15-740/18-740
Computer Architecture
Lecture 25: Control Flow II

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Midterm II
  - November 22

- Project Poster Session
  - December 10
  - NSH Atrium
    - 2:30-6:30pm

Readings

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

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

Approaches to Conditional Branch Handling

- Branch prediction
  - Static
  - Dynamic

- Eliminating branches

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

  II. Predicate combining (and condition registers)

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

Direction Prediction

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

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

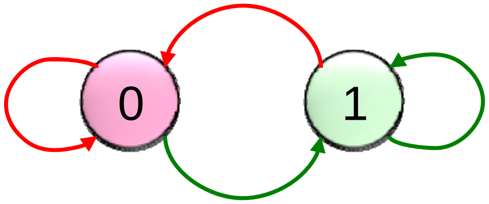
Two-Bit Counter Based Prediction

● Predict NT

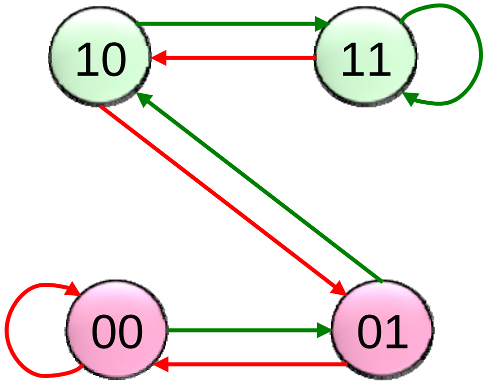
○ Predict T

→ Transistion on T outcome

→ Transistion on NT outcome



Finite State Machine for Last-time Predictor



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

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

Two-Bit Counter Based Prediction

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

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

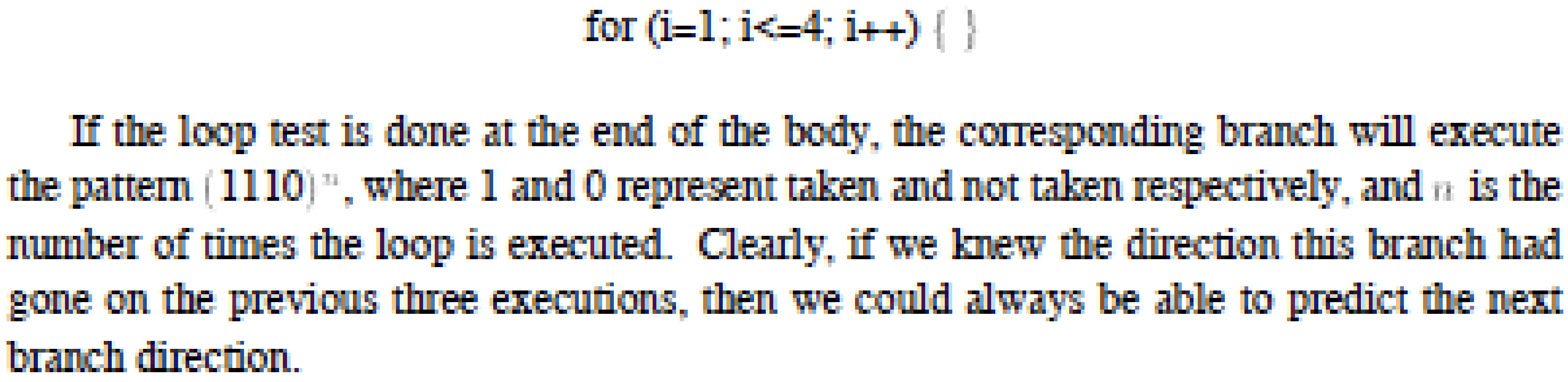
  TNTNTNTNTNTNTNTNTNTN → 50% accuracy

  (assuming init to weakly taken)

+ Better prediction accuracy

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

Can We Do Better?



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

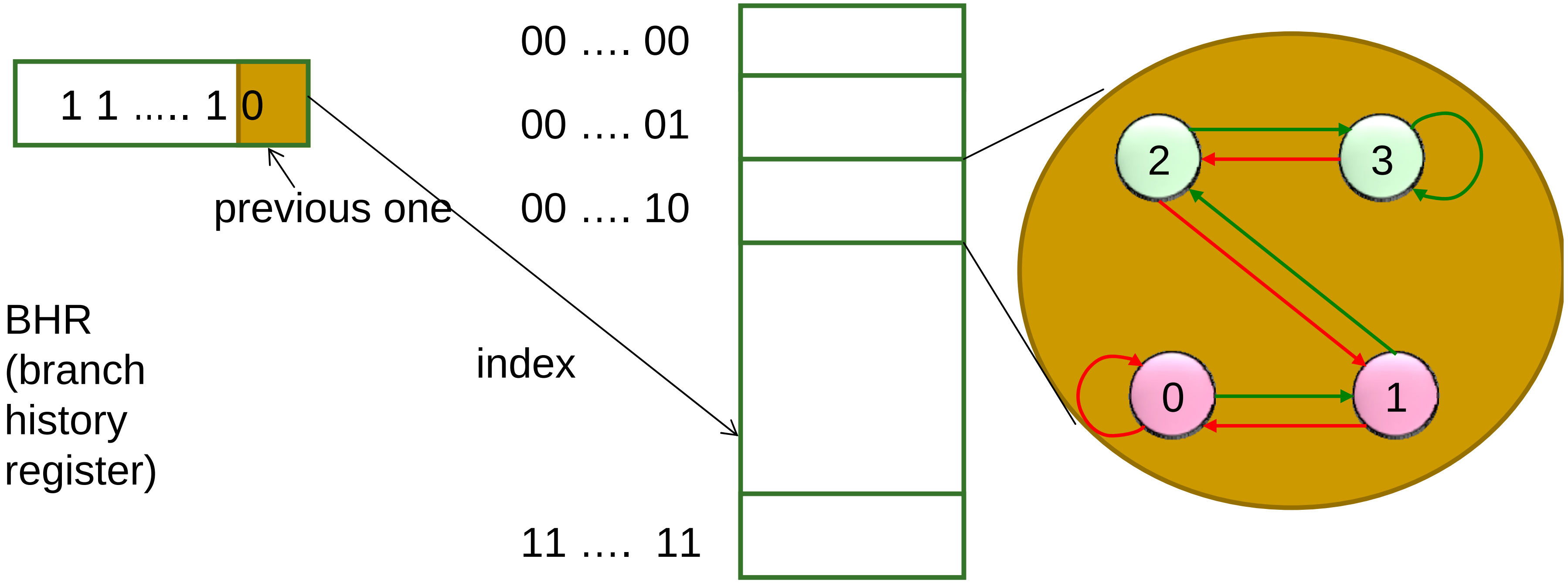
If the loop test is done at the end of the body, the corresponding branch will execute the pattern $(1110)^n$, where 1 and 0 represent taken and not taken respectively, and n is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions,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

- McFarling, “Combining Branch Predictors,” DEC WRL TR 1993.

Two Level Branch Predictors

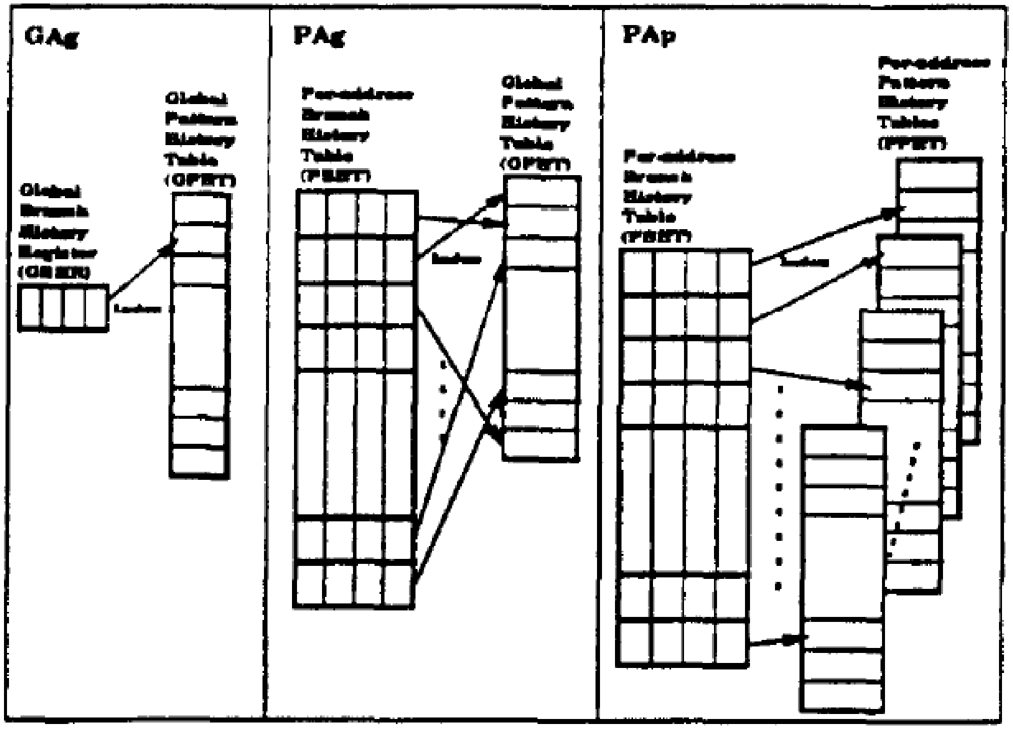
- First level: Branch history register (N bits)
  - The direction of last N branches
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen?

Pattern History Table (PHT)



Two-Level Predictor Variations

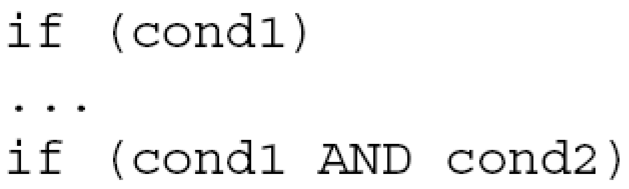
- BHR can be global (G), per set of branches (S), or per branch (P)
- PHT counters can be adaptive (A) or static (S)
- PHT can be global (g), per set of branches (s), or per branch (p)


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

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

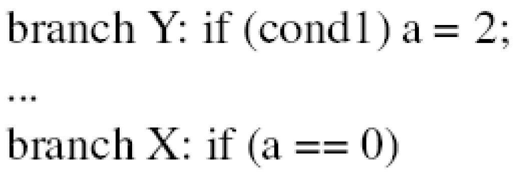
Global Branch Correlation (I)

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



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

- If first branch not taken, second also not taken



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

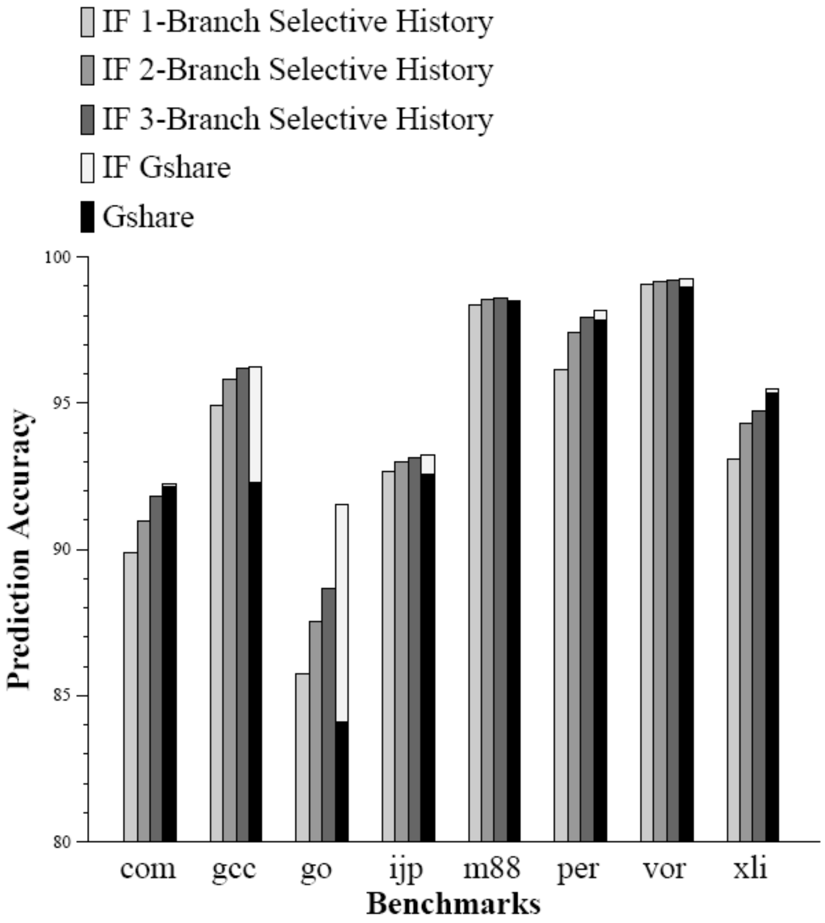
- If first branch taken, second definitely not taken

Global Branch Correlation (II)

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

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

- Only 3 past branches' directions really matter
- Evers et al., "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work," ISCA 1998.

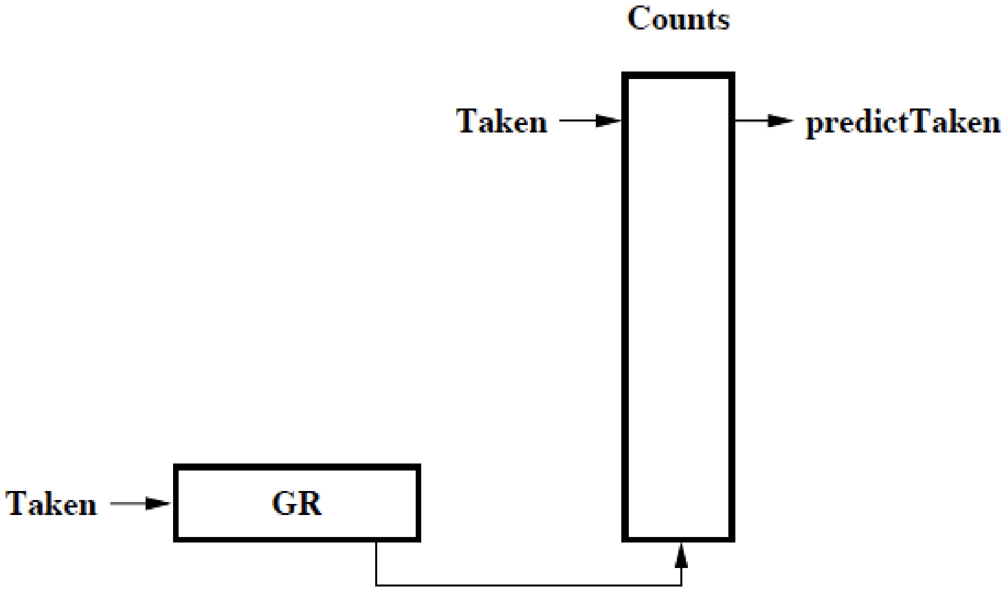


Global Two-Level Prediction

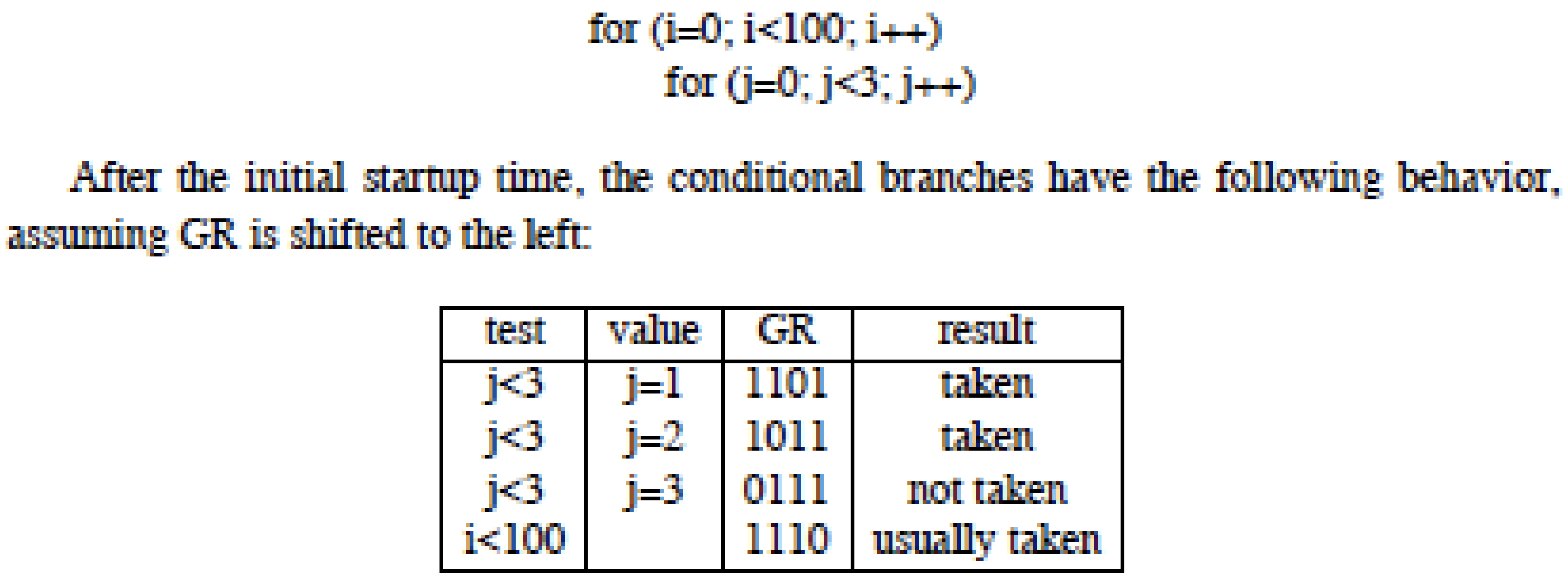
- Idea: Have a single history register for all branches (called global history register)

+ Exploits correlation between different branches (as well as the instances of the same branch)

-- Different branches interfere with each other in the history register → cannot separate the local history of each branch



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:

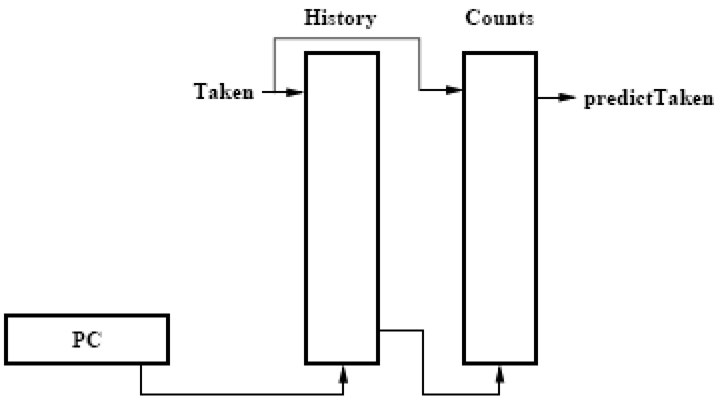
test	value	GR	result
j<3	j=1	1101	taken
j<3	j=2	1011	taken
j<3	j=3	0111	not taken
i<100		1110	usually taken

Pentium Pro Branch Predictor

- GAs
- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
    - PHT determined by lower order bits of the branch address

Local Two-Level Prediction

- PAg, Pas, PAp
- Global history register produces interference
  - Different branches can go different ways for the same history
- Idea: Have a per-branch history register

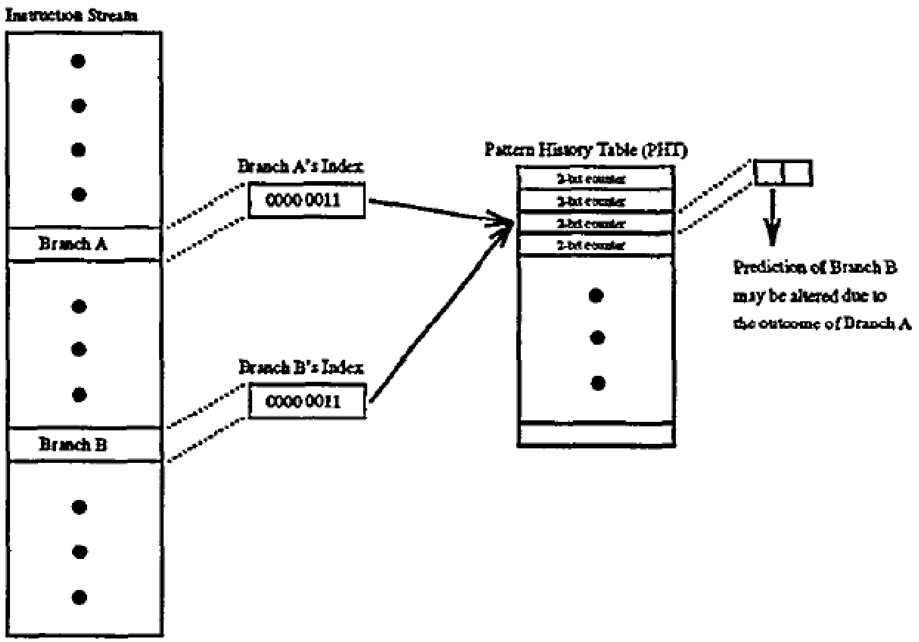


+ No interference in the history register between branches

-- Cannot exploit global branch correlation

Interference in the PHTs

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



- Interference can be eliminated by dedicating a PHT per branch

  -- Too much hardware cost
- How else can you eliminate interference?

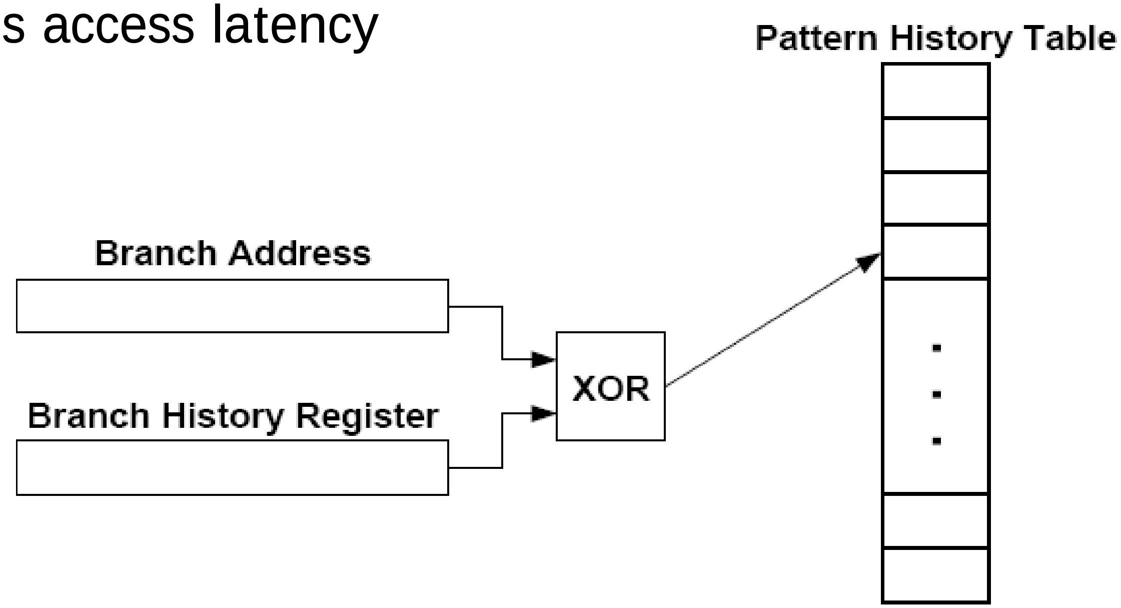
Reducing Interference in PHTs (II)

- 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



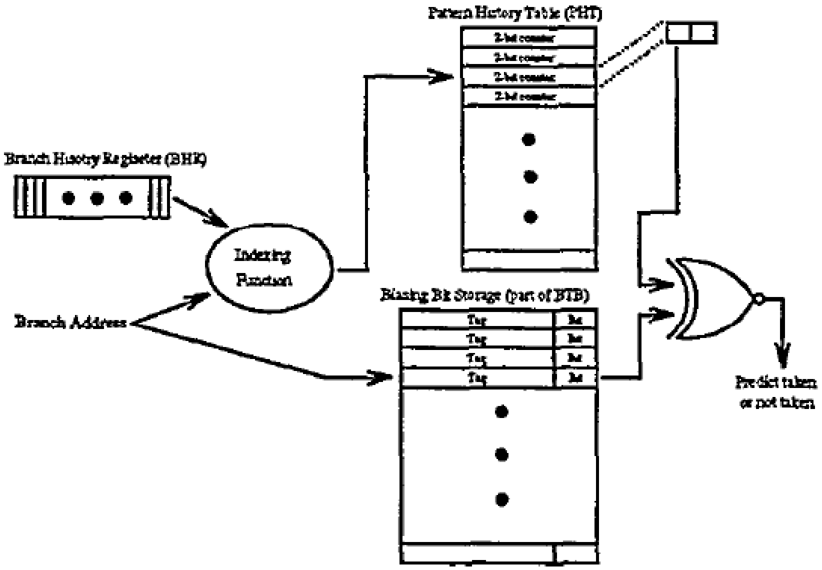
  - McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

Reducing Interference in PHTs (III)

- 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)


Pattern History Table (PHT)
2-bit counter
2-bit counter
2-bit counter
2-bit counter
Branch History Register (BHR)
Indexing Function
Branch Address
Biasing Bit Storage (part of BTB)
Tag
Bit
Tag
Bit
Tag
Bit
Tag
Bit
Predict taken or not taken

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 (b1, b2) have taken rates of 85% and 15%.
- Assume they conflict in the PHT

- 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%

- Agree prediction reduces the probability that two branches have opposite predictions in the PHT entry
  - Works because most branches are biased (not 50% taken)

Hybrid Branch Predictors

- Idea: Use more than one type of predictors (i.e., 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)

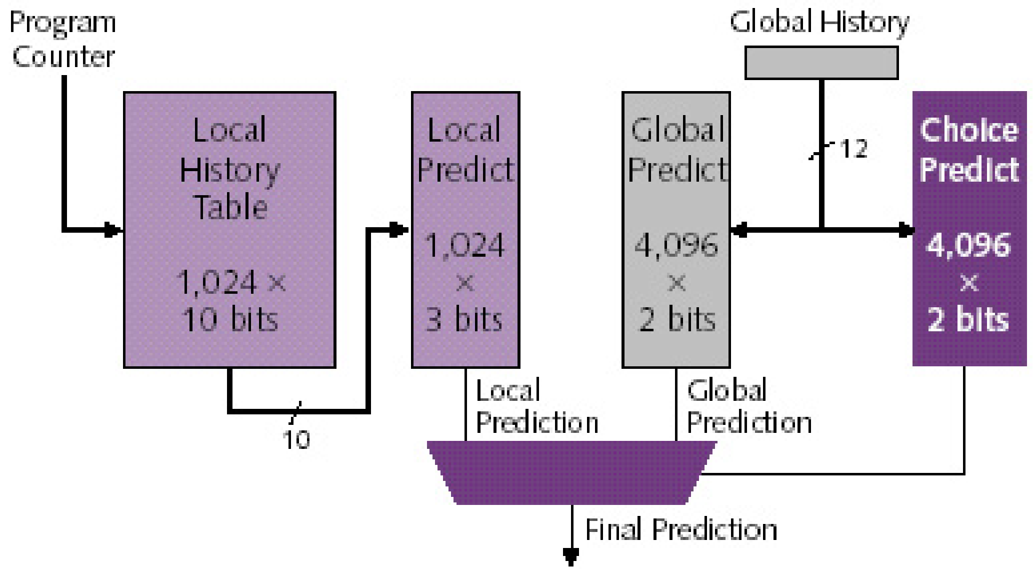
- Disadvantages:

  -- 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
- 32-entry return address stack
- Predictor tables are reset on a context switch

Effect on Prediction Accuracy

- Bimodal: table of 2bc indexed by branch address

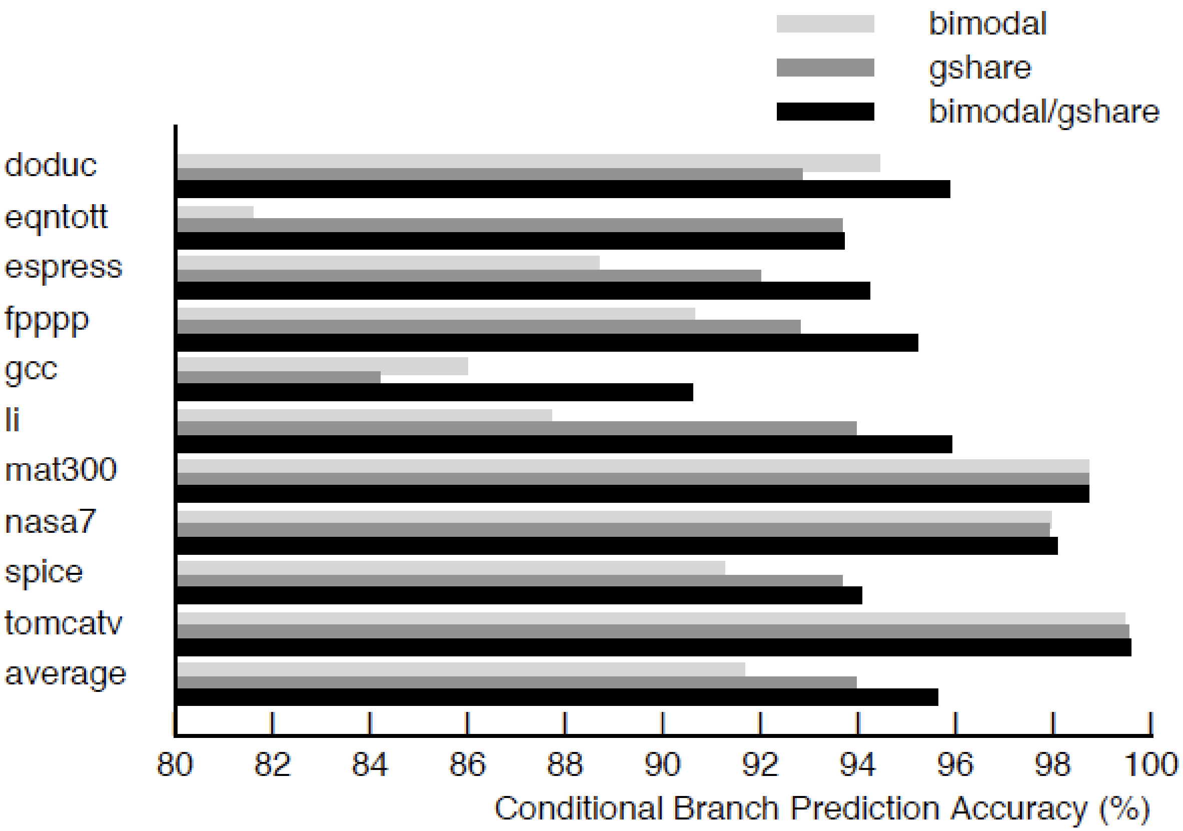


Figure 13: Combined Predictor Performance by Benchmark

Improved Branch Prediction Algorithms

- Perceptron predictor
  - Learns the correlations between branches in the global history register and the current branch using a perceptron
  - Past branches that are highly correlated have larger weights and influence the prediction outcome more
  - Jimenez and Lin, "Dynamic Branch Prediction with Perceptrons," HPCA 2001.

- Enhanced hybrid predictors
  - Multi-hybrid with different history lengths
  - Seznec, "Analysis of the O-GEometric History Length Branch Predictor," ISCA 2005.

- Pre-execution
  - Similar to pre-execution based prefetching
  - Chappell et al., "Difficult-Path Branch Prediction Using Subordinate Microthreads," ISCA 2002.

Call and Return Prediction

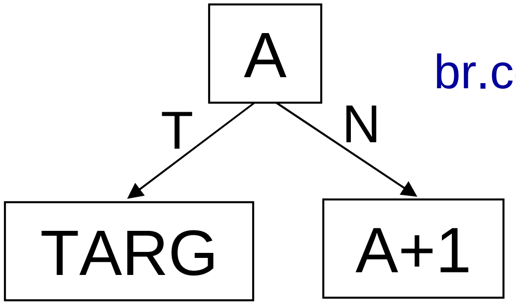
- Direct calls are easy to predict
  - Always taken, single target
  - Call marked in BTB, target predicted by BTB

- 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

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

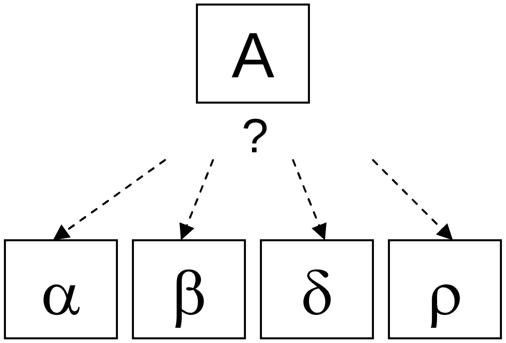
Indirect Branch Prediction (I)

- 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

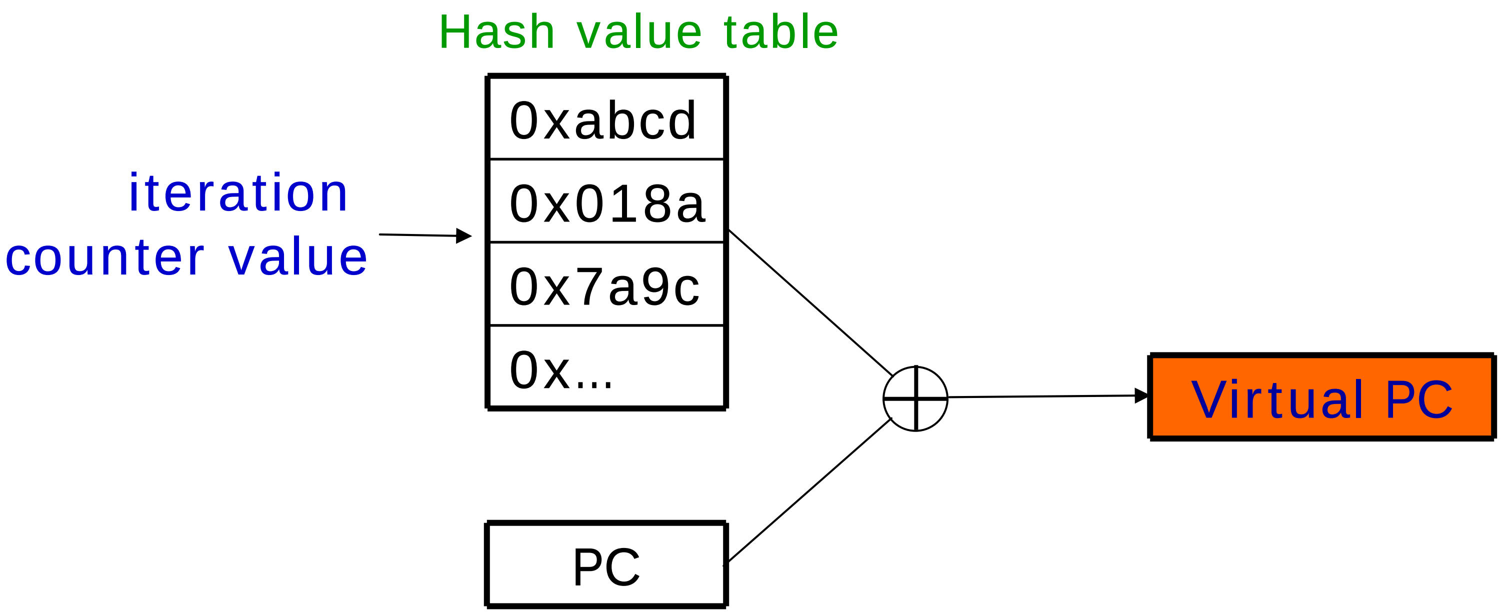
  -- An indirect branch maps to (too) many entries in BTB

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

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

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)

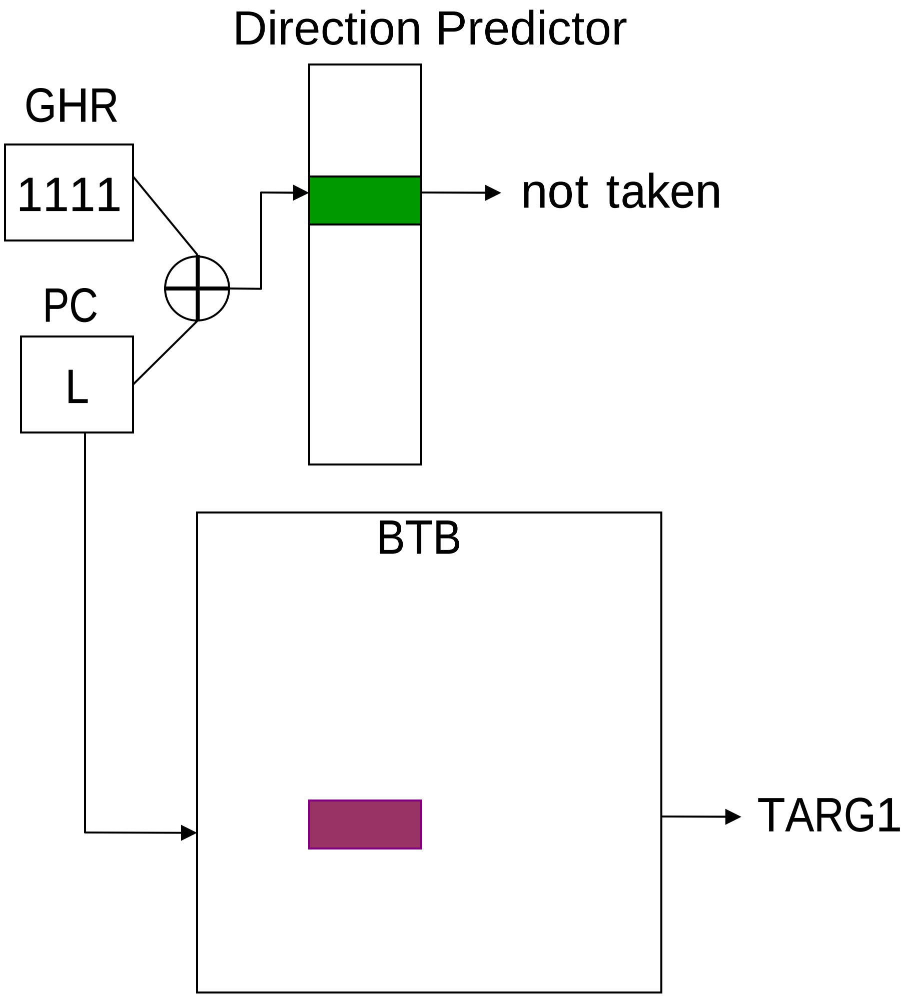
Real Instruction

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

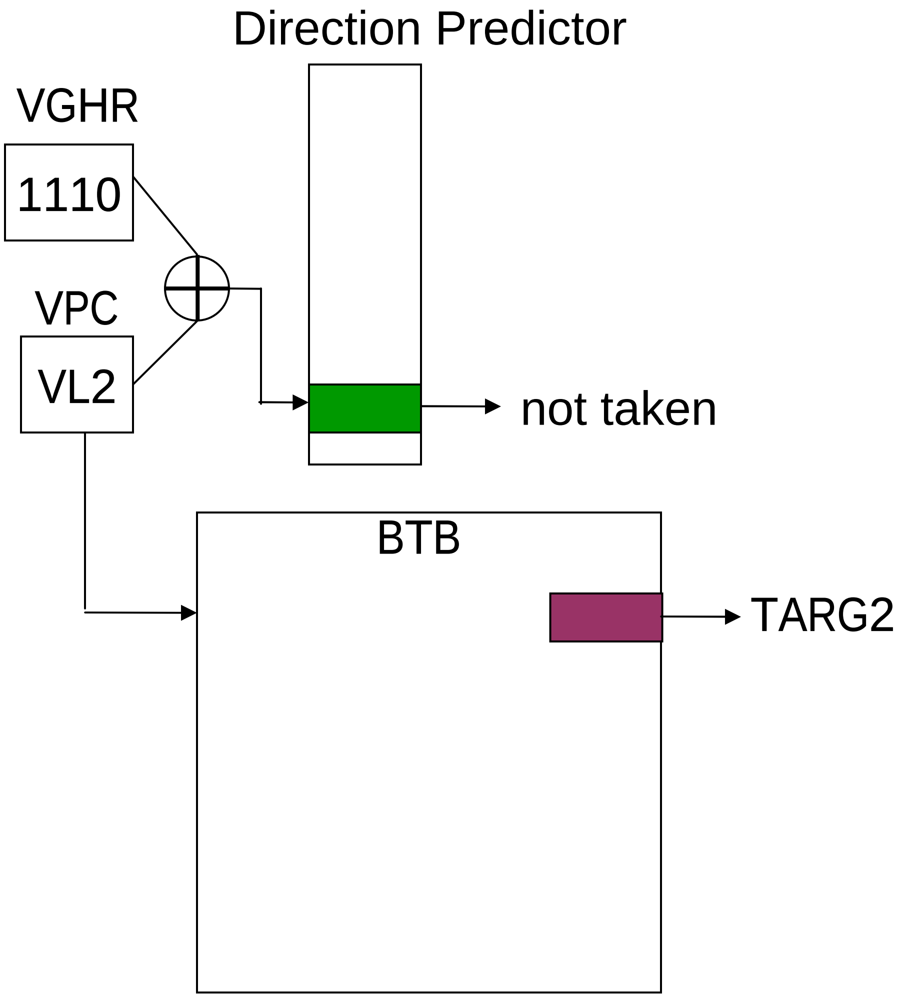
cond. jump TARG1 // VPC: L

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

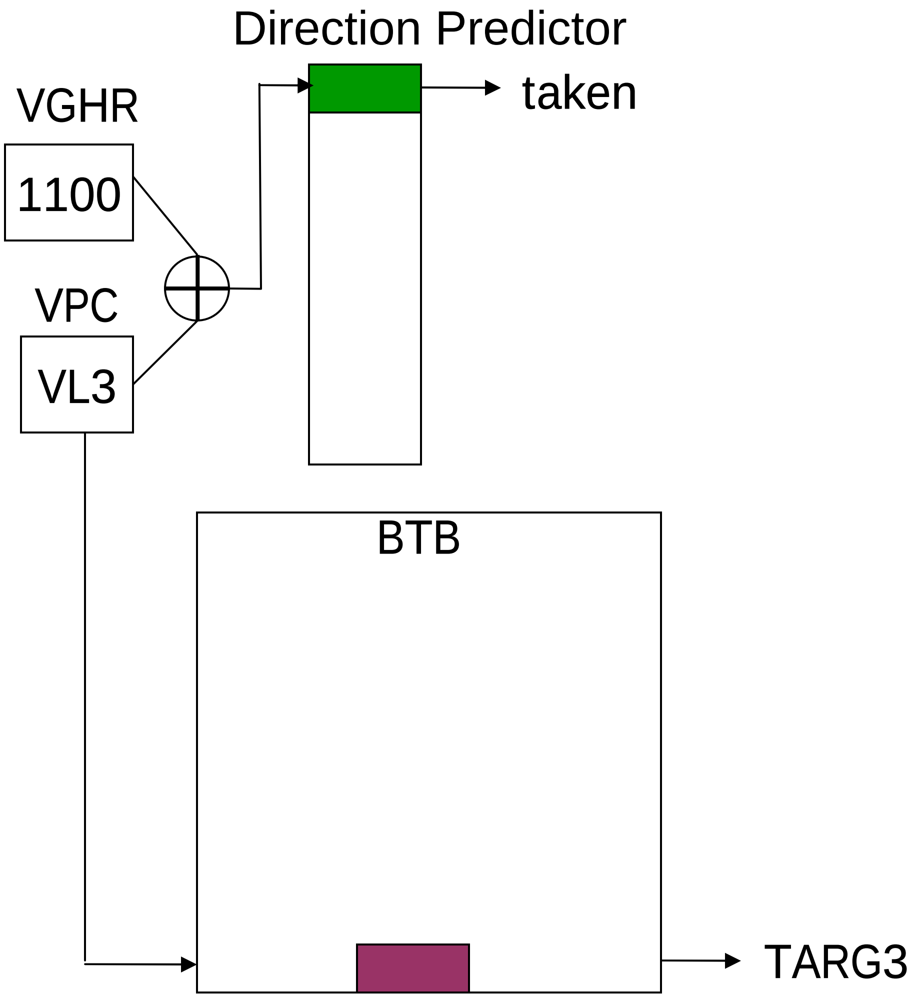
cond. jump TARG1 // VPC: L

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)

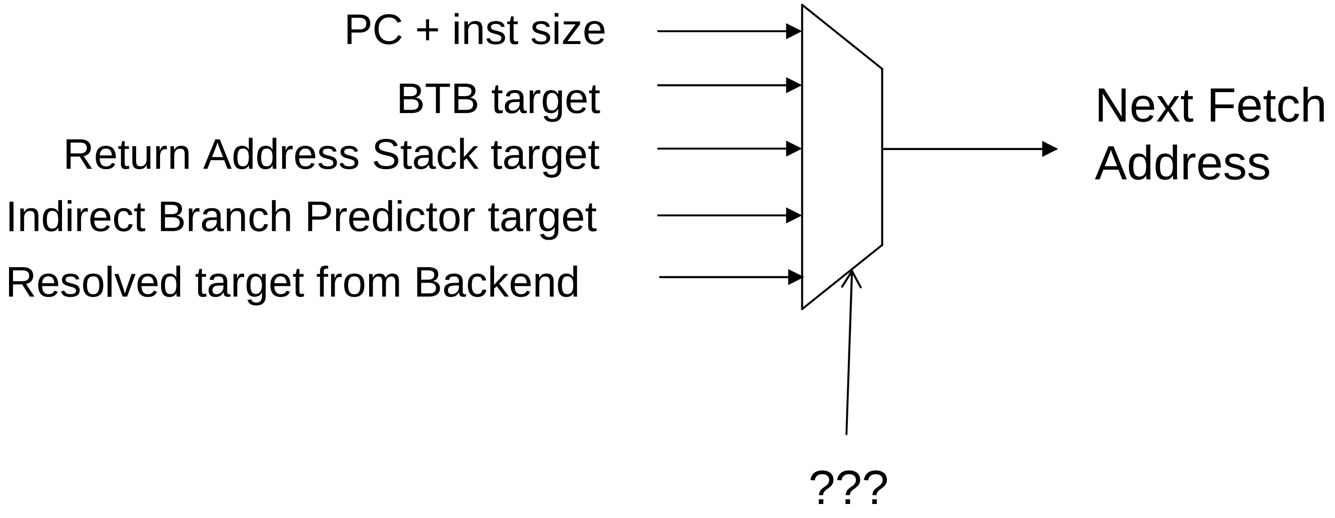
- Need to identify a branch before it is fetched

- How do we do this?
  - BTB hit → indicates that the fetched instruction is a branch
  - BTB entry contains the “type” of the branch

- 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


PC + inst size
BTB target
Return Address Stack target
Indirect Branch Predictor target
Resolved target from Backend
Next Fetch Address
???

Issues in Branch Prediction (III)

- State recovery upon misprediction
  - Misprediction detected when branch executes
  - Need to flush all instructions younger than the branch
    - Easy to invalidate instructions not yet renamed
    - Need to invalidate instructions in reservation stations and reorder buffer
  - Need to recover the Register Alias Table
    - Pentium 4: Retirement RAT copied to Frontend RAT

      + Simple

      -- Increases recovery latency (Branch has to be the oldest instruction in the machine!) ← **Why is this not as bad???**
    - Alpha 21264: Checkpoint RAT when branch is renamed, recover to checkpoint when misprediction detected

      + Immediate recovery of RAT

      -- More expensive (multiple RATs)

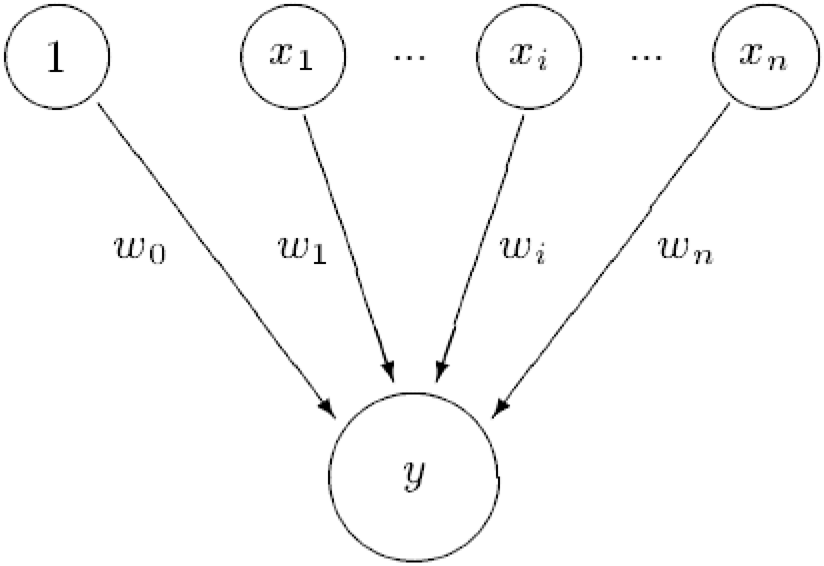
Open Research Issues in Branch Prediction

- Better algorithms
  - Machine learning techniques?
    - Needs to be low cost and *fast*

- Progressive evaluation of earlier prediction for a branch
  - As branch moves through the pipeline, more information becomes available → can we use this to override earlier prediction?
  - Falcon et al., "Prophet-critic hybrid branch prediction," ISCA 2004.

Perceptron Branch Predictor (I)

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



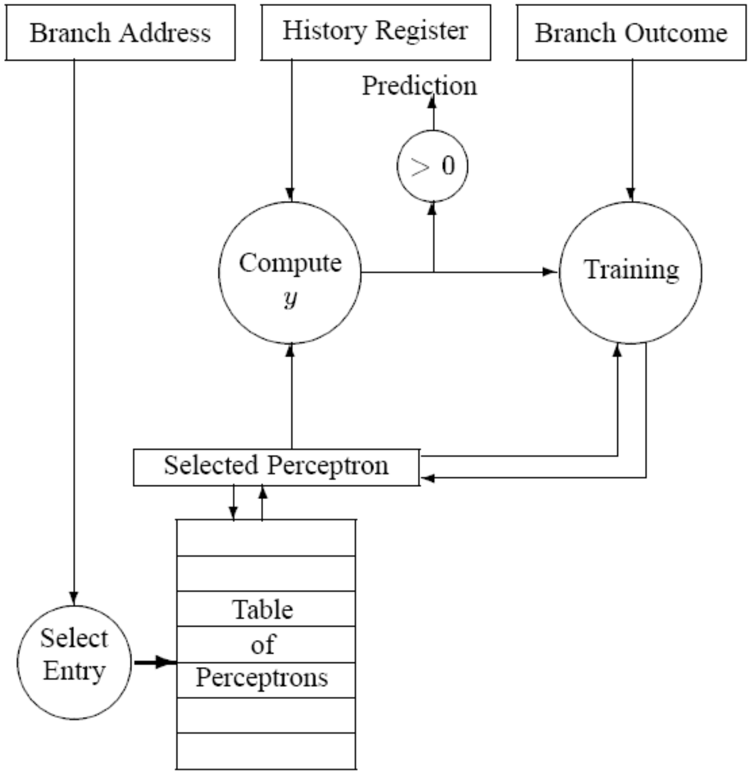
Each branch associated with a perceptron

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

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

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

Perceptron Branch Predictor (II)



Prediction function:

Dot product of GHR and perceptron weights

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

Output compared to 0

Bias weight (bias of branch independent of the history)

Training function:

```
if sign($y_{out}$) $\neq t$ or $|y_{out}| \leq \theta$ 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

Approaches to Conditional Branch Handling

- Branch prediction
  - Static
  - Dynamic

- Eliminating branches

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

  II. Predicate combining (and condition registers)

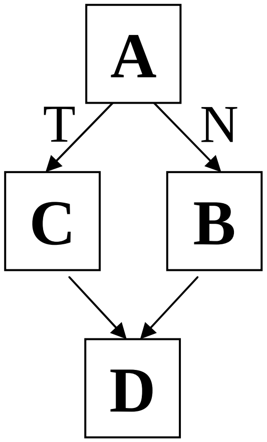
- Multi-path execution
- Delayed branching (branch delay slot)
- Fine-grained multithreading

Predication (Predicated Execution)

- Idea: Compiler converts control dependency into a data dependency → 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)

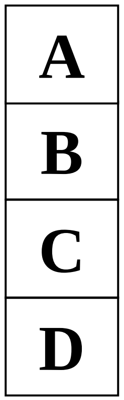
```
if (cond) {
      b = 0;
}
else {
      b = 1;
}
```

(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
```

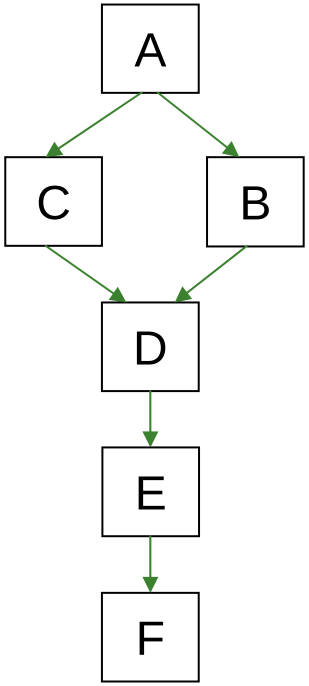
Conditional Move Operations

- 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

							F	E	D	C	B	A

nop

Branch Prediction

Fetch Decode Rename Schedule RegisterRead Execute

								F	E	D	B	A

Pipeline flush!!

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

    + Vectorization with control flow

  + Reduces fetch breaks (straight-line 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, phase, control-flow path.

  -- Additional hardware and ISA support (complicates renaming and OOO)

  -- Cannot eliminate all hard to predict branches

    -- Complex control flow graphs, function calls, and loop branches

  -- Additional data dependencies delay execution (problem esp. for easy branches)