EE382A Lecture 5:

Branch Prediction

Department of Electrical Engineering
Stanford University

`http://eeclass.stanford.edu/ee382a`

Announcements

- Project proposal due on Mo 10/14
  - List the group members
  - Describe the topic including why it is important and your thesis
  - Describe the methodology you will use (experiments, tools, machines)
  - Statement of expected results
  - Few key references to related work
- Still missing most photos

Branch Prediction Review

- Why do we need branch prediction?
- What do we need to predict about branches?
- Why are branches predictable?
- What mechanisms do we need for branch prediction?

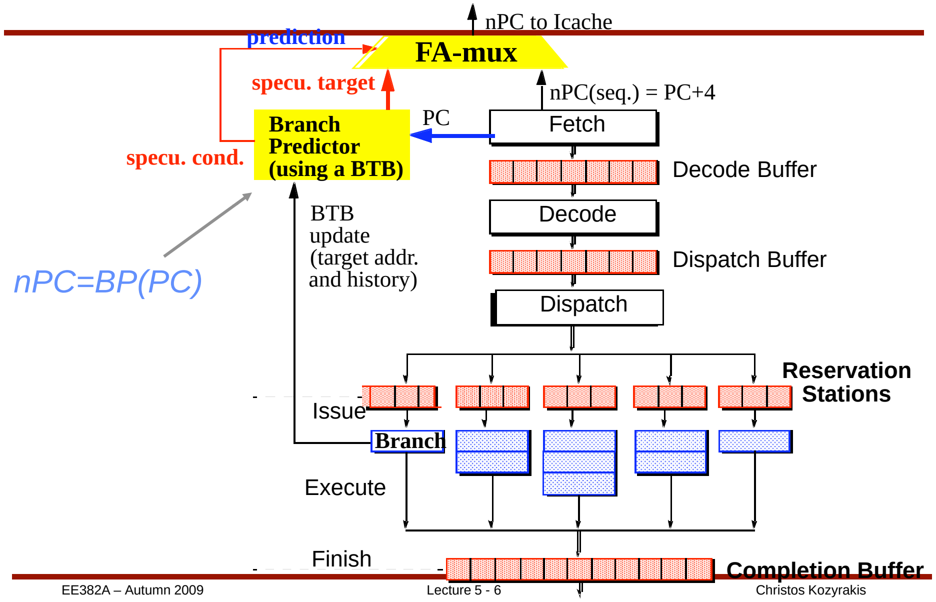
Static Branch Prediction

- Option #1: based on type or use of instruction
  - E.g., assume backwards branches are taken (predicting a loop)
  - Can be used as a backup even if dynamic schemes are used
- Option #2: compiler or profile branch prediction
  - Collect information from instrumented run(s)
  - Recompile program with branch annotations (hints) for prediction
    - See heuristics list in next slide
  - Can achieve 75% to 80% prediction accuracy
- Why would dynamic branch prediction do better?

Heuristics for Static Prediction
(Ball & Larus, PPoPP1993)

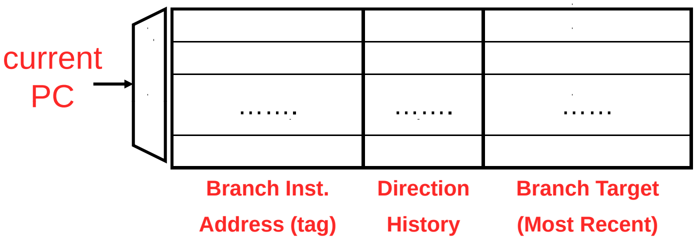
Heuristic	Description
Loop Branch	If the branch target is back to the head of a loop, predict taken.
Pointer	If a branch compares a pointer with NULL, or if two pointers are compared, predict in the direction that corresponds to the pointer being not NULL, or the two pointers not being equal.
Opcode	If a branch is testing that an integer is less than zero, less than or equal to zero, or equal to a constant, predict in the direction that corresponds to the test evaluating to false.
Guard	If the operand of the branch instruction is a register that gets used before being redefined in the successor block, predict that the branch goes to the successor block.
Loop Exit	If a branch occurs inside a loop, and neither of the targets is the loop head, then predict that the branch does not go to the successor that is the loop exit.
Loop Header	Predict that the successor block of a branch that is a loop header or a loop pre-header is taken.
Call	If a successor block contains a subroutine call, predict that the branch goes to that successor block.
Store	If a successor block contains a store instruction, predict that the branch does not go to that successor block.
Return	If a successor block contains a return from subroutine instruction, predict that the branch does not go to that successor block.

Dynamic Branch Prediction Using History



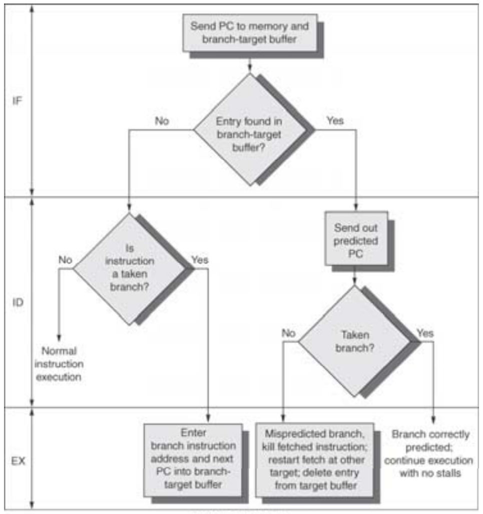
Review: Branch Target Buffer (BTB)

- A small “cache-like” memory in the instruction fetch stage

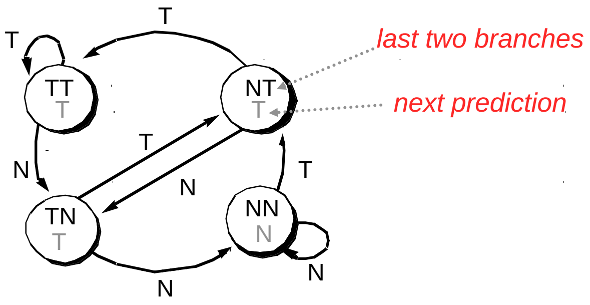


- Remembers previously executed branches, their addresses, information to aid prediction, and most recent target addresses
- Predicts both branch direction and target
- When branch is actually resolved, BTB must be updated updated

Review: BTB Algorithm


IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

Review: Keeping Track of Direction History: 2-bit Finite State Machines



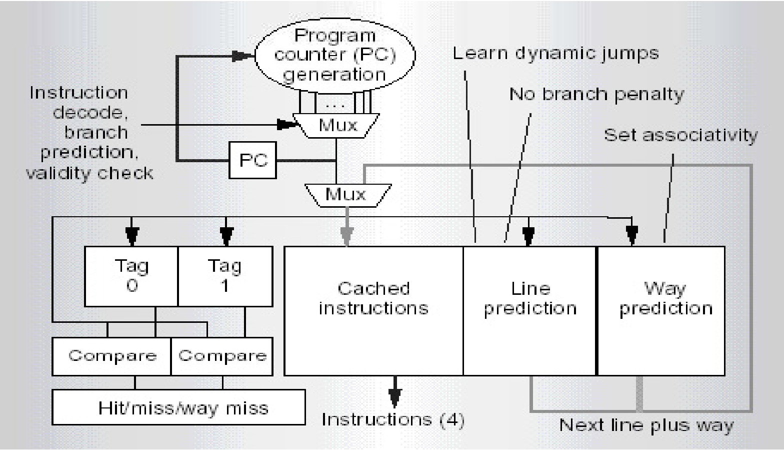
- History avoids mispredictions due to one time events
  - Canonical example: loop exit
- 2-bit FSM as good as n-bit FSM
- Saturating counter as good as any FSM

I-Cache & BTB Integration

- Why does this make sense?
- Place a BTB entry in each cache line
  - Each cache line tells you which line to address next
  - Do no need the full PC, just an index the cache + a way select
  - This is called way & line prediction

- Implemented in Alpha 21264 processor
  - On refills, prediction value points to the next sequential fetch line.
  - Prediction is trained later on as program executes…
    - When correct targets are known…
  - Line prediction is verified in next cycle. If line-way prediction is incorrect, slot stage is flushed and PC generated using instruction info and direction prediction

Alpha 21264 Line & Way Prediction

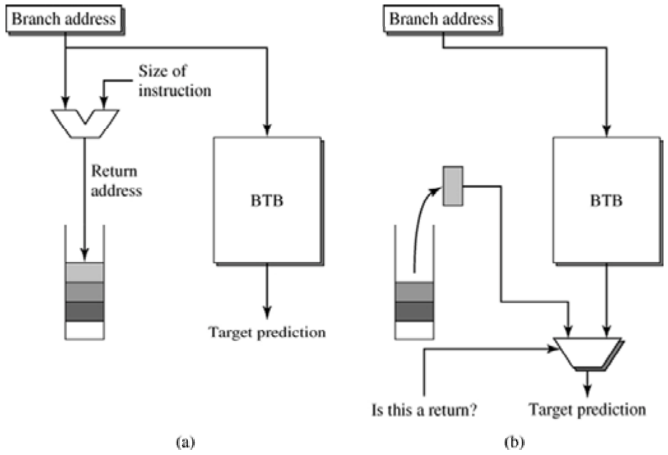


Source: IEEE Micro, March-April 1999

Branch Target Prediction for Function Returns

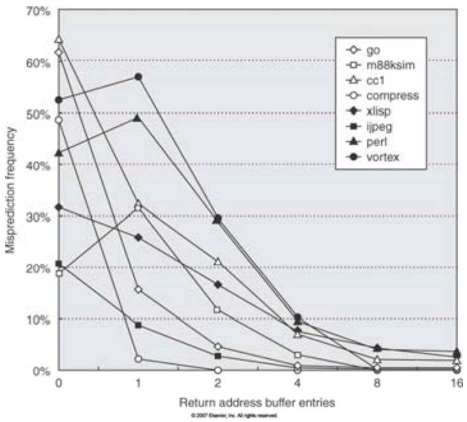
- In most languages, function calls are fully nested
  - If you call A() ⇒ B() ⇒ C() ⇒ D()
  - Your return targets are PCc ⇒ PCb ⇒ PCa ⇒ PCmain
- Return address stack (RAS)
  - A FILO structure for capturing function return addresses
  - Operation
    - On a function call retirement, push call PC into the stack
    - On a function return, use the top value in the stack & pop
  - A 16-entry RAS can predict returns almost perfectly
    - Most programs do not have such a deep call tree
  - Sources of RAS inaccuracies
    - Deep call statements (circular buffer overflow – will lose older calls)
    - Setjmp and longjmp C functions (irregular call semantics)

RAS Operation



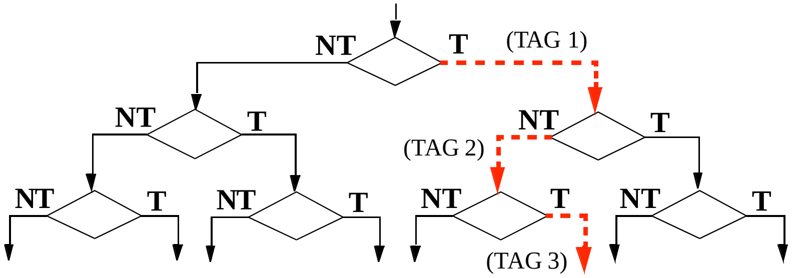
(a) (b)

RAS Effectiveness & Size (SPEC CPU'95)



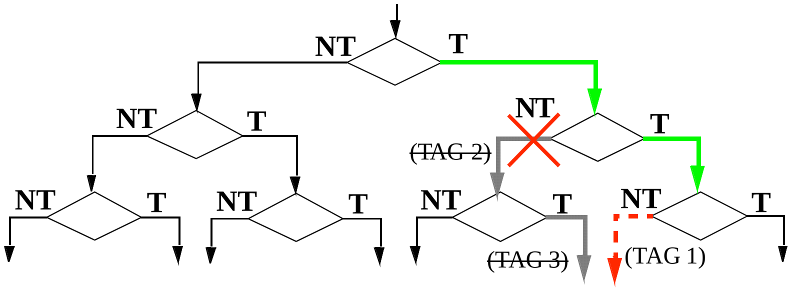
- Can you see any catch?

Tracking Branch Speculation



- At leading speculation
  - For each branch, remember the predicted branch outcome
  - For each branch, assign a tag to each speculated branch (circular order)
  - Tag all following instructions with the same tag
- At trailing confirmation (case of correct prediction)
  - Remove the tag
  - Allow branch and all following instructions to retire (based on ROB order)

Recovering from Incorrect Speculation



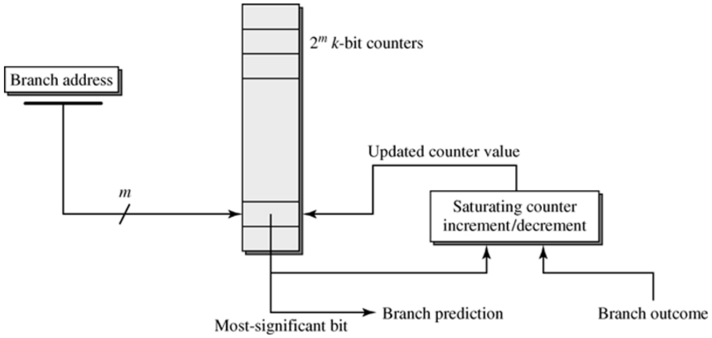
- Eliminate incorrect path
  - Must ensure that the misspeculated instructions produce no side effects
- Start new correct path
  - Must remember the alternate (non-predicted) path

Mis-speculation Recovery

- Eliminate incorrect path
  1. Use tag(s) to deallocate ROB entries with speculative instructions
     - Can structure ROB around groups of instructions with same tag
     - Leads to some inefficiency but makes tracking simpler
  2. Invalidate all instructions in the decode and dispatch buffers, as well as those in reservation stations

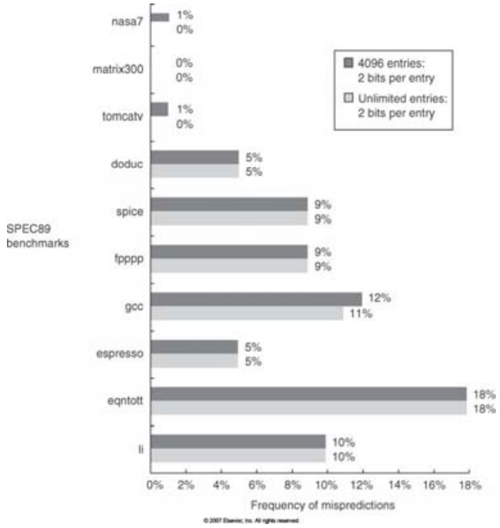
- Start new correct path
  1. Update PC with computed branch target (if predicted NT)
  2. Update PC with sequential instruction address (if predicted T)
  3. Can begin speculation again at next branch

Review: Coupling the BTB with a Simple Branch History Table (BHT)



- Why would you have a BHT in addition to the BTB?
- How would you use its predictions?
- What if the BHT has a 2-cycle latency?
- See any shortcoming?

BHT Accuracy and Limitations



Branch Correlation

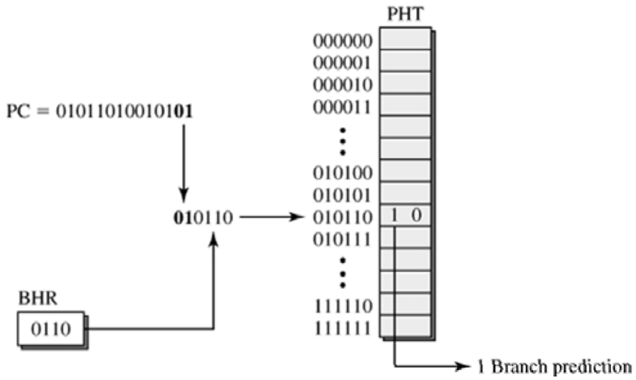
- So far, the prediction of each static branch instruction is based solely on its own past behavior and not the behaviors of other neighboring static branch instructions

- How about this one?

```
x=0;
If (someCondition) x=3;             /* Branch A*/
If (someOtherCondition) y+=19;      /* Branch B*/
If (x<=0) dosomething();            /* Branch C*?
```

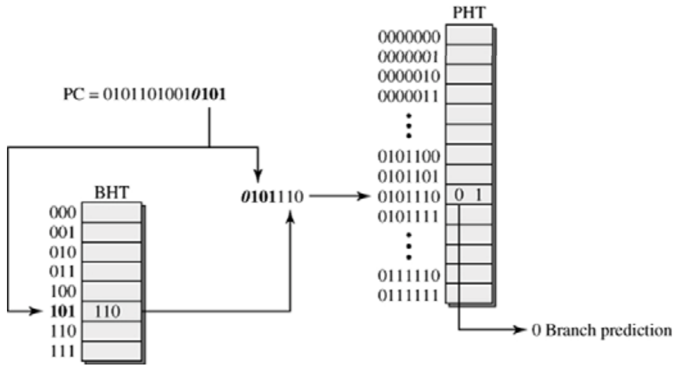
- Other correlation examples?

Global History Branch Predictor



- BHR: a shift register for global history
  - Shift in latest result in each cycle
  - Provides global context
- Advantages & shortcomings?

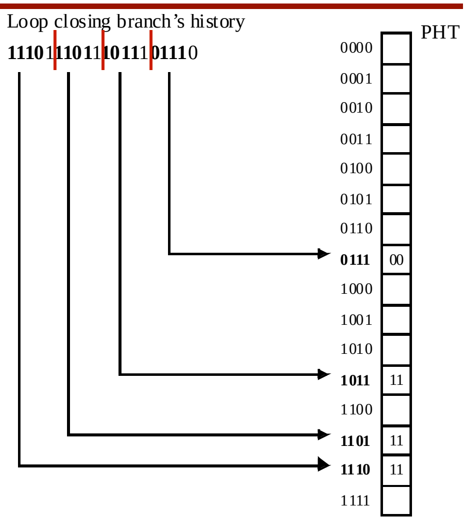
Local History Branch Predictor



- BHT keeps track of local history
  - Select entry based on PC bits; shift in latest result in each cycle
- Advantages & shortcomings?

Local History Predictor Example: Short Loops

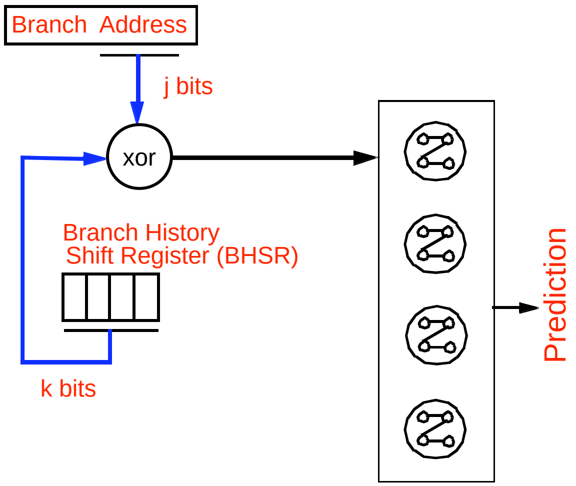
- Must identify the last iteration of short loop
  - Predict its branch not-taken
- BHT allows us to use a different PHT entry for each iteration of the loop
  - In this example, the loop has 4 iterations
  - '0111' entry predicts not taken



Two-level Adaptive Branch predictors Two-level Taxonomy

- Based on indices for branch history and pattern history
  - BHR: {G,P}: {Global history, Per-address history}
  - PHT: {g,p,s}: {Global, Per-address, Set}
    - g: use the BHR output as the address into the PHT
    - p: combine the BHR output with some bits from the PC
    - s: use an arbitrary hashing function for PHT addressing
  - 9 combinations: GAg, GAp, GAs, PAg, PAp, PAs, SAg, SAp and SAs
- Examples
  - Our global predictor so far is a GAp
  - Our local predictor so far was a PAp
- T. Yeh and Y. Patt. Two-Level Adaptive Branch Prediction. Intl. Symposium on Microarchitecture, November 1991.

Gshare Branch Prediction [McFarling]



Branch Address

j bits

xor

Branch History Shift Register (BHSR)

k bits

Prediction

BHT of 2 x $2^{\max(j,k)}$

Combining, Hybrid, or Tournament Branch Predictors

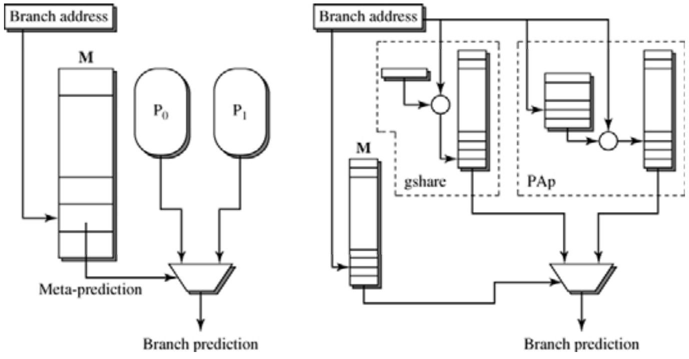
- What if different programs exhibit different patterns?

- Combining predictors: use multiple predictors
  - Each type tries to capture a particular program behavior
  - Use another history-based prediction scheme to “predict” which predictor should be used for a particular branch

You get the best of all worlds. This works quite well

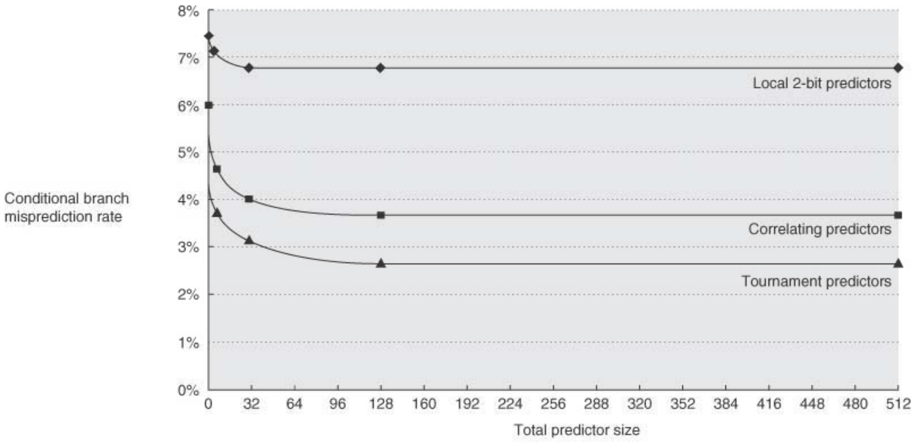
  - Variations:
    - Static prediction using software hints
    - Select from more than one alternative (multihybrid and fusion predictors)

Combining, Hybrid, or Tournament Branch Predictors

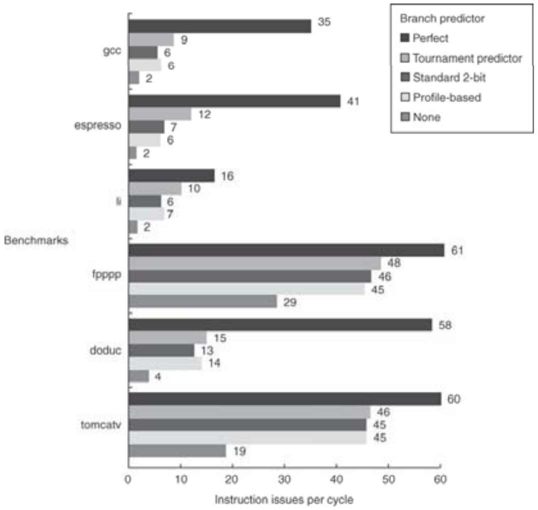


- E.g. Alpha 21264 used this approach
  - Predictor 1: a gshare with 12 bits of history (4K counters)
  - Predictor 2: a Pap with 1K history entries (10b) and 1K BHT
  - Selector: a 4K entry BHT

Comparison of Branch Predictor (SPEC’92)



Are We Done with Branch Predictors?



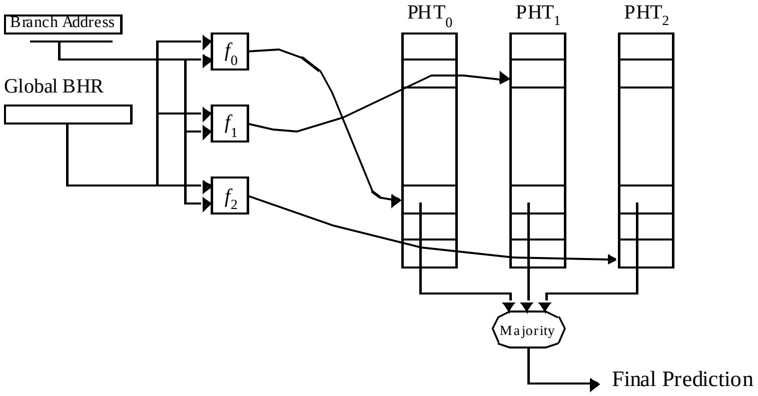
Causes for Mispredictions

- Fundamentally unpredictable branches
  - Cold miss, data dependent, …
- Training period
  - Need some time to warm up the predictor
  - The more patterns detected, the longer it takes to train
    - E.g. assume a global history predictor with 10 bits of history
    - Need to potentially train up to 2^10 entries for a specific branch
- Insufficient history or patterns
- Aliasing/interference
  - Branch predictors have limited capacity and no tags
  - Negative aliasing: two branches train same entry in opposite directions
  - Positive/neutral aliasing: two branches train same entry in same direction

Advanced Branch Predictors (see textbook for details)

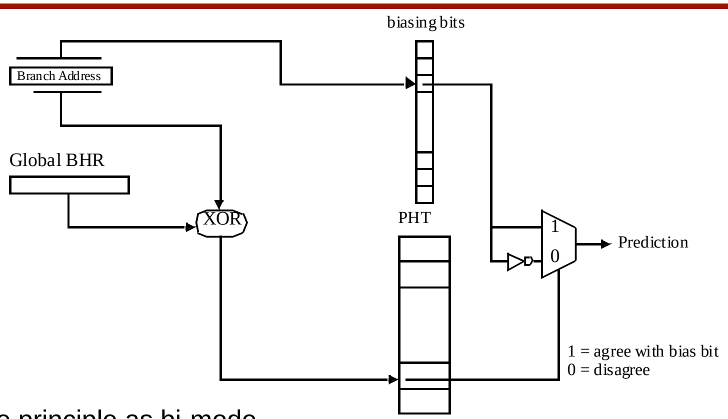
- Bi-mode predictor
  - Separate PHT for mostly taken and mostly non-taken branches
  - Eliminate negative aliasing
  - Use predictor to select the type
- G-skew predictor (used in Alpha EV8)
  - Use multiple hash-functions into PHTs & vote on outcome
  - Reduce chance of negative interference affecting prediction
- Agree predictor
  - BTB gives you a basic prediction
  - Extra PHT tells you if you should agree with the BTB
  - Biased branches have positive interference regardless of direction…
- YAGS
  - Keep a small tagged cache with branches that experience interference
- Other related ideas:
  - Branch filtering, selective branch inversion, alloyed history predictors, path history predictors, variable path length predictors, dynamic history length fitting predictors, loop counting predictors, percepton predictors, data-flow predictors, two-level predictors, analog circuit predictors, …

gskewed Predictor



- Multiple PHT banks indexed by different hash functions
  - Conflicting branch pair unlikely to conflict in more than one PHT
- Majority vote determines prediction
- Used in the cancelled Alpha 21464

Agree Predictor

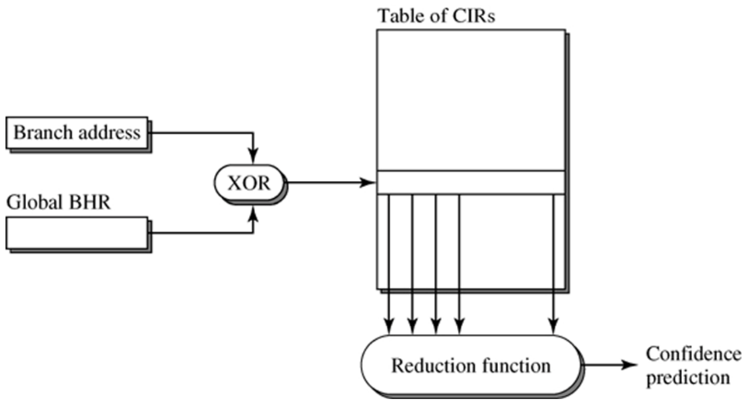


- Same principle as bi-mode
- PHT records whether branch bias matches outcome
  - Exploits 70-80% static predictability
- Used in in HP PA-8700

Prediction Confidence
A Very Useful Tool for Speculation

- Estimate if your prediction is likely to be correct
- Applications
  - Avoid fetching down unlikely path
    - Save time & power by waiting
  - Start executing down both paths (selective eager execution)
  - Switch to another thread (for multithreaded processors)
- Implementation
  - Naïve: don’t use NT or TN states in 2-bit counters
  - Better: array of CIR (correct/incorrect registers)
    - Shift in if last prediction was correct/incorrect
    - Count the number of 0s to determine confidence
  - Many other implementations are possible
    - Using counters etc

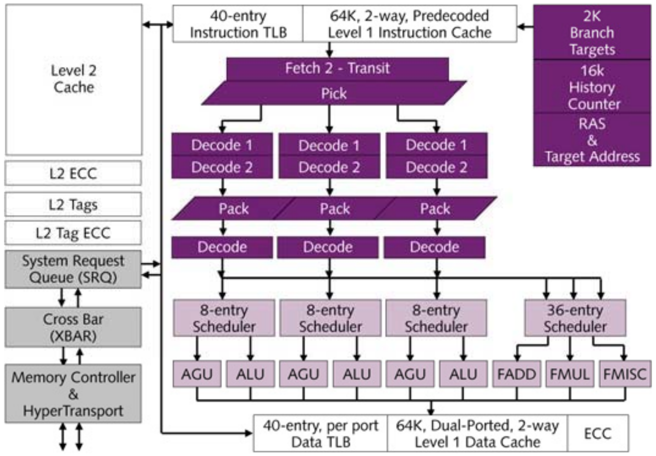
Branch Confidence Prediction



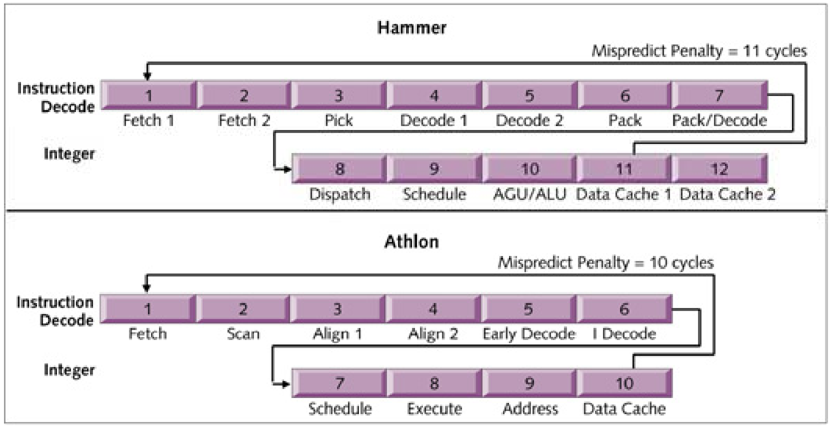
Other Branch Prediction Related Issues

- Multi-cycle BTB
  - Keep fetching sequentially, repair later (bubbles for taken branches)
  - Need pipelined access though
- BTB & predictor in series
  - Get fast target/direction prediction from BTB only
  - After decoding, use predictor to verify BTB
    - Causes a pipeline mini-flush if BTB was wrong
  - This approach allows for a much larger/slower predictor
- BTB and predictor integration
  - Can merge BTB with the local part of a predictor
  - Can merge both with I-cache entries
- Predictor/BTB/RAS updates
  - Can you see any issue?

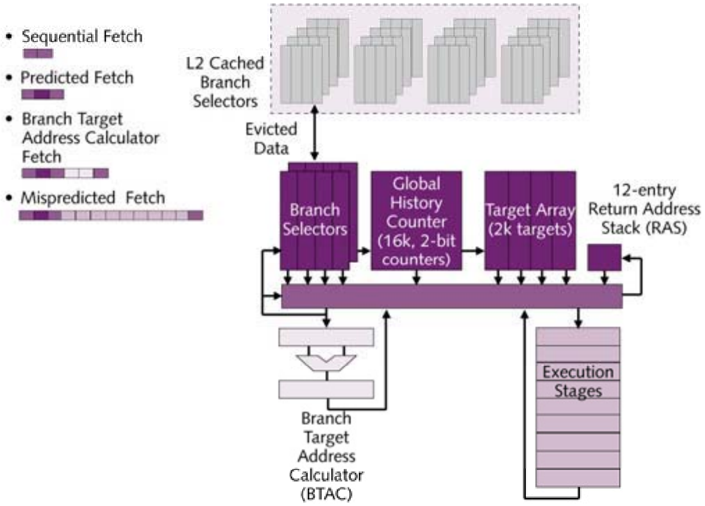
Fetch & Predict Example: AMD Opteron



Why is Prediction Important in Opteron?



Fetch & Predict Example: AMD Opteron


Sequential Fetch
Predicted Fetch
Branch Target Address Calculator Fetch
Mispredicted Fetch
L2 Cached Branch Selectors
Evicted Data
Branch Selectors
Global History Counter (16k, 2-bit counters)
Target Array (2k targets)
12-entry Return Address Stack (RAS)
Execution Stages
Branch Target Address Calculator (BTAC)

Fetch & Predict Example AMD Opteron

- Branch selectors: a local history table
  - Stored along with L1 and L2 (!) lines for instructions (2 bits per 2 bytes)
  - Allow to predict up to 2 branches + 1 return
- Global history counters: 4 counters per line
  - Remember there can be >1 branch per cache line
  - 4 bits from PC, 8 bits from global history
- BTB: each entry has up to 4 targets
  - Remember there can be >1 branch per cache line
  - Partial targets are just enough to index the I-Cache
  - They also integrate a selector between global and local history
- BTAC: a functional unit for early branch target calculation