Branch Prediction Review

The nub of the problem:

- In what pipeline stage does the processor fetch the next instruction?
- If that instruction is a conditional branch, when does the processor know whether the conditional branch is taken (execute code at the target address) or not taken (execute the sequential code)?
- What is the difference in cycles between them?

The cost of stalling until you know whether to branch

- number of cycles in between * branch frequency = the contribution to CPI due to branches

Predict the branch outcome to avoid stalling

Branch Prediction Review

Branch prediction:

- Try to resolve a branch hazard by predicting which path will be taken
- Proceed under that assumption
- Flush the wrong-path instructions from pipeline & fetch the right path if wrong

***Dynamic* branch prediction:**

- the prediction changes as program behavior changes
- branch prediction implemented in hardware (**static branch prediction** is done by the compiler)
- common algorithm:
  - predict the branch **taken** if branched the last time
  - predict the branch **not-taken** if didn’t branch the last time

Performance improvement depends on:

- how soon you can check the prediction
- whether the prediction is correct (here’s most of the innovation)

Branch Prediction Buffer

Branch prediction buffer

- small memory indexed by the lower bits of the address of a branch instruction during the fetch stage
- contains a prediction
  (which path the last branch to index to this BPB location took)
- do what the prediction says to do
- if the prediction is taken & it is right
  - only incur a one-cycle penalty – why?
- if the prediction is not taken & it is right
  - incur no penalty – why?
- if the prediction is wrong
  - change the prediction
  - also flush the pipeline – why?
  - penalty is the same as if there were no branch prediction

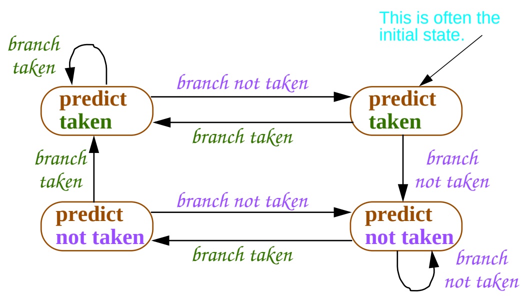
Two-bit Prediction

A single prediction bit does not work well with loops

- mispredicts the first & last iterations of a nested loop

Two-bit branch prediction for loops

- Algorithm: have to be wrong twice before the prediction is changed



- Works well when branches predominantly go in one direction
  - How? A second check is made to make sure that a short & temporary change of direction does not change the prediction away from the dominant direction
- What pattern is bad for two-bit branch prediction?

Two-bit Prediction

Often implemented as a table (a prediction buffer) of 2-bit **saturating counters**

- increase on a taken branch, not greater than 3
- decrease on a not-taken branch, not less than 0
- most significant bit is the prediction
- indexed by the low-order bits of the PC
- prediction improves with table size: **why?**

Could also be bits/counters associated with each cache line

Branch Prediction is More Important Today

Conditional branches still occur 25% to 15% of the time

Correct predictions are more important today – why?

- **pipelines deeper**
  branch not resolved until more cycles from fetching
  therefore the **misprediction penalty** greater
  - cycle times smaller: more emphasis on throughput (performance)
  - more functionality between fetch & execute
- **multiple instruction issue** (superscalars & VLIW)
  branch occurs almost every cycle
  - flushing & refetching more instructions
- **object-oriented programming**
  more indirect branches which harder to predict
- **dual of Amdahl’s Law**
  other forms of pipeline stalling are being addressed so the portion of CPI due to branch delays is relatively larger

All this means that the potential stalling due to branches is greater

Also, an opportunity – chips are denser so can consider sophisticated HW solutions

- hardware cost is small compared to the performance gain

Directions in Branch Prediction

1 Improve the prediction

- correlated predictor (Pentium Pro, Pentium III)
- hybrid local/global predictor (Alpha 21264)
- confidence predictors

2 Determine the target earlier

- branch target buffer (Pentium Pro, IA-64 Itanium)
- next address in I-cache (Alpha 21264, UltraSPARC)
- return address stack (Alpha 21264, IA-64 Itanium, MIPS R10000, Pentium Pro, UltraSPARC-3)

3 Reduce misprediction penalty

- fetch both instruction streams (IBM mainframes, SuperSPARC)
- resume buffer (MIPS R10000, UltraSPARC-3)

4 Eliminate the branch

- predicated execution (IA-64 Itanium, Alpha 21264)

1 Improve the Prediction

Correlated (global) branch prediction:

- the rationale: some branch outcomes are correlated

example: same cond var *example: related cond var*

```
if (d==0)            if (d==0)
...                     b=1;
if (d!=0)            if (b==1)
```

- so having the prediction depend on the outcome of only 1 branch might produce bad predictions

another example: related cond var

```
if (x==2)            /* branch 1 */
    x=0;
if (y==2)            /* branch 2 */
    y=0;
if (x!=y)            /* branch 3 */
    do this; else do that;
```

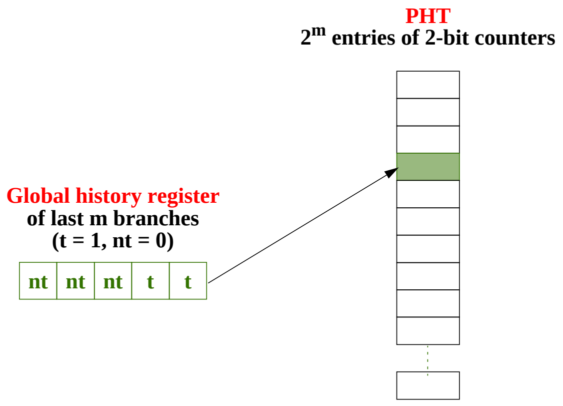
- if branches 1 & 2 are taken, branch 3 is not taken; 2-bit saturating counter cannot predict this behavior

⇒ use a **history of the past m branches**
represents a path through the program
(but still n bits of prediction)

1 Improve the Prediction

General idea of correlated branch prediction:

- use the global branch history in a **global history register**
    - global history is a **shift register**: shift left in the new branch outcome
- to access a **pattern history table (PHT)** of 2-bit saturating counters



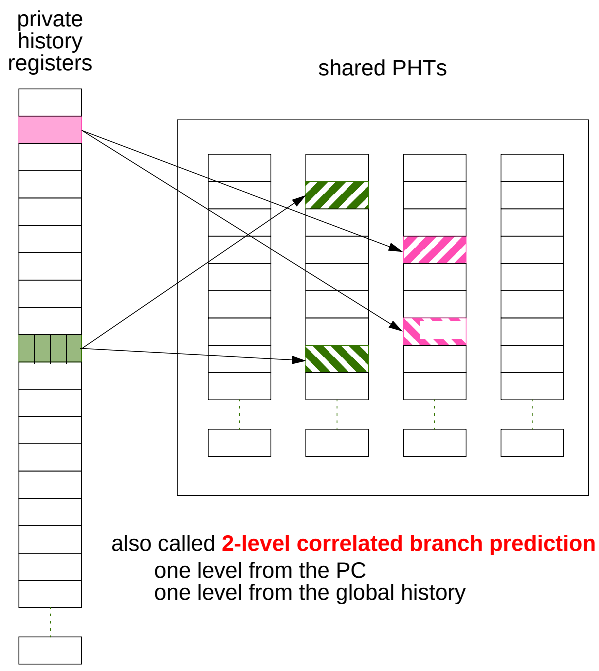
1 Improve the Prediction

Many implementation variations

- number of history registers
    - 1 history register for all branches (global)
    - table of history registers, 1 for each branch (private)
    - table of history registers, each shared by several branches (shared)
- history length (size of history registers)
- number of PHTs
- **What is the trade-off?**

1 Improve the Prediction

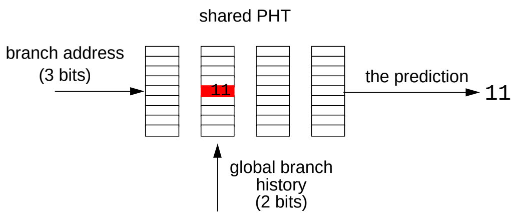
Private history tables, 4 bits of history, shared PHTs



also called **2-level correlated branch prediction**
one level from the PC
one level from the global history

1 Improve the Prediction

Organization in the book:



- really a linear buffer, accessed by concatenating the global history with the low-order bits from the PC
- current implementations XOR branch address & global history bits
  - called **gshare**
  - more accuracy with same bits or equivalent accuracy with fewer bits

1 Improve the Prediction

Predictor classification (the book's simple version)

- (m,n) predictors
  - m = history bits, number of branches in the history
  - n = prediction bits
- (0,1) = 1-bit branch prediction buffer
- (0,2) = 2-bit branch prediction buffer
- (5,2) = first picture
- (4,2) = second picture
- (2,2) = book picture
- (4,2) = Pentium Pro scheme

1 Improve the Prediction

Combining branch predictors

- local, per-branch prediction, accessed by the PC
- global prediction based on the last *m* branches, assessed by the global history
- indicator of which had been the best predictor for this branch
  - 2-bit counter: increase for one, decrease for the other
- Compaq Alpha 21264
  - ~5% misprediction on SPEC95
  - 2% of die

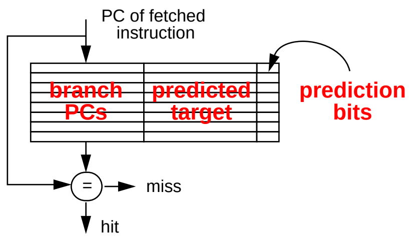
Confidence predictors

- an indication of how confident you can be of the prediction
  - if very confident, then follow the prediction
  - if not confident, then stall
- implementation:
  - a counter which is increased when a prediction is correct and cleared when it is wrong
  - the higher the value, the more confident you can be of the prediction
  - pick a threshold value for predictors you believe

2 Determine the Target Earlier

Branch target buffer (BTB)

- cache that stores: the PCs of branches *(tag)*
  the predicted target address *(data)*
  optional branch prediction bits *(data)*
  (~ BPB + target address + tag)
- accessed by PC address in fetch stage
  if hit: address was for *this* branch instruction
  fetch the target instruction if prediction bits say taken



- **no** branch delay if: branch found in BTB
  prediction is correct
  (assume BTB update is done in the next cycles)

2 Determine the Target Earlier

Return address stack

- the bad news:
  - indirect jumps are hard to predict
  - registers are accessed several stages after fetch
- the good news: most indirect jumps (85%) are returns
  - optimize for the common case
- small stack: return address pushed on a call, popped on a return
- best for procedures that are called from multiple call sites
  - BTB would predict address of the return from the last call
- if big enough, can predict returns perfectly
  these days 1-32 entries

3 Reduce the Misprediction Penalty

Fetch both instruction streams

- requires a dual-ported instruction cache
- requires independent bank accessing

3 Reduce the Misprediction Penalty

Resume buffer

- technique used when you don’t know the target address until the end of decoding, so there is a one-cycle bubble in the pipeline if the prediction is taken
- save the sequential instructions that were fetched in that cycle in a buffer
  - if the taken prediction is wrong, reload these instructions from the resume cache
- shaves a cycle off the misprediction penalty