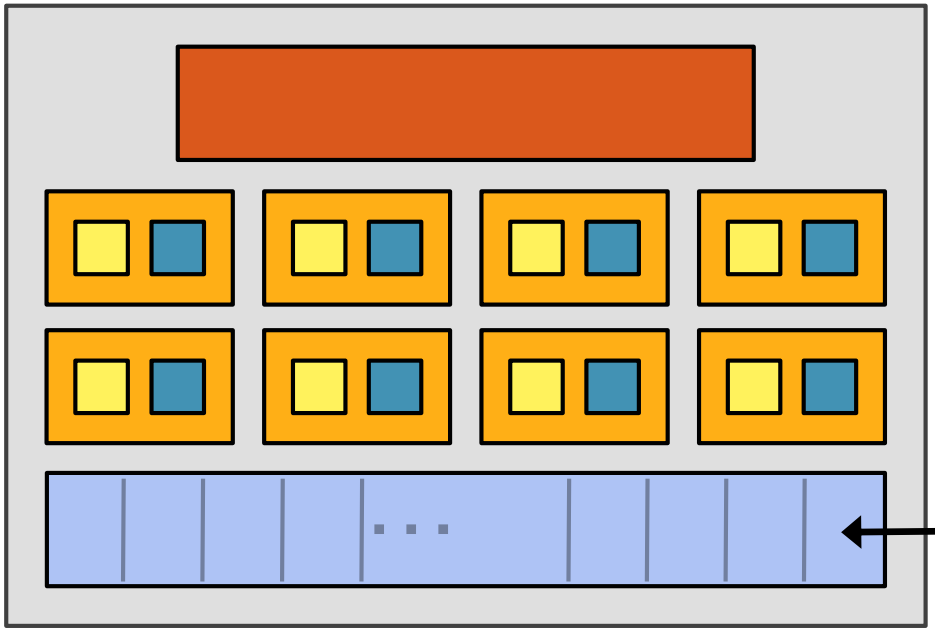
= multiply

= instruction stream decode

= execution context storage

Slide credit: Kayvon Fatahalian

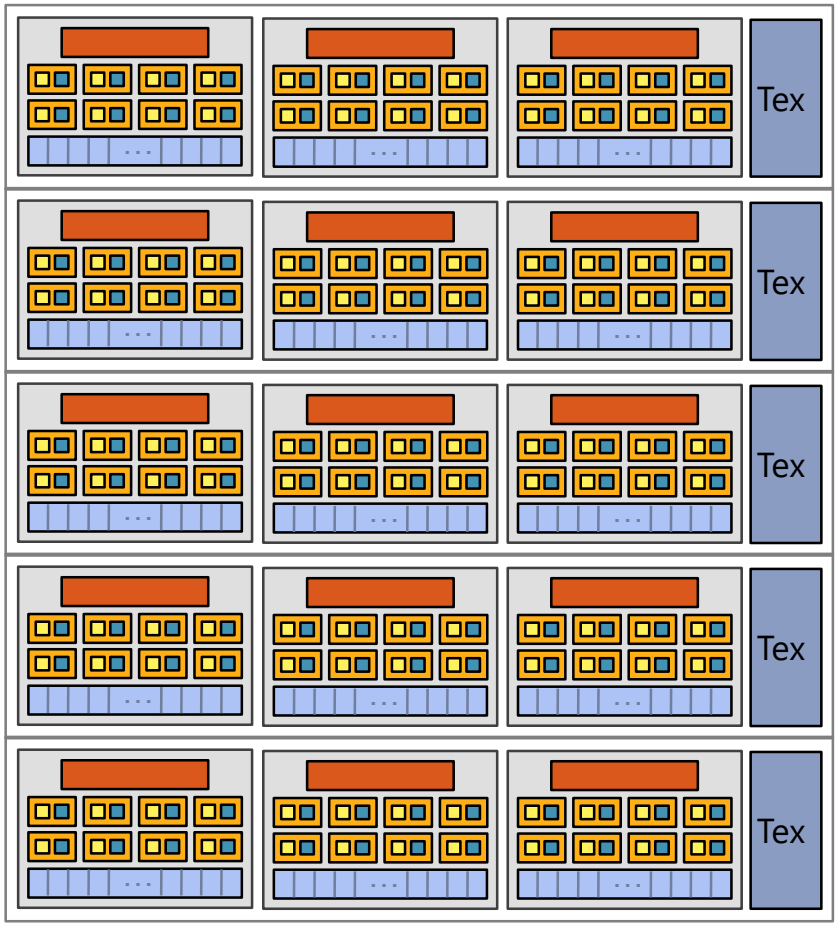
NVIDIA GeForce GTX 285 "core"

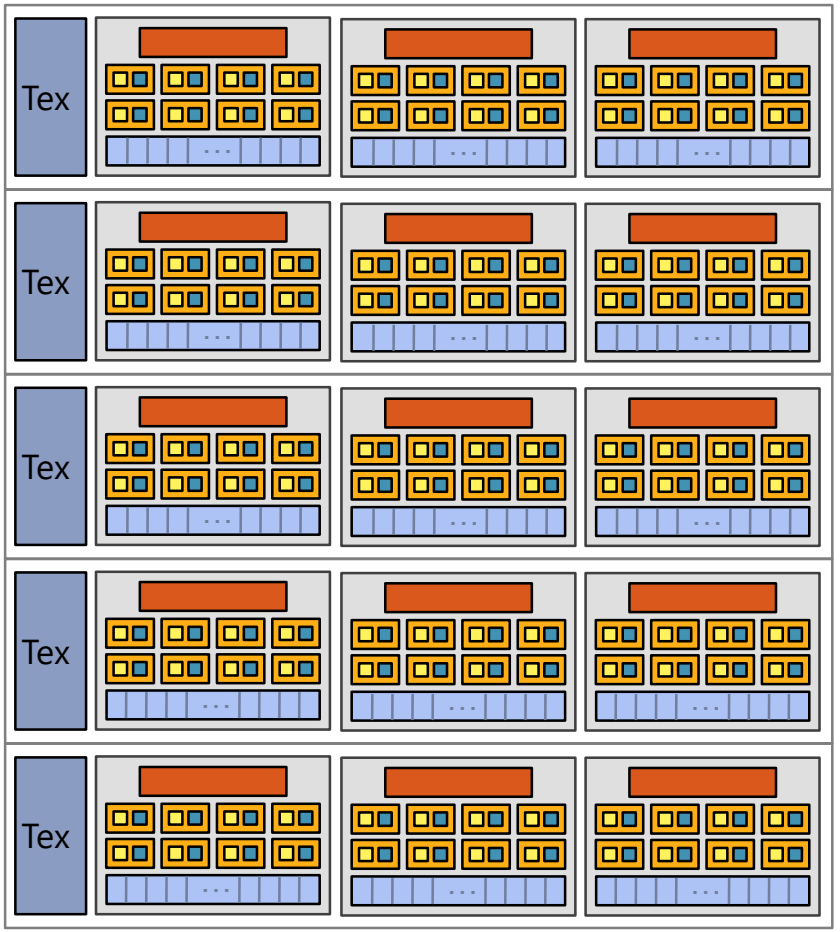


- Groups of 32 threads share instruction stream (each group is a Warp): they execute the same instruction on different data
- **Up to 32 warps are interleaved in an FGMT manner**
- Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285





30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian

End of Fine-Grained Multithreading

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)

Predicate Combining (*not* Predicated Execution)

- 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 instead of having one branch for each
  - 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

Predication (Predicated Execution)

- Idea: Convert control dependence to data dependence
- Simple example: Suppose we had a Conditional Move instruction…
  - CMOV condition, R1 ← R2
  - R1 = (condition == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)
- Code example with branches vs. CMOVs

  if (a == 5) {b = 4;} else {b = 3;}

  CMPEQ condition, a, 5;

  CMOV condition, b ← 4;

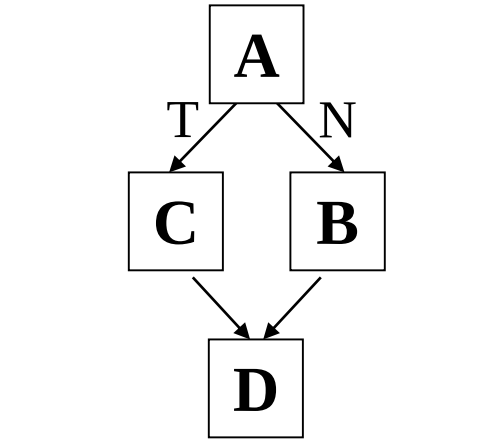
  CMOV !condition, b ← 3;

Predication (Predicated Execution)

- Idea: Compiler converts control dependence into data dependence → branch is eliminated
  - Each instruction has a predicate bit set based on the predicate computation
  - Only instructions with TRUE predicates are committed (others turned into NOPs)

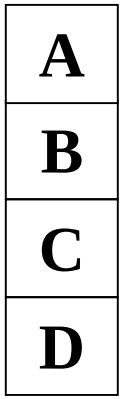
```
if (cond) {
    b = 0;
}
else {
    b = 1;
}
```

(normal branch code)




```
A   p1 = (cond)
    branch p1, TARGET
B   mov b, 1
    jmp JOIN
C   TARGET:
    mov b, 0
D   add  x, b, 1
```

(predicated code)




```
A   p1 = (cond)
B   (!p1) mov  b, 1
C   (p1) mov  b, 0
D   add  x, b, 1
```

Predicated Execution References

- Allen et al., "Conversion of control dependence to data dependence," POPL 1983.

- Kim et al., "Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution," MICRO 2005.

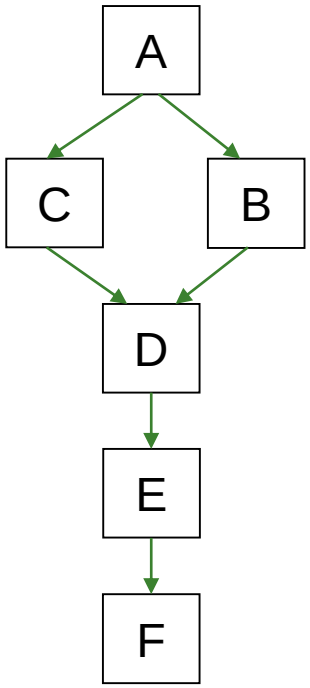
Conditional Move Operations

- Very limited form of predicated execution

- CMOV R1 ← R2
  - R1 = (ConditionCode == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)

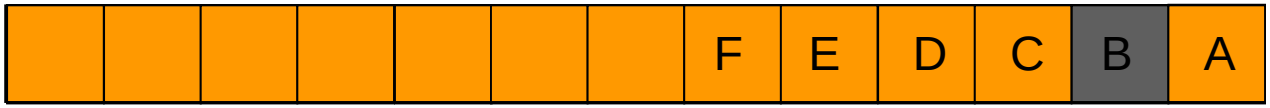
Predicated Execution (II)

- Predicated execution can be high performance and energy-efficient



Predicated Execution

Fetch Decode Rename Schedule RegisterRead Execute



nop

Branch Prediction

Fetch Decode Rename Schedule RegisterRead Execute

Pipeline flush!!

Predicated Execution

- Eliminates branches → enables straight line code (i.e., larger basic blocks in code)
- Advantages
  - Eliminates hard-to-predict branches
  - Always-not-taken prediction works better (no branches)
  - Compiler has more freedom to optimize code (no branches)
    - control flow does not hinder inst. reordering optimizations
    - code optimizations hindered only by data dependencies
- Disadvantages
  - Useless work: some instructions fetched/executed but discarded (especially bad for easy-to-predict branches)
  - Requires additional ISA support
  - Can we eliminate all branches this way?

Predicated Execution (III)

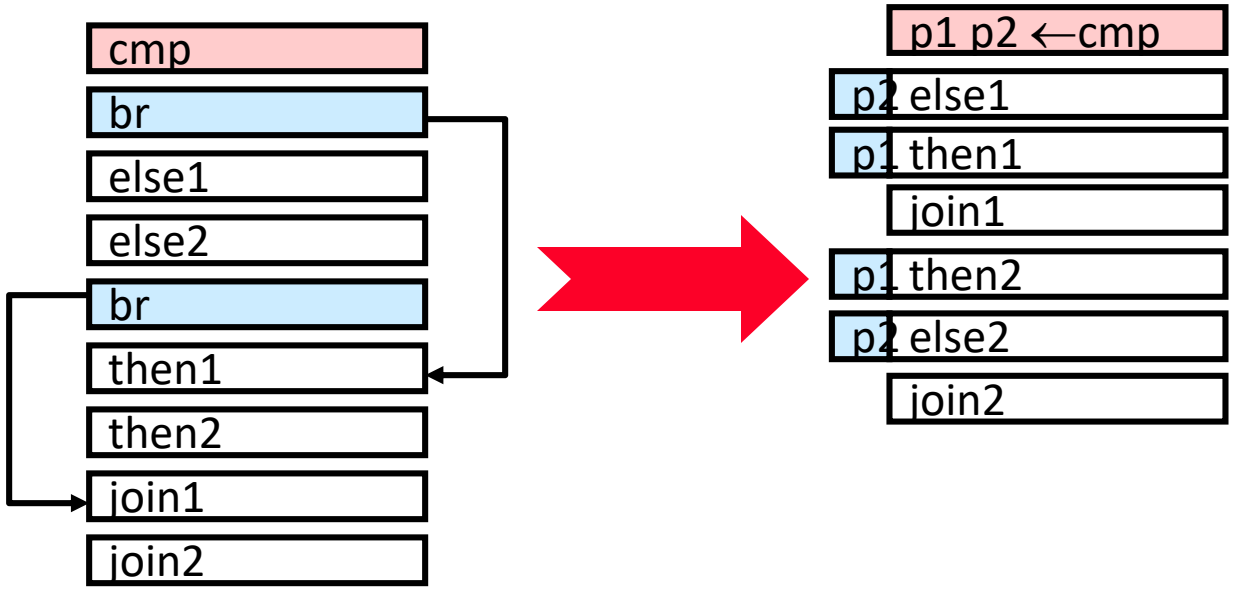
- Advantages:
  - + Eliminates mispredictions for hard-to-predict branches
    - + No need for branch prediction for some branches
    - + Good if misprediction cost > useless work due to predication
  - + Enables code optimizations hindered by the control dependency
    - + Can move instructions more freely within predicated code

- Disadvantages:
  - -- Causes useless work for branches that are easy to predict
    - -- Reduces performance if misprediction cost < useless work
    - -- Adaptivity: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, program phase, control-flow path.
  - -- Additional hardware and ISA support
  - -- Cannot eliminate all hard to predict branches
    - -- Loop branches

Predicated Execution in Intel Itanium

- Each instruction can be separately predicated
- 64 one-bit predicate registers

  each instruction carries a 6-bit predicate field
- An instruction is effectively a NOP if its predicate is false

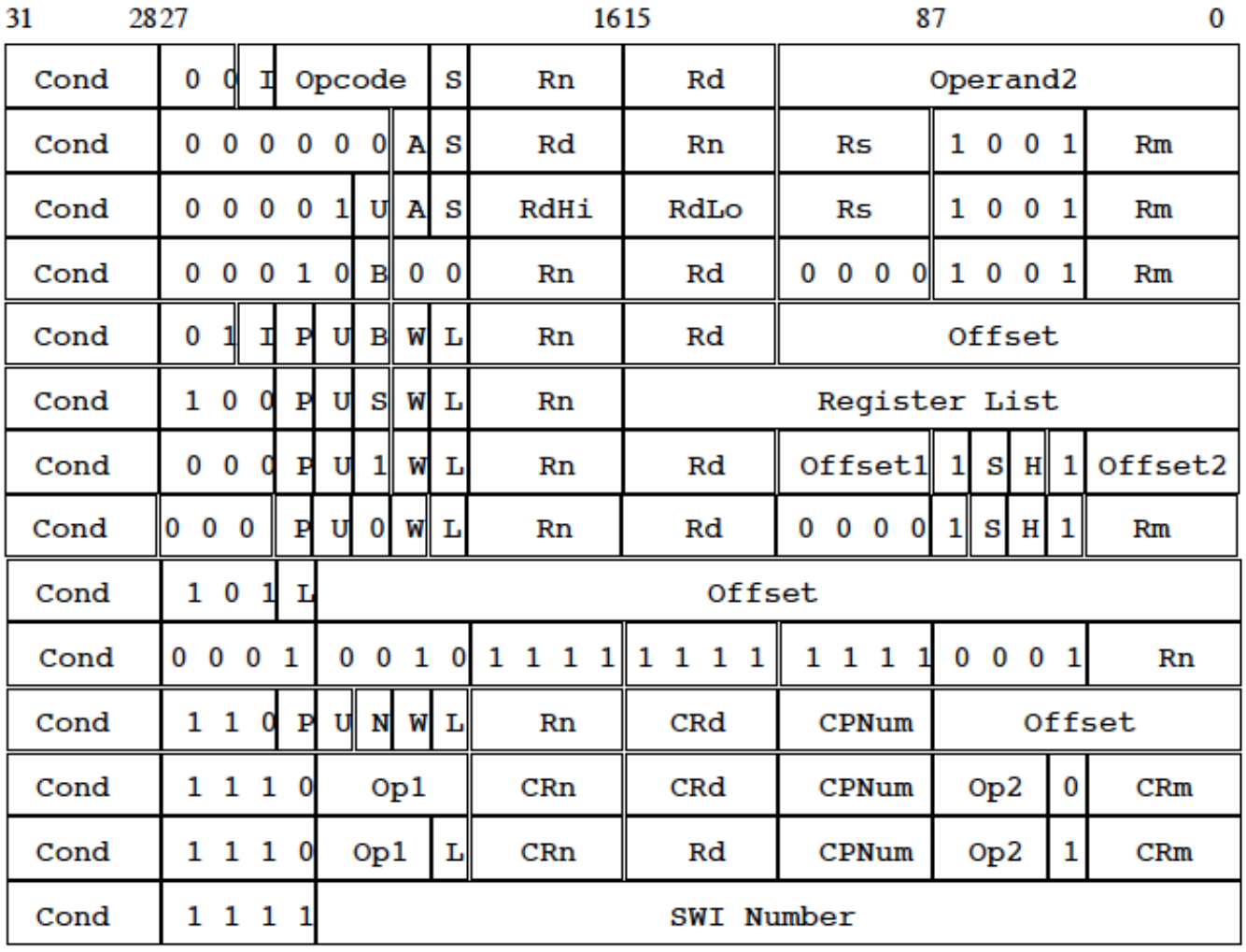


Conditional Execution in the ARM ISA

- Almost all ARM instructions can include an optional condition code.
  - Prior to ARM v8

- An instruction with a condition code is executed only if the condition code flags in the CPSR meet the specified condition.

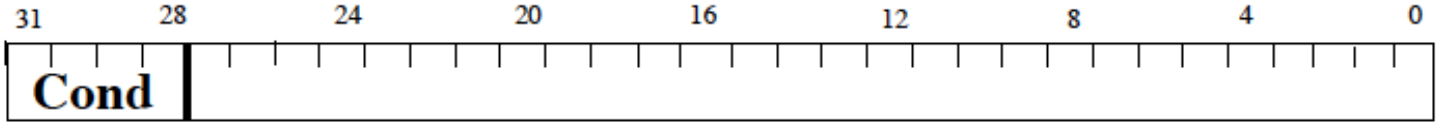
Conditional Execution in ARM ISA



Instruction type

- Data processing / PSR Transfer
- Multiply
- Long Multiply **(v3M / v4 only)**
- Swap
- Load/Store Byte/Word
- Load/Store Multiple
- Halfword transfer : Immediate offset **(v4 only)**
- Halfword transfer: Register offset **(v4 only)**
- Branch
- Branch Exchange **(v4T only)**
- Coprocessor data transfer
- Coprocessor data operation
- Coprocessor register transfer
- Software interrupt

Conditional Execution in ARM ISA



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

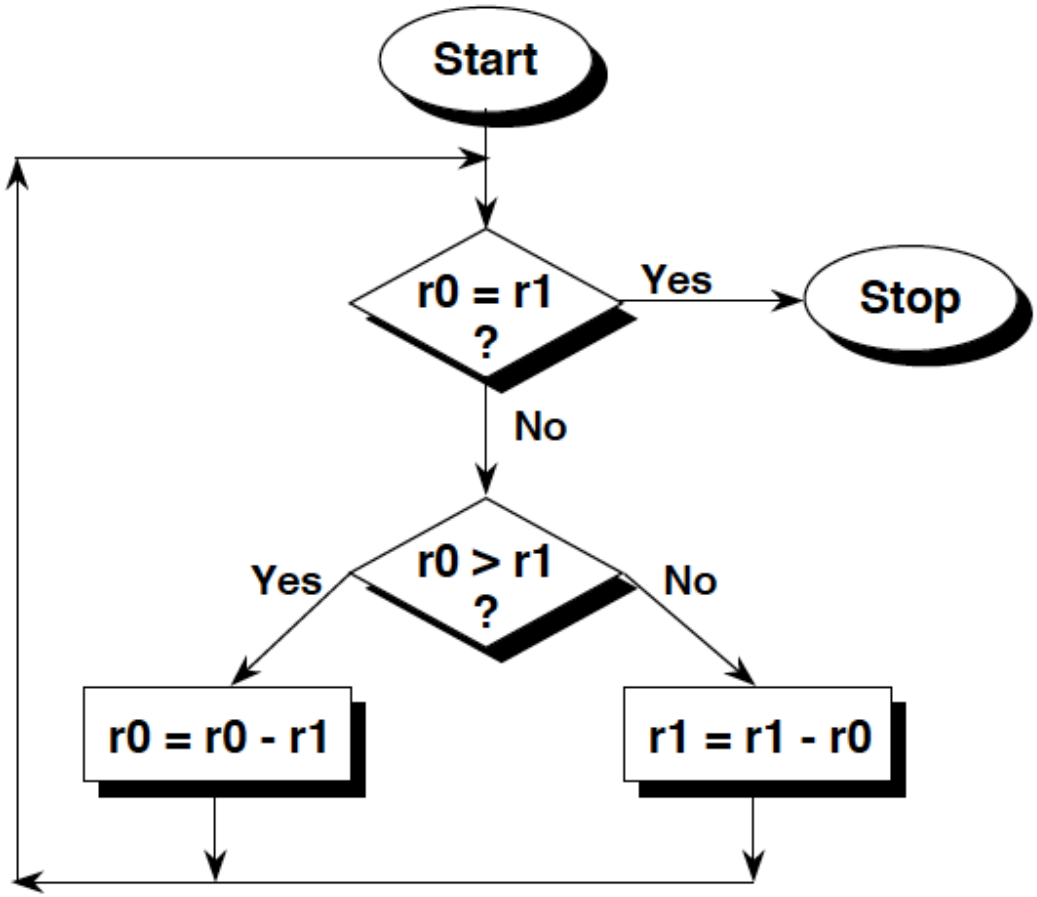
1110 = AL - always

1111 = NV - reserved.

Conditional Execution in ARM ISA

* **To execute an instruction conditionally, simply postfix it with the appropriate condition:**
  - For example an add instruction takes the form:
    - `ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)`
  - To execute this only if the zero flag is set:
    - `ADDEQ r0,r1,r2 ; If zero flag set then...`
      `; ... r0 = r1 + r2`
* **By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an "S".**
  - For example to add two numbers and set the condition flags:
    - `ADDS r0,r1,r2 ; r0 = r1 + r2`
      `; ... and set flags`

Conditional Execution in ARM ISA



* **Convert the GCD algorithm given in this flowchart into**
  1) "Normal" assembler, where only branches can be conditional.
  2) ARM assembler, where all instructions are conditional, thus improving code density.

* **The only instructions you need are CMP, B and SUB.**

Conditional Execution in ARM ISA

"Normal" Assembler

```
gcd     cmp r0, r1        ;reached the end?
        beq stop
        blt less          ;if r0 > r1
        sub r0, r0, r1    ;subtract r1 from r0
        bal gcd
less    sub r1, r1, r0    ;subtract r0 from r1
        bal gcd
stop
```

ARM Conditional Assembler

```
gcd     cmp   r0, r1      ;if r0 > r1
        subgt r0, r0, r1  ;subtract r1 from r0
        sublt r1, r1, r0  ;else subtract r0 from r1
        bne   gcd         ;reached the end?
```

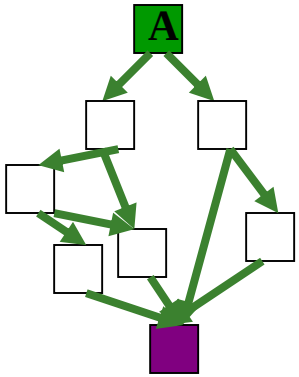
Idealism

- Wouldn't it be nice
  - If the branch is eliminated (predicated) only when it would actually be mispredicted
  - If the branch were predicted when it would actually be correctly predicted

- Wouldn't it be nice
  - If predication did not require ISA support

Improving Predicated Execution

- Three major limitations of predication
  1. Adaptivity: non-adaptive to branch behavior
  2. Complex CFG: inapplicable to loops/complex control flow graphs
  3. ISA: Requires large ISA changes



- Wish Branches [Kim+, MICRO 2005]
  - Solve 1 and partially 2 (for loops)

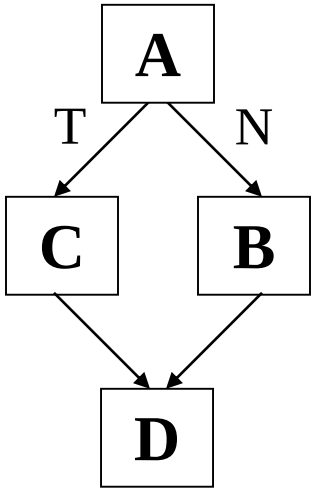
- Dynamic Predicated Execution
  - Diverge-Merge Processor [Kim+, MICRO 2006]
    - Solves 1, 2 (partially), 3

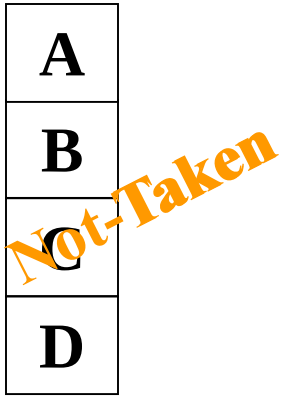
Wish Branches

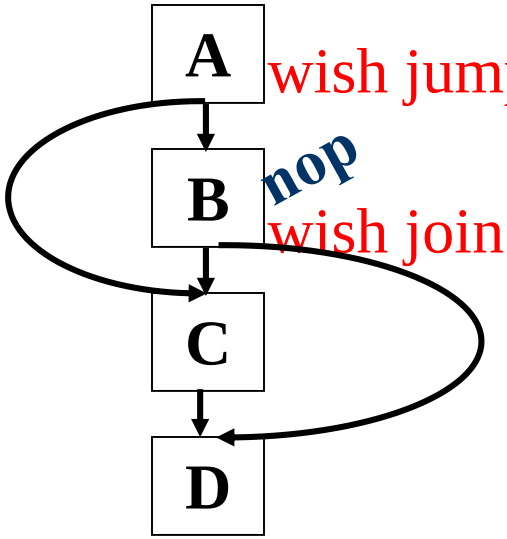
- The **compiler** generates code (with wish branches) that can be executed **either** as predicated code **or** non-predicated code (normal branch code)
- The **hardware decides** to execute predicated code or normal branch code at run-time based on the confidence of branch prediction
- **Easy to predict: normal branch code**
- **Hard to predict: predicated code**

- Kim et al., "Wish Branches: Enabling Adaptive and Aggressive Predicated Execution," MICRO 2006, IEEE Micro Top Picks, Jan/Feb 2006.

Wish Jump/Join








```
A   p1 = (cond)
    branch  p1, TARGET
B   mov b, 1
    jmp JOIN
C   TARGET:
    mov b,0
```

normal branch code

```
A   p1 = (cond)
B   (!p1) mov  b,1
C   (p1) mov  b,0
```

predicated code

```
A   p1=(cond)
    wish.jump p1 TARGET
B   (!p1) mov b,1
    wish.join !p1 JOIN
C   TARGET:
    (p1) mov b,0
D   JOIN:
```

nop

wish jump/join code

Wish Branches vs. Predicated Execution

- Advantages compared to predicated execution
  - **Reduces the overhead** of predication
  - Increases the benefits of predicated code by allowing the compiler to generate more **aggressively-predicated code**
  - Makes predicated code less dependent on machine configuration (e.g. branch predictor)

- Disadvantages compared to predicated execution
  - Extra branch instructions use machine resources
  - Extra branch instructions increase the contention for branch predictor table entries
  - Constrains the compiler's scope for **code optimizations**

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)

Multi-Path Execution

- Idea: Execute both paths after a conditional branch
  - For all branches: Riseman and Foster, "The inhibition of potential parallelism by conditional jumps," IEEE Transactions on Computers, 1972.
  - For a hard-to-predict branch: Use dynamic confidence estimation

- Advantages:

  + Improves performance if misprediction cost > useless work

  + No ISA change needed

- Disadvantages:

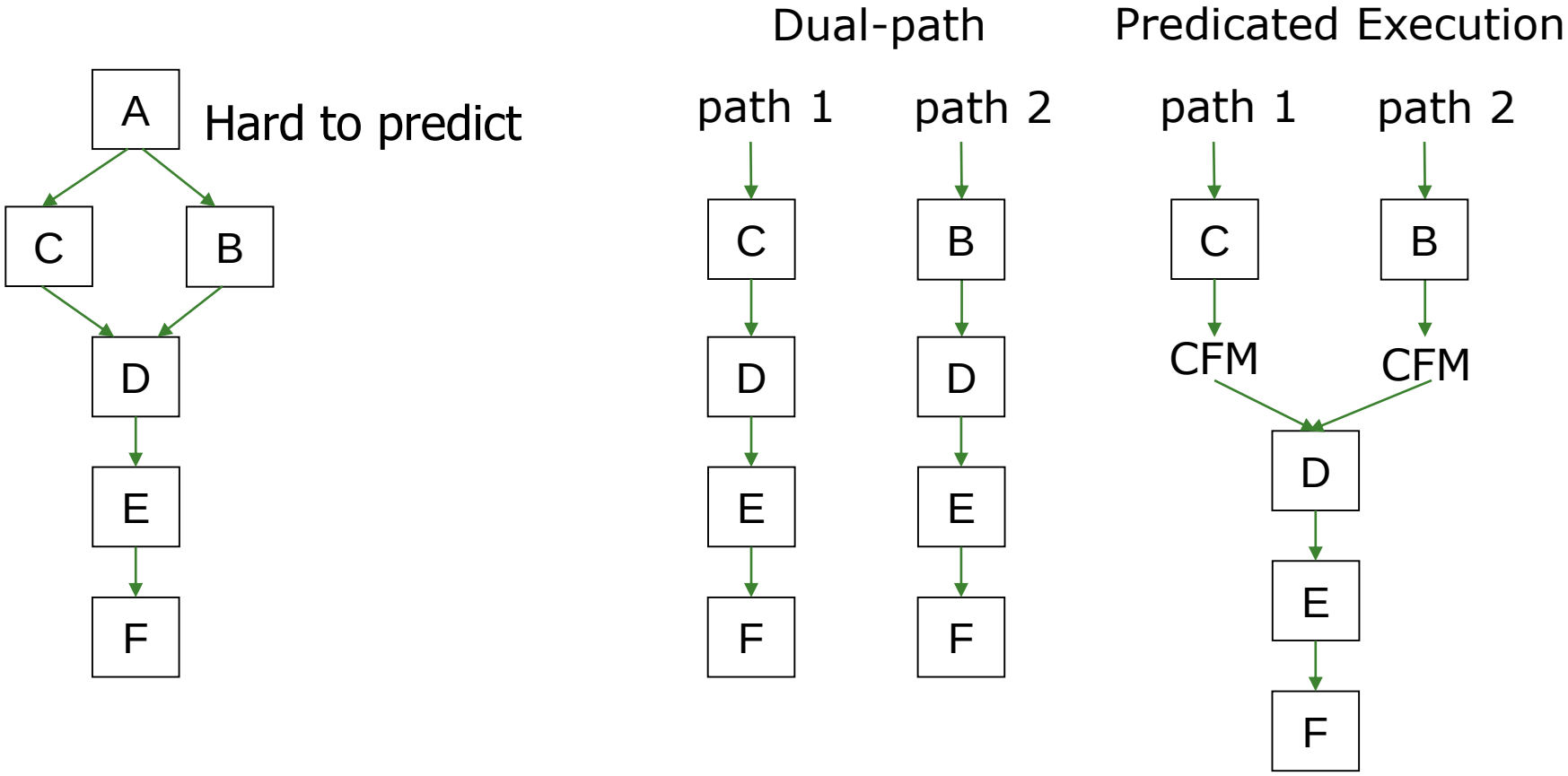
  -- What happens when the machine encounters another hard-to-predict branch? Execute both paths again?

  -- Paths followed quickly become exponential

  -- Each followed path requires its own context (registers, PC, GHR)

  -- Wasted work (and reduced performance) if paths merge

Dual-Path Execution versus Predication



Handling Other Types of Branches

Remember: 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)

How can we predict an indirect branch with many target addresses?

Call and Return Prediction

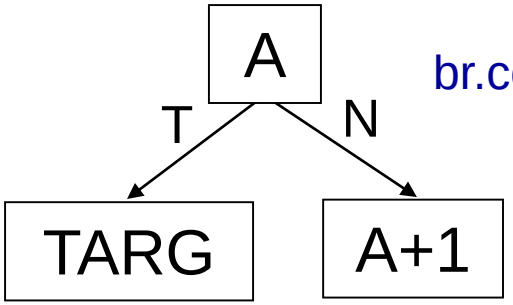
- Direct calls are easy to predict
  - Always taken, single target
  - Call marked in BTB, target predicted by BTB

```
Call X
        …
        Call X
                …
                Call X
                …
                Return
        Return
Return
```

- Returns are indirect branches
  - A function can be called from many points in code
  - A return instruction can have many target addresses
    - Next instruction after each call point for the same function
  - Observation: Usually a return matches a call
  - Idea: Use a stack to predict return addresses (Return Address Stack)
    - A fetched call: pushes the return (next instruction) address on the stack
    - A fetched return: pops the stack and uses the address as its predicted target
    - Accurate most of the time: 8-entry stack → > 95% accuracy

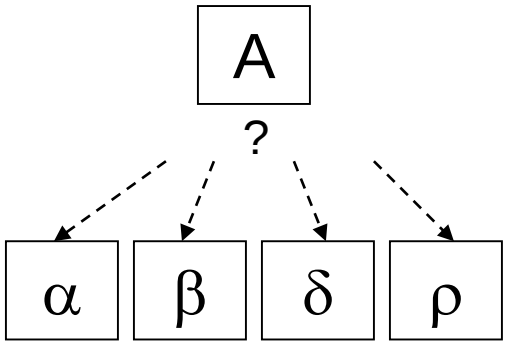
Indirect Branch Prediction (I)

- Register-indirect branches have multiple targets



br.cond TARGET

Conditional (Direct) Branch



R1 = MEM[R2]
branch R1

Indirect Jump

- Used to implement
  - Switch-case statements
  - Virtual function calls
  - Jump tables (of function pointers)
  - Interface calls

Indirect Branch Prediction (II)

- No direction prediction needed
- Idea 1: Predict the last resolved target as the next fetch address

  + Simple: Use the BTB to store the target address

  -- Inaccurate: 50% accuracy (empirical). Many indirect branches switch between different targets

- Idea 2: Use history based target prediction
  - E.g., Index the BTB with GHR XORed with Indirect Branch PC
  - Chang et al., "Target Prediction for Indirect Jumps," ISCA 1997.

  + More accurate

  -- An indirect branch maps to (too) many entries in BTB

  -- Conflict misses with other branches (direct or indirect)

  -- Inefficient use of space if branch has few target addresses

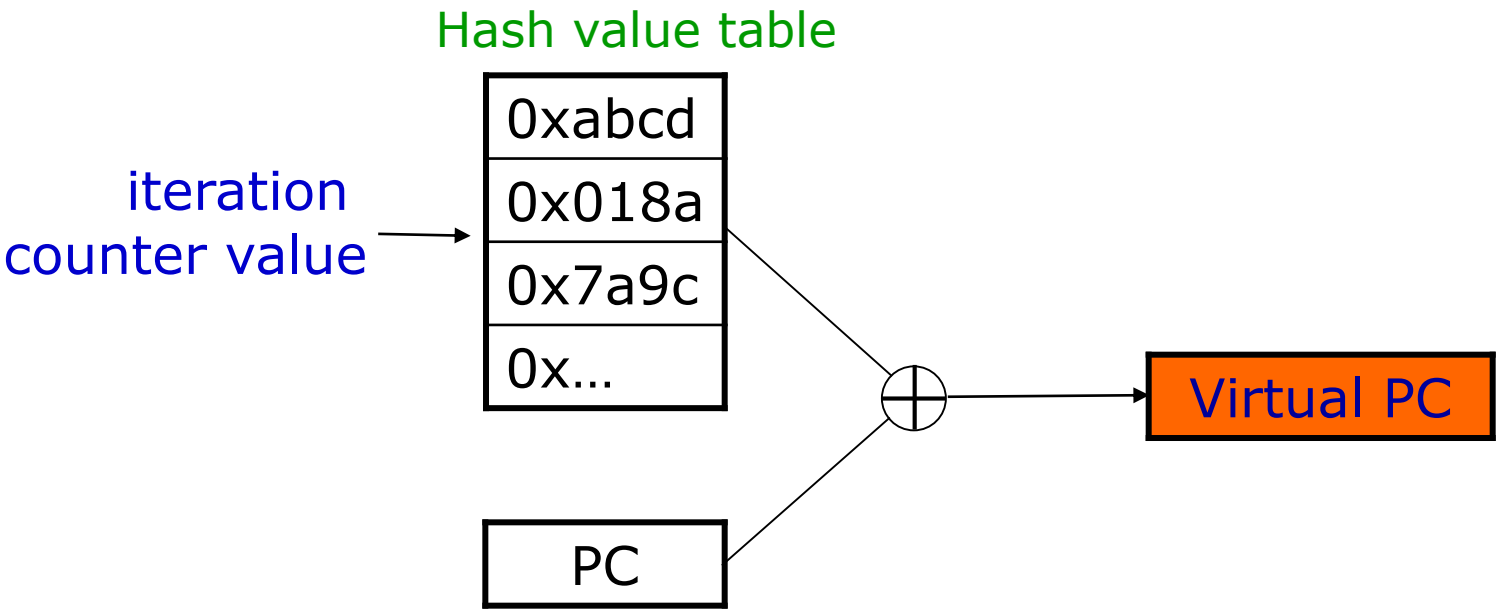
More Ideas on Indirect Branches?

- Virtual Program Counter prediction
  - Idea: Use conditional branch prediction structures ***iteratively*** to make an indirect branch prediction
  - i.e., *devirtualize* the indirect branch in hardware

- Curious?
  - Kim et al., "VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization," ISCA 2007.

Indirect Branch Prediction (III)

- Idea 3: Treat an indirect branch as "multiple virtual conditional branches" in hardware
  - Only for prediction purposes
  - Predict each "virtual conditional branch" iteratively
  - Kim et al., "VPC prediction," ISCA 2007.



VPC Prediction (I)

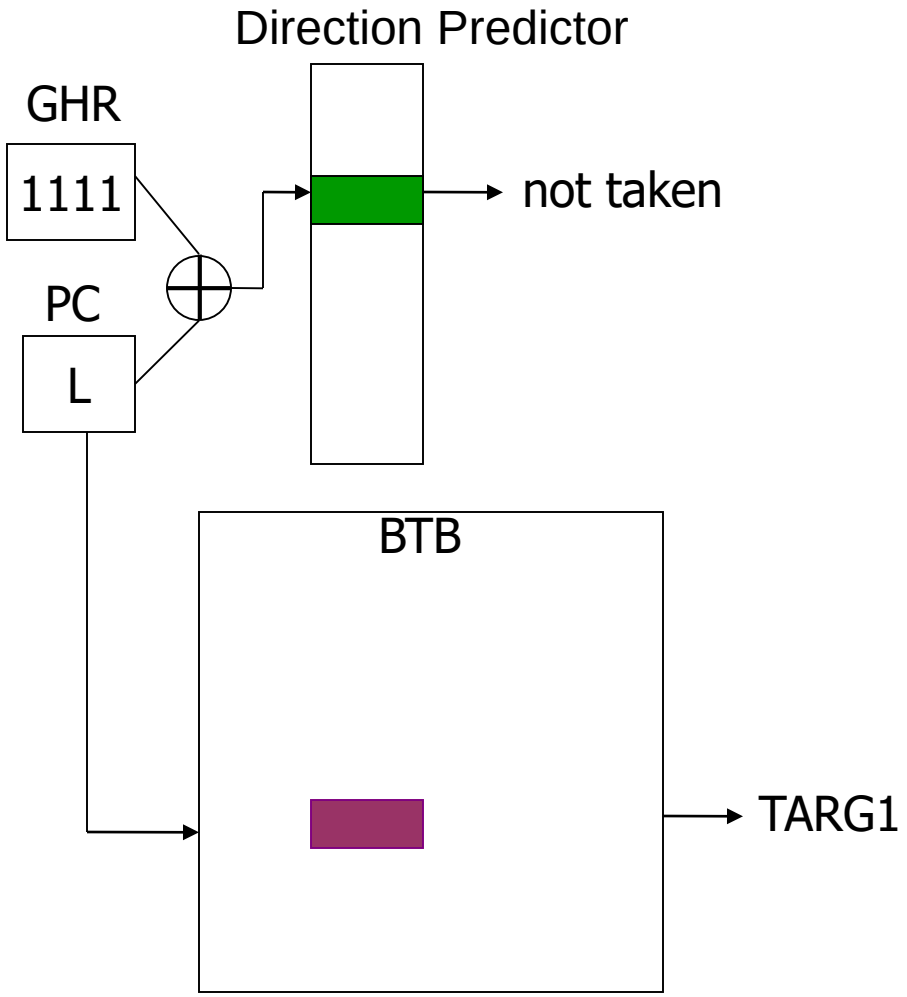
Real Instruction

call R1 // PC: L

Virtual Instructions

cond. jump TARG1 // VPC: L
cond. jump TARG2 // VPC: VL2
cond. jump TARG3 // VPC: VL3
cond. jump TARG4 // VPC: VL4

Next iteration



VPC Prediction (II)

Real Instruction

call R1 // PC: L

Virtual Instructions

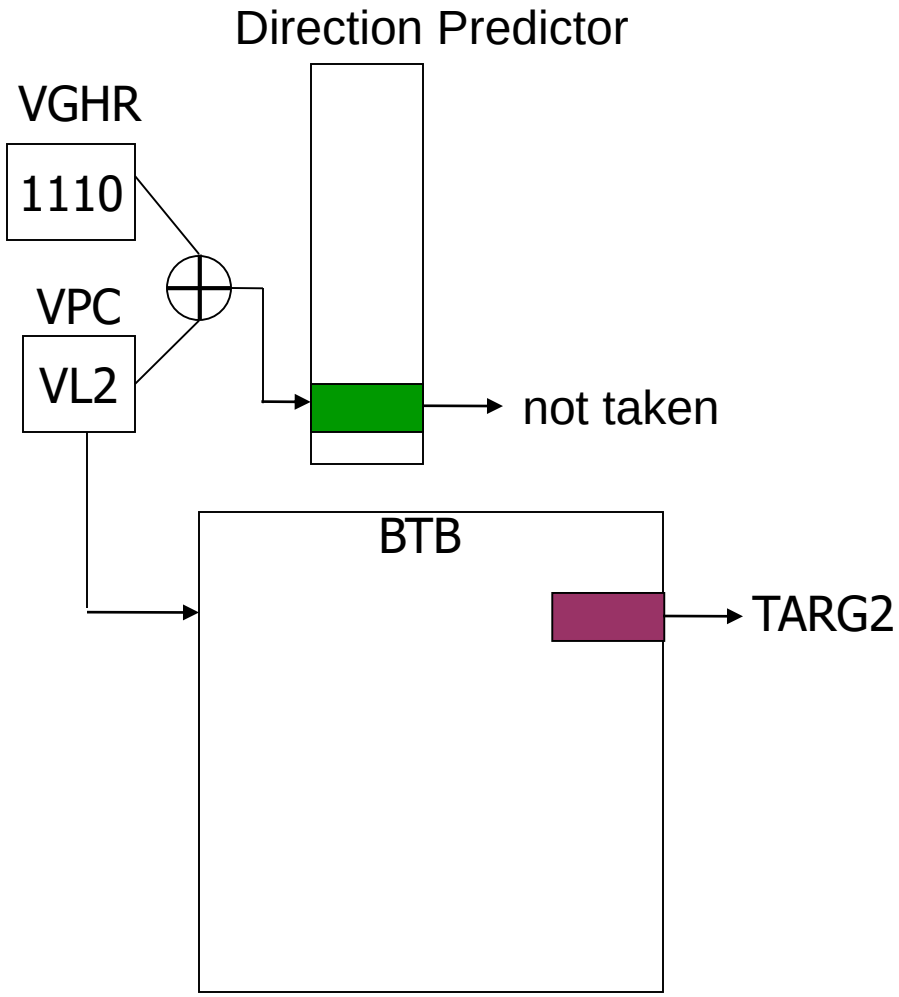
cond. jump TARG1 // VPC: L

cond. jump TARG2 // VPC: VL2

cond. jump TARG3 // VPC: VL3

cond. jump TARG4 // VPC: VL4

Next iteration



VPC Prediction (III)

Real Instruction

call R1 // PC: L

Virtual Instructions

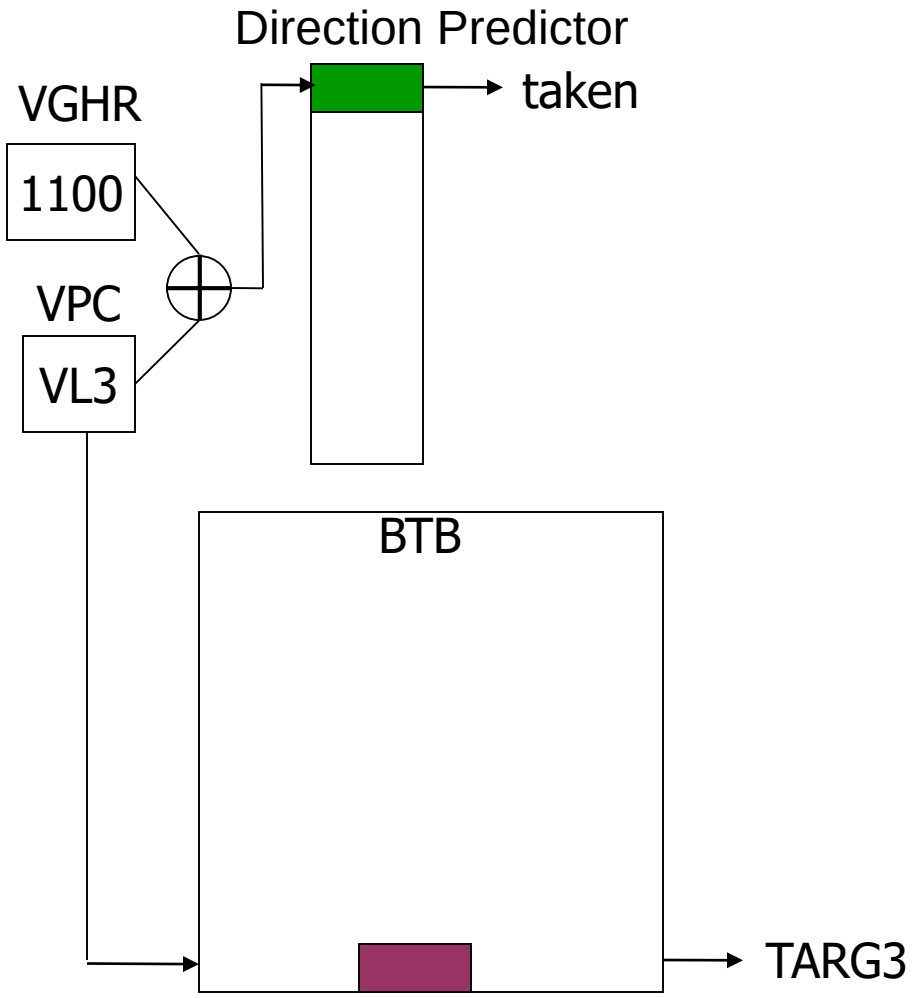
cond. jump TARG1 // VPC: L

cond. jump TARG2 // VPC: VL2

cond. jump TARG3 // VPC: VL3

cond. jump TARG4 // VPC: VL4

Predicted Target = TARG3



VPC Prediction (IV)

- Advantages:
  + High prediction accuracy (>90%)
  + No separate indirect branch predictor
  + Resource efficient (reuses existing components)
  + Improvement in conditional branch prediction algorithms also improves indirect branch prediction
  + Number of locations in BTB consumed for a branch = number of target addresses seen

- Disadvantages:
  -- Takes multiple cycles (sometimes) to predict the target address
  -- More interference in direction predictor and BTB

Issues in Branch Prediction (I)

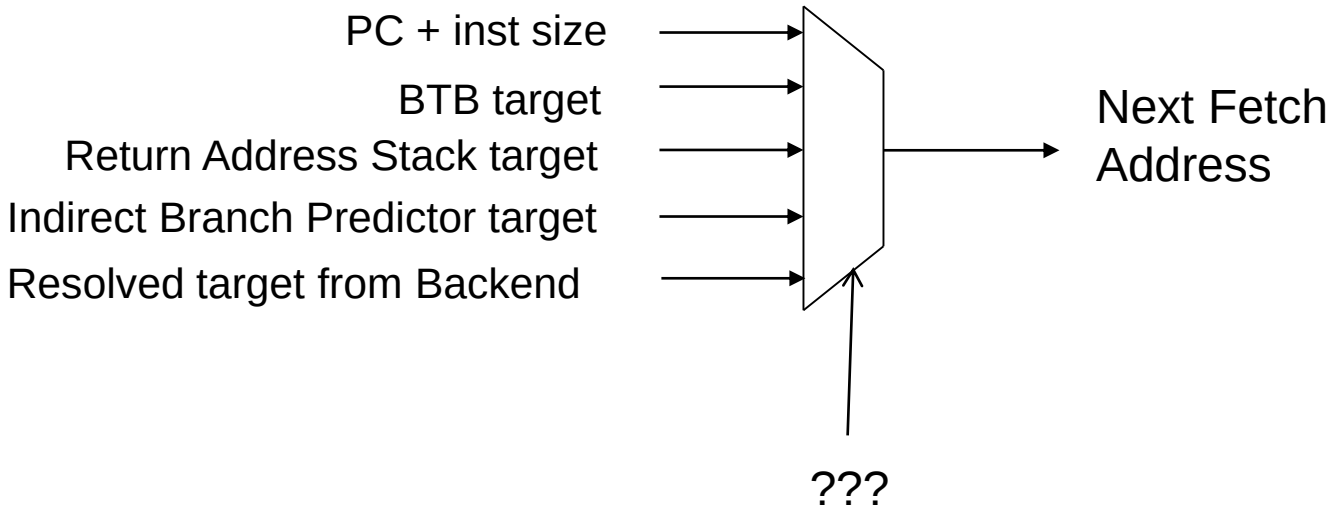
- Need to identify a branch before it is fetched

- How do we do this?
  - BTB hit → indicates that the fetched instruction is a branch
  - BTB entry contains the "type" of the branch
  - Pre-decoded "branch type" information stored in the instruction cache identifies type of branch

- What if no BTB?
  - Bubble in the pipeline until target address is computed
  - E.g., IBM POWER4

Issues in Branch Prediction (II)

- Latency: Prediction is latency critical
  - Need to generate next fetch address for the next cycle
  - Bigger, more complex predictors are more accurate but slower



Complications in Superscalar Processors

- Superscalar processors
  - attempt to execute more than 1 instruction-per-cycle
  - must fetch multiple instructions per cycle
- What if there is a branch in the middle of fetched instructions?

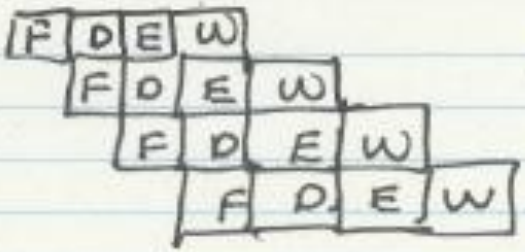
- Consider a 2-way superscalar fetch scenario

  (case 1) Both insts are not taken control flow inst
  - nPC = PC + 8

  (case 2) One of the insts is a taken control flow inst
  - nPC = predicted target addr
  - *NOTE* both instructions could be control-flow; prediction based on the first one predicted taken
  - If the 1st instruction is the predicted taken branch

    → nullify 2nd instruction fetched

Multiple Instruction Fetch: Concepts



← Fetch 1 inst/cycle

- Downside:
  Flynn's bottleneck
  If you fetch 1 inst/cycle you cannot fetch >1 inst/cycle

← Fetch 4 inst/cycle

Two major approaches

1) VLIW
   Compiler decides what insts. can be executed in parallel
   → Simple hardware

2) Superscalar
   Hardware detects dependences between instructions that are fetched in the same cycle.

Doing Better than Stalling Fetch …

- Rather than waiting for true-dependence on PC to resolve, just guess nextPC = PC+4 to keep fetching every cycle

  Is this a good guess?

  What do you lose if you guessed incorrectly?

- ~20% of the instruction mix is control flow
  - ~50 % of "forward" control flow (i.e., if-then-else) is taken
  - ~90% of "backward" control flow (i.e., loop back) is taken

  Overall, typically ~70% taken and ~30% not taken
  [Lee and Smith, 1984]

- Expect "nextPC = PC+4" ~86% of the time, but what about the remaining 14%?

Conditional Execution in ARM (Prior to v8)

- Same as predicated execution

- Every instruction is conditionally executed
  - in ARM ISAs prior to v8

Trace Cache Design Issues (I)

- Granularity of prediction: Trace based versus branch based?
  - + Trace based eliminates the need for multiple predictions/cycle
  - -- Trace based can be less accurate
  - -- Trace based: How do you distinguish traces with the same start address?

- When to form traces: Based on fetched or retired blocks?
  - + Retired: Likely to be more accurate
  - -- Retired: Formation of trace is delayed until blocks are committed
    - -- Very tight loops with short trip count might not benefit

- When to terminate the formation of a trace
  - After N instructions, after B branches, at an indirect jump or return

Trace Cache Design Issues (II)

- Should entire "path" match for a trace cache hit?
- Partial matching: A piece of a trace is supplied based on branch prediction

+ Increases hit rate when there is not a full path match

-- Lengthens critical path (next fetch address dependent on the match)

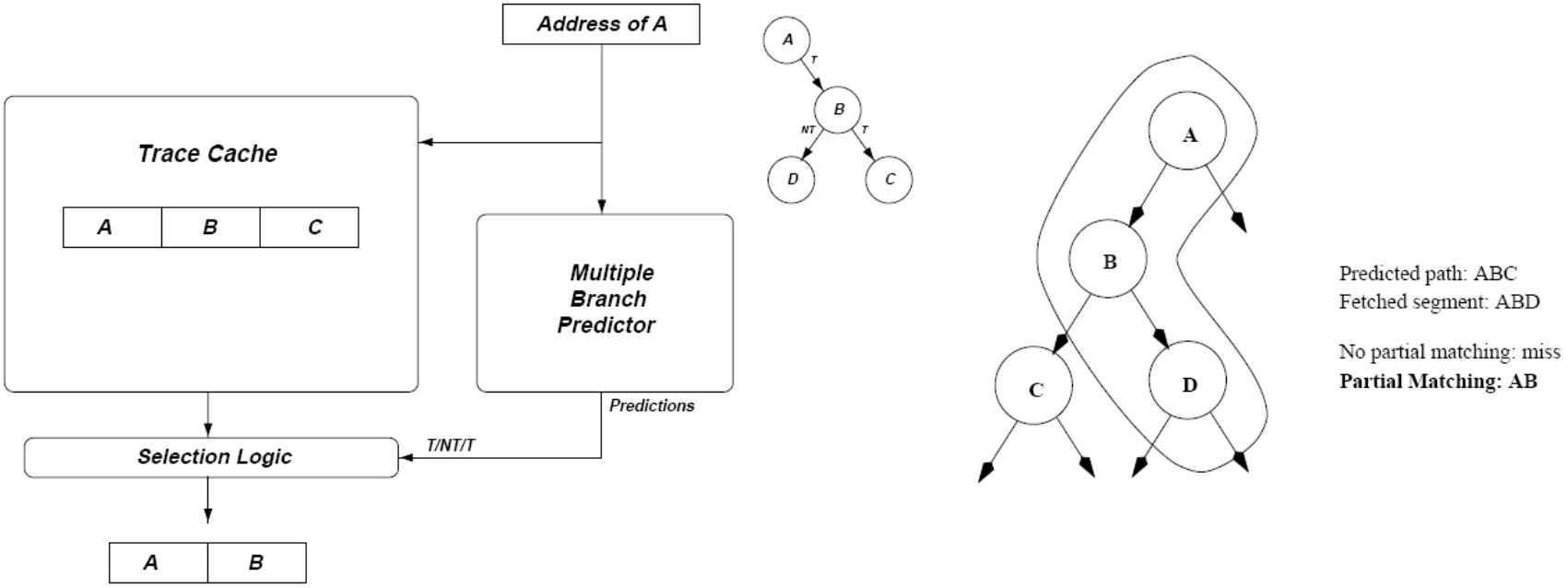


Figure 6.1: The trace cache and branch predictor are indexed with the address of block A. The inset figure shows the control flow from block A. The predictor selects the sequence ABD. The trace cache only contains ABC. AB is supplied.

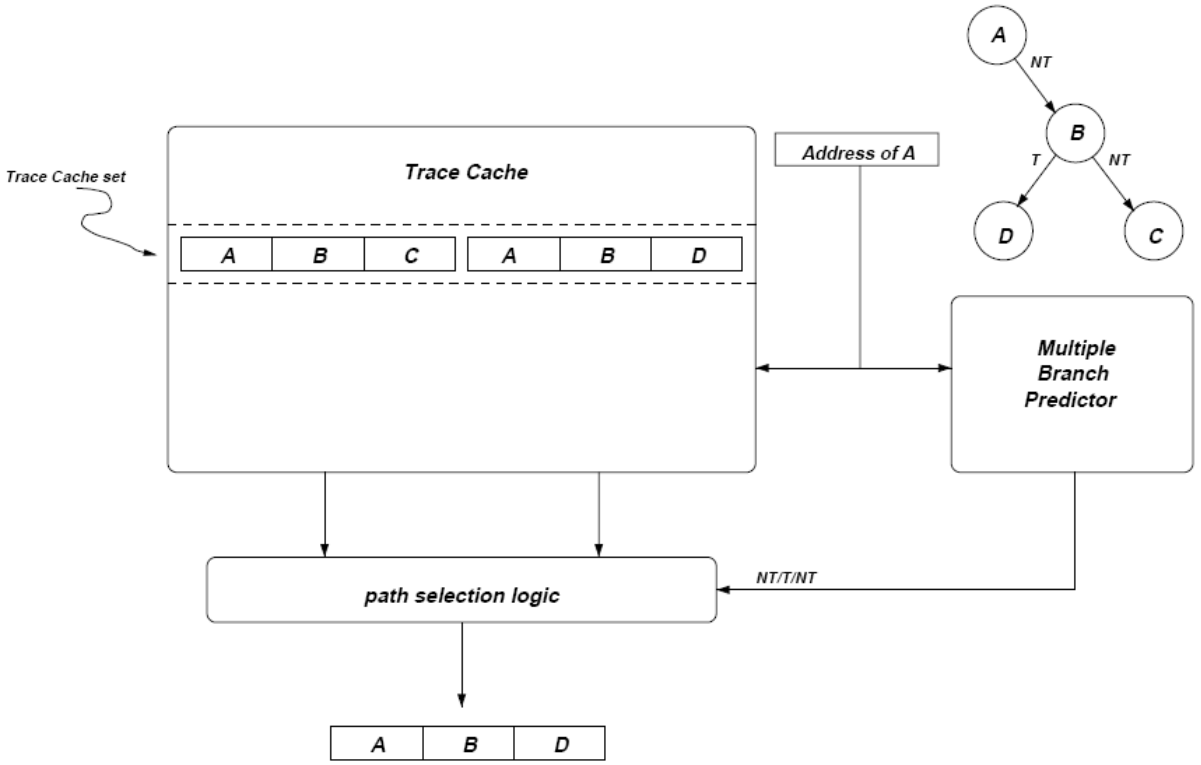
Trace Cache Design Issues (III)

- Path associativity: Multiple traces starting at the same address can be present in the cache at the same time.

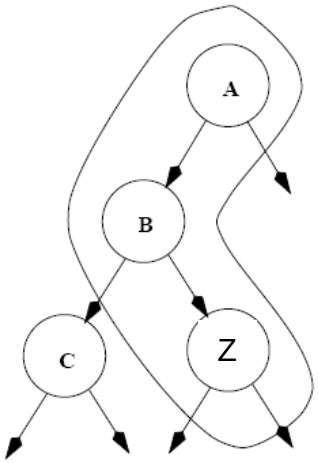
+ Good for traces with unbiased branches (e.g., ping pong between C and D)

-- Need to determine longest matching path

-- Increased cache pressure



Trace Cache Design Issues (IV)



Predicted path: ABC
Fetched segment: AB Z

No partial matching: miss
Partial matching: AB
Inactive Issue: AB (active) Z ıctive)

- Inactive issue: All blocks within a trace cache line are issued even if they do not match the predicted path

+ Reduces impact of branch mispredictions

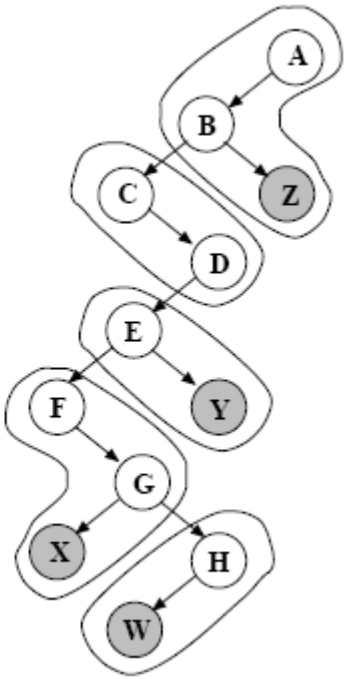
+ Reduces basic block duplication in trace cache

-- Slightly more complex scheduling/branch resolution

-- Some instructions not dispatched/flushed

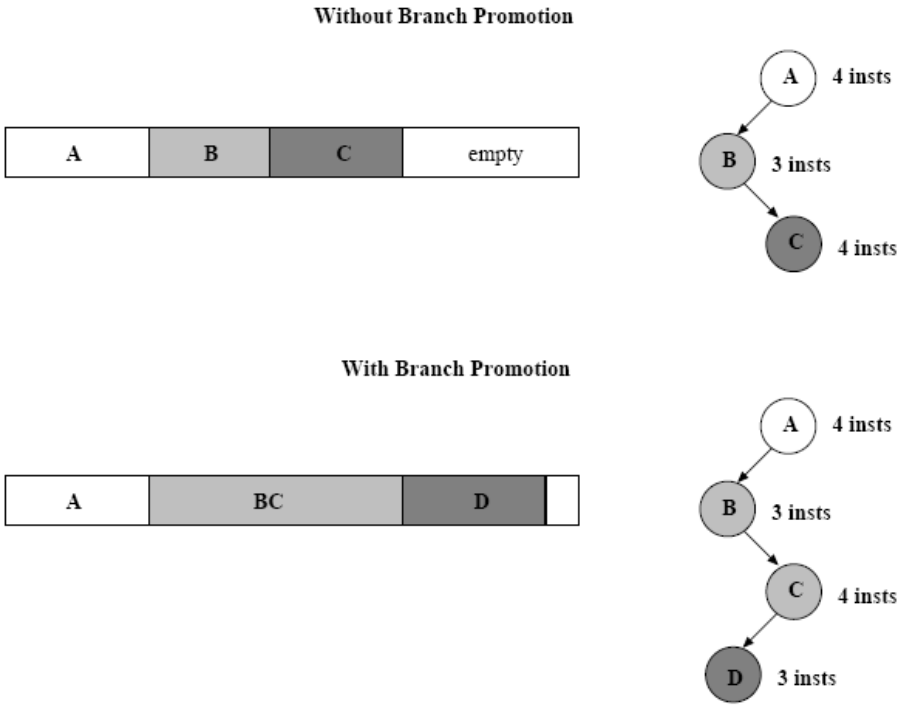
Instruction Window

H		W
F	G	X
E		Y
C		D
A	B	Z



Trace Cache Design Issues (V)

- Branch promotion: promote highly-biased branches to branches with static prediction
  - + Larger traces
  - + No need for consuming branch predictor BW
  - + Can enable optimizations within trace
  - -- Requires hardware to determine highly-biased branches



How to Determine Biased Branches

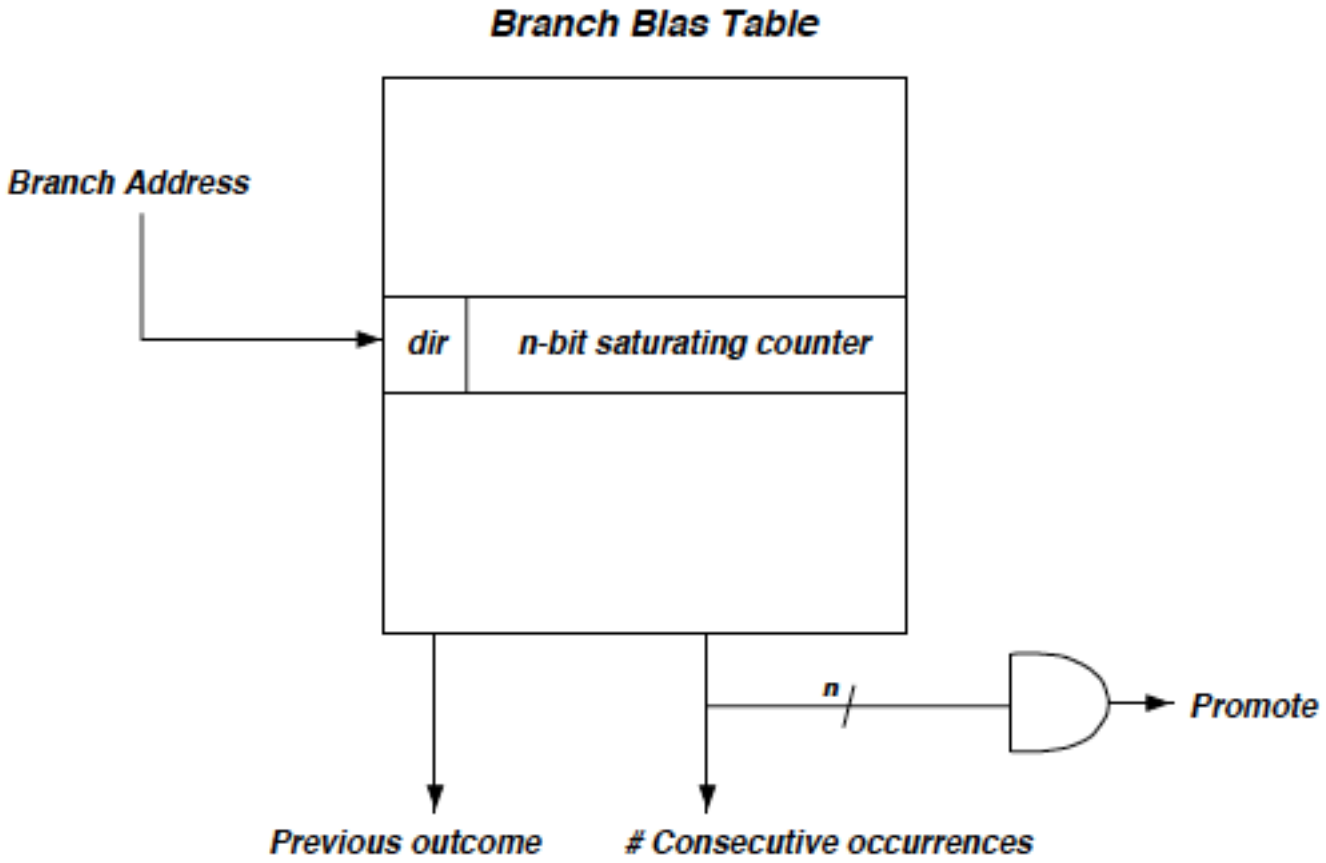
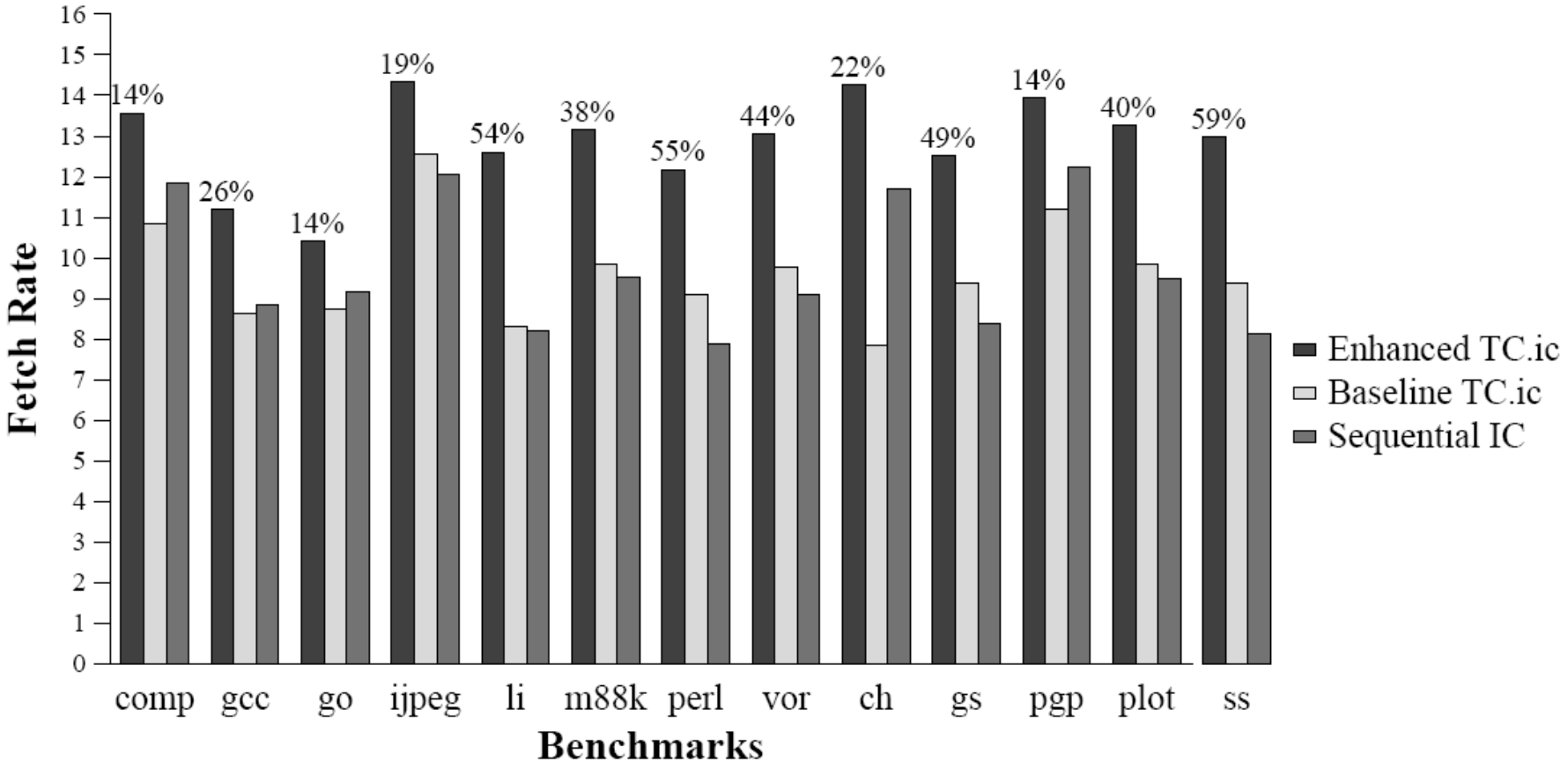
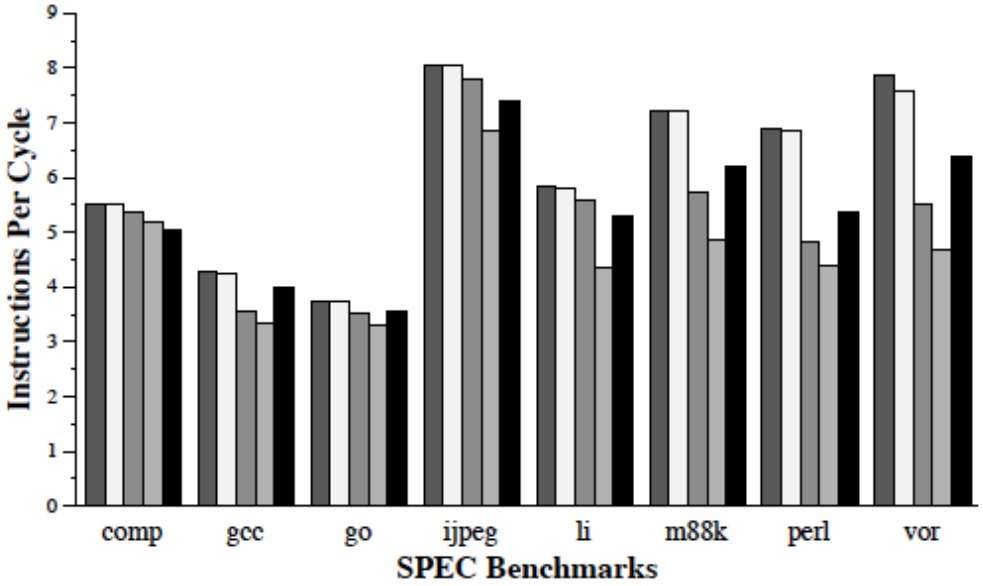


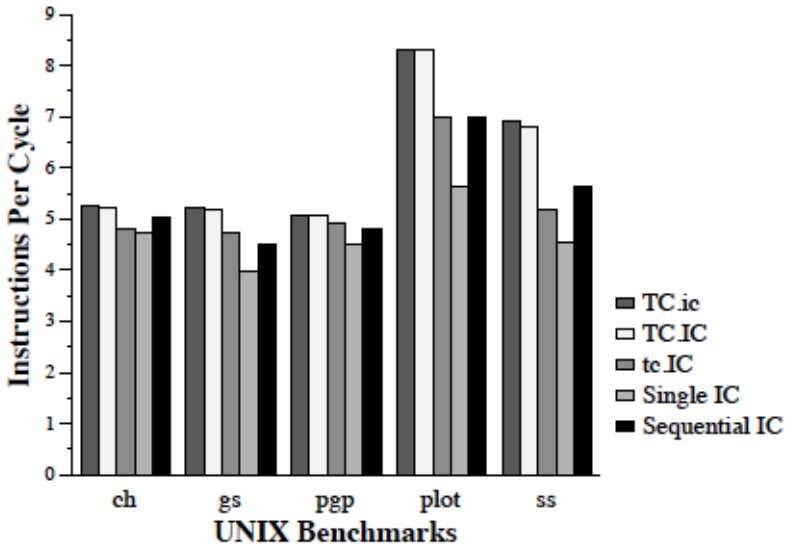
Figure 6.19: Diagram of the branch bias table.

Effect on Fetch Rate



Effect on IPC (16-wide superscalar)



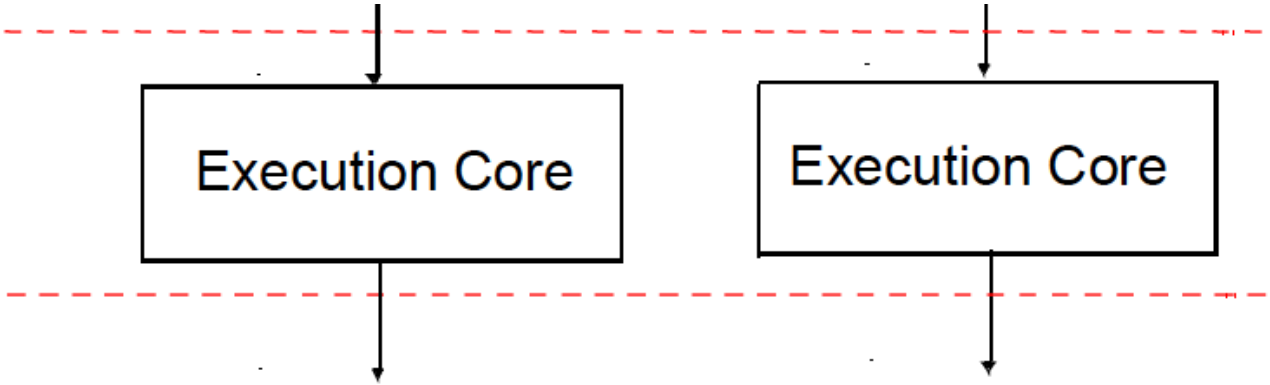


Configuration Name	TCache Size	ICache Size	Blocks per Fetch	Br Pred Type	BTB Size
TC.ic	128KB	4KB	3	Multiple	1KB
TC.IC	64KB	64KB	3	Multiple	8KB
tc.IC	4KB	128KB	3	Multiple	16KB
Single	–	128KB	1	Hybrid	20KB
Sequential	–	128KB	3	Multiple	16KB

- ~15% IPC increase over “sequential I-cache” that breaks fetch on a predicted-taken branch

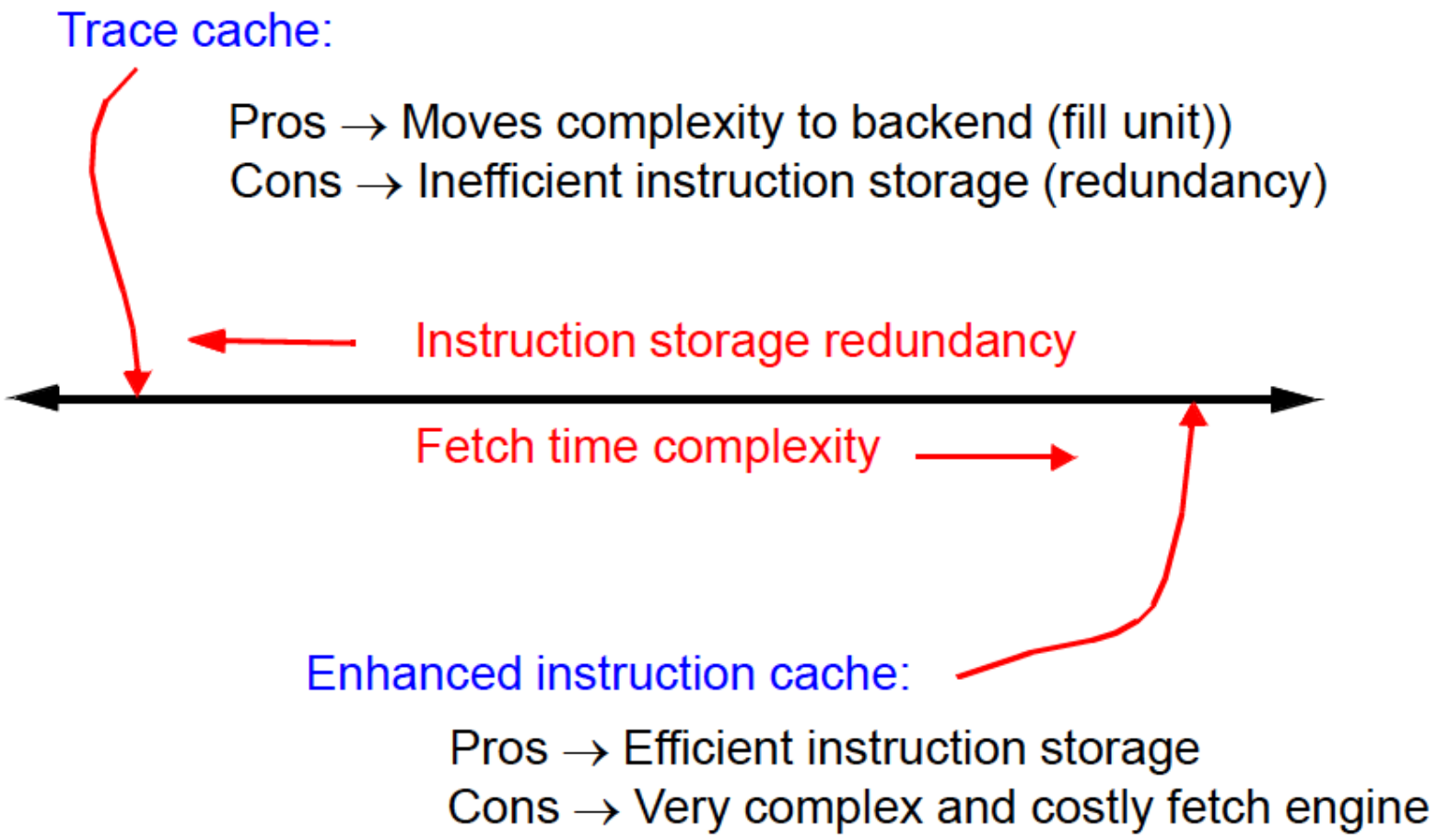
Enhanced I-Cache vs. Trace Cache (I)

	Enhanced Instruction Cache	Trace Cache
Fetch	1. Multiple-branch prediction 2. Instruction cache fetch from multiple blocks (N ports) 3. Instruction alignment & collapsing	1. Next trace prediction 2. Trace cache fetch



Completion	1. Multiple-branch predictor update	1. Trace construction and fill 2. Trace predictor update

Enhanced I-Cache vs. Trace Cache (II)



Frontend vs. Backend Complexity

- Backend is not on the critical path of instruction execution
  - Easier to increase its latency without affecting performance

- Frontend is on the critical path
  - Increased latency fetch directly increases
    - Branch misprediction penalty
  - Increased complexity can affect cycle time

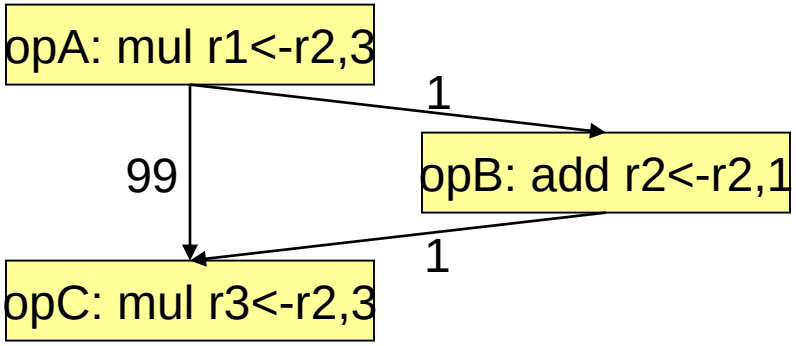
Fill Unit Optimizations

- Fill unit constructs traces out of decoded instructions
- Can perform optimizations across basic blocks
  - Branch promotion: promote highly-biased branches to branches with static prediction
  - Can treat the whole trace as an atomic execution unit
    - All or none of the trace is retired (based on branch directions in trace)
    - Enables many optimizations across blocks
  - Dead code elimination
  - Instruction reordering
  - Reassociation

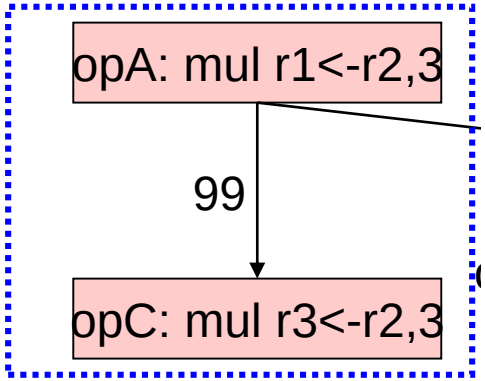
$$\begin{array}{l} ADDI\ Rx \leftarrow Ry + 4 \\ ADDI\ Rz \leftarrow Rx + 4 \end{array} \longrightarrow \begin{array}{l} ADDI\ Rx \leftarrow Ry + 4 \\ ADDI\ Rz \leftarrow Ry + 8 \end{array}$$

- Friendly et al., "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," MICRO 1998.

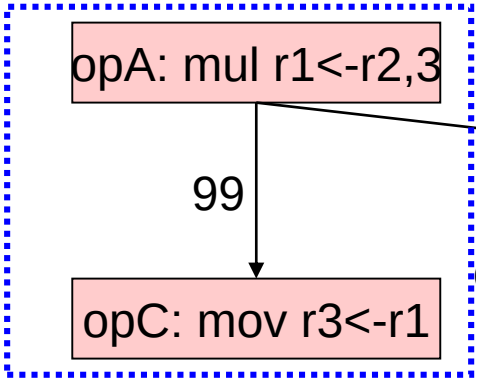
Remember This Optimization?



Original Code



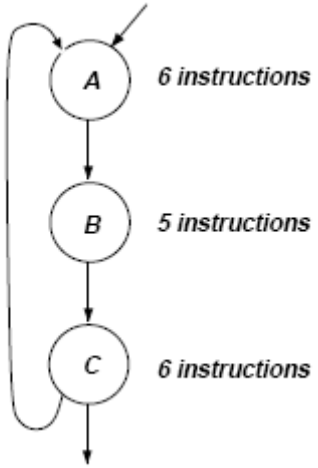
Part of Trace in Fill Unit



Optimized Trace

Redundancy in the Trace Cache

- ABC, BCA, CAB can all be in the trace cache
- Leads to contention and reduced hit rate



- One possible solution: Block based trace cache (Black et al., ISCA 1999)
- Idea: Decouple storage of basic blocks from their "names"
  - Store traces of pointers to basic blocks rather than traces of basic blocks themselves
  - Basic blocks stored in a separate "block table"

+ Reduces redundancy of basic blocks

-- Lengthens fetch cycle (indirection needed to access blocks)

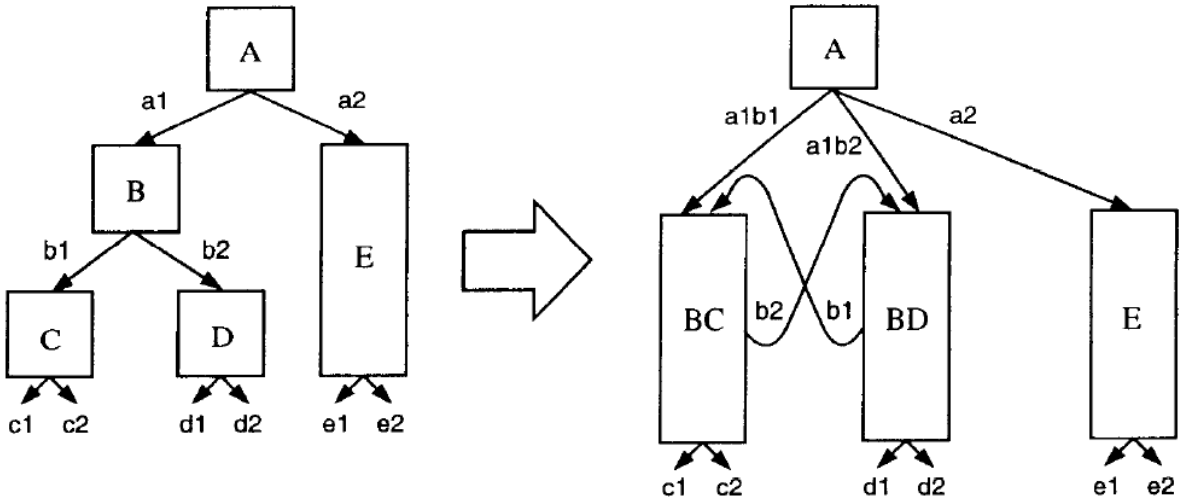
-- Block table needs to be multiported to obtain multiple blocks per cycle

Techniques to Reduce Fetch Breaks

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

Block Structured ISA

- Blocks (> instructions) are atomic (all-or-none) operations
  - Either all of the block is committed or none of it
- Compiler enlarges blocks by combining basic blocks with their control flow successors
  - Branches within the enlarged block converted to "fault" operations → if the fault operation evaluates to true, the block is discarded and the target of fault is fetched



Block Structured ISA (II)

- Advantages:
  + Larger blocks → larger units can be fetched from I-cache
  + Aggressive compiler optimizations (e.g. reordering) can be enabled within atomic blocks
  + Can explicitly represent dependencies among operations within an enlarged block

- Disadvantages:
  -- "Fault operations" can lead to work to be wasted (atomicity)
  -- Code bloat (multiple copies of the same basic block exists in the binary and possibly in I-cache)
    -- Need to predict which enlarged block comes next

- Optimizations
  - Within an enlarged block, the compiler can perform optimizations that cannot normally be performed across basic blocks

Block Structured ISA (III)

- Hao et al., "Increasing the instruction fetch rate via block-structured instruction set architectures," MICRO 1996.

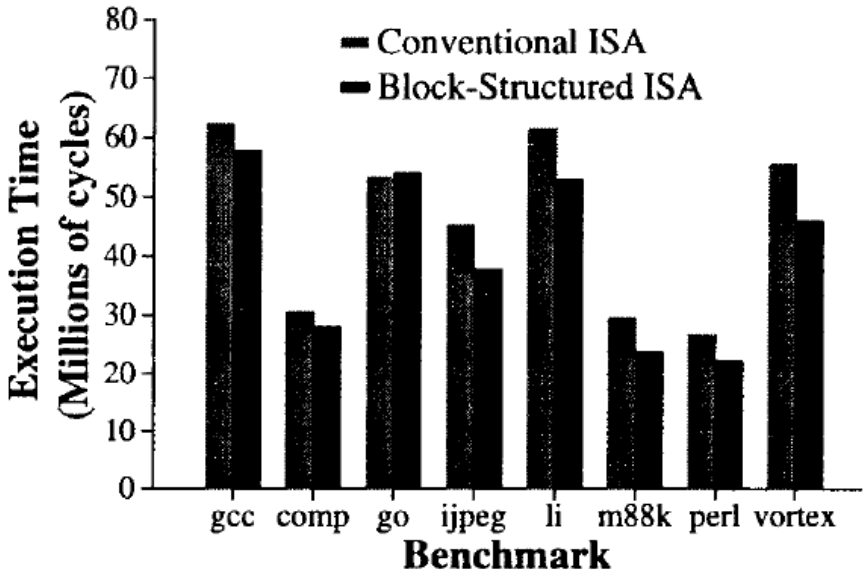


Figure 3. Performance comparison of block-structured ISA executables and conventional ISA executables.

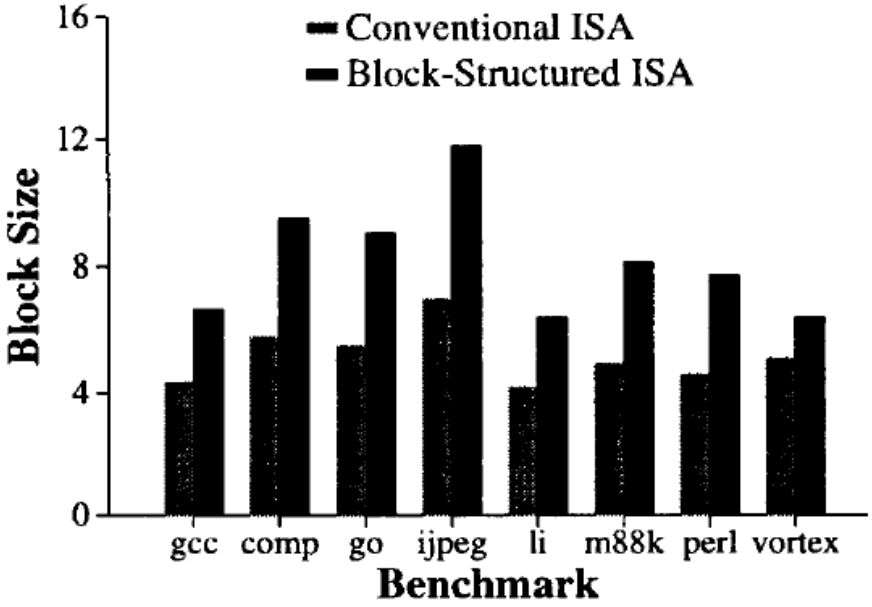


Figure 5. Average block sizes for block-structured and conventional ISA executables.

Superblock vs. BS-ISA

- Superblock
  - Single-entry, multiple exit code block
  - Not atomic
  - Compiler inserts fix-up code on superblock side exit

- BS-ISA blocks
  - Single-entry, single exit
  - Atomic

Superblock vs. BS-ISA

- Superblock
  - + No ISA support needed
  - -- Optimizes for only 1 frequently executed path
    - -- Not good if dynamic path deviates from profiled path → missed opportunity to optimize another path

- Block Structured ISA
  - + Enables optimization of multiple paths and their dynamic selection.
  - + Dynamic prediction to choose the next enlarged block. Can dynamically adapt to changes in frequently executed paths at run-time
  - + Atomicity can enable more aggressive code optimization
  - -- Code bloat becomes severe as more blocks are combined
  - -- Requires "next enlarged block" prediction, ISA+HW support
  - -- More wasted work on "fault" due to atomicity requirement