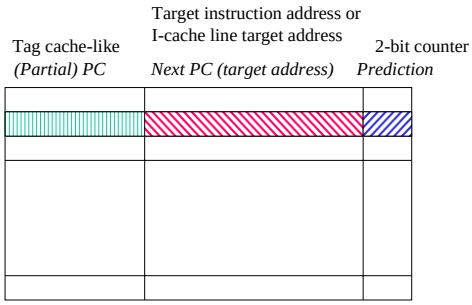
Branch Target Buffers

- BPB: Tag + Prediction
- BTB: Tag + prediction + next address
- Now we predict and "precompute" branch outcome and target address during IF
  - Of course more costly
  - Can still be associated with cache line (UltraSparc)
  - Implemented in a straightforward way in Pentium; not so straightforward in Pentium Pro (see later)
  - Decoupling (see later) of BPB and BTB in Power PC and PA-8000
  - Entries put in BTB only on taken branches (small benefit)

BTB layout



During IF, check if there is a hit in the BTB. If so, the instruction must be a branch and we can get the target address – if predicted taken – during IF. If correct, no stall

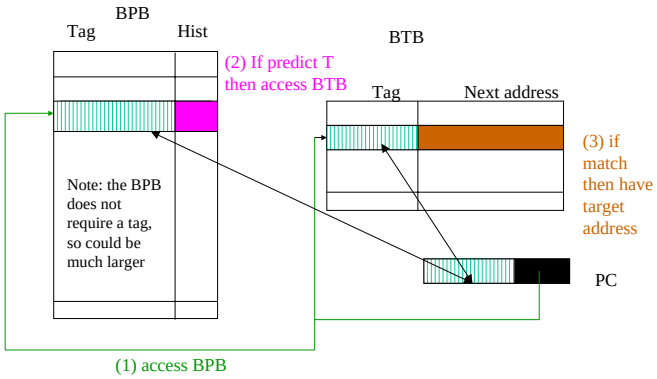
Another Form of Misprediction in BTB

- Correct "Taken" prediction but incorrect target address
- Can happen for "return" (but see later)
- Can happen for "indirect jumps" (rare but costly)
  - Might become more frequent in object-oriented programming a la C++, Java

Decoupled BPB and BTB

- For a fixed real estate (i.e., fixed area on the chip):
  - Increasing the number of entries implies less bits for history or no field for target instruction or fewer bits for tags (more aliasing)
  - Increasing the number of entries implies better accuracy of prediction.
- Decoupled design
  - Separate – and different sizes – BPB and BTB
  - BPB. If it predicts *taken* then go to BTB (see next slide)
  - Power PC 620: 2K entries BPB + 256 entries BTB
  - HP PA-8000: 256*3 BPB + 32 (fully-associative) BTB

Decoupled BTB



Correlated or 2-level branch prediction

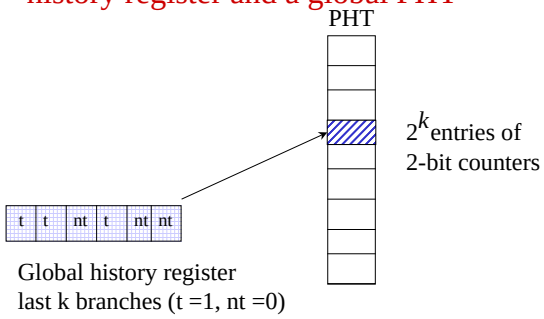
- Outcomes of consecutive branches are not independent
- Classical example

```
loop
  ….
   if ( x = = 2)                    /* branch b1 */
              x = 0;
   if ( y = = 2)                    /* branch b2 */
              y = 0;
   if ( x != y)                     /* branch b3 */
              do this
              else do that
```

What should a good predictor do?

- In previous example if both b1 and b2 are Taken, b3 should be Not-Taken
- A two-bit counter scheme cannot predict this behavior.
- Needs history of previous branches hence correlated schemes for BPB's
  - Requires history of *n* previous branches (shift register)
  - Use of this vector (maybe more than one) to index a Pattern History Table (PHT) (maybe more than one)

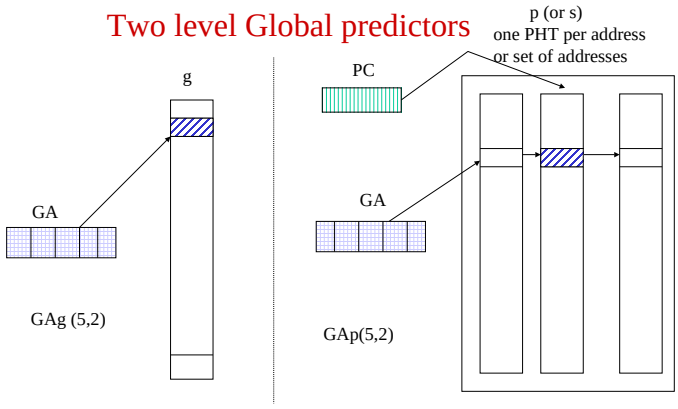
General idea: implementation using a global history register and a global PHT



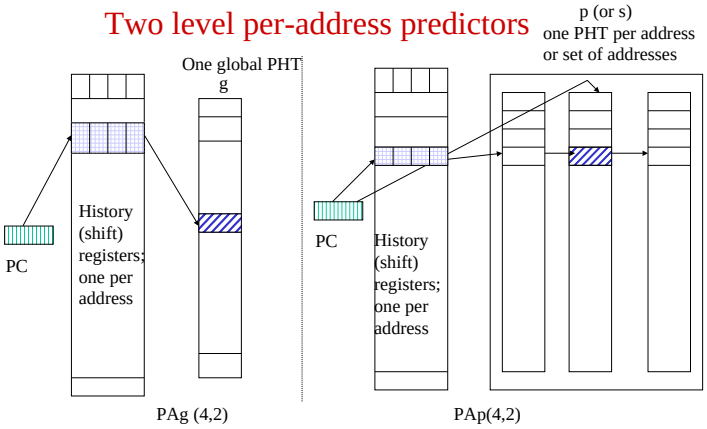
Classification of 2-level (correlated) branch predictors

- How many global registers and their length:
  - GA: Global (one)
  - PA: One per branch address (Local)
  - SA: Group several branch addresses
- How many PHT's:
  - g: Global (one)
  - p : One per branch address
  - s: Group several branch addresses
- Previous slide was GAg (6,2)
  - The "6" refers to the length of the global register
  - The "2" means we are using 2-bit counters

Two level Global predictors

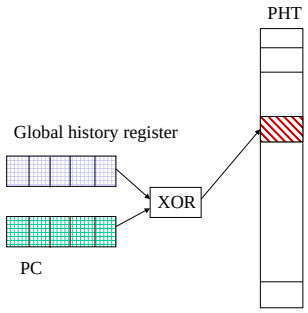


Two level per-address predictors



Gshare: a popular predictor

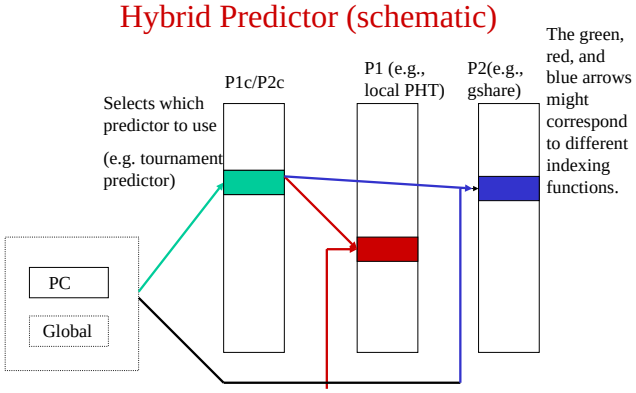
(the one simulated in Bliss)



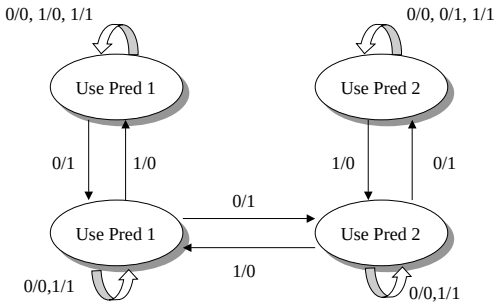
The Global history register and selected bits of the PC are XORed to provide the index in a single PHT

The idea is to try and avoid aliasing, i.e. avoid interference for two different branches with the same pattern

Hybrid Predictor (schematic)



Tournament Predictor



0: pred is incorrect; 1 pred is correct;
a/b pred for Pred 1 / Pred 2

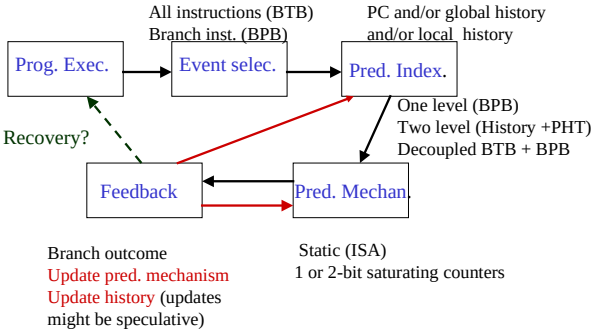
Evaluation

- The more hardware (real estate) the better!
  - GA s for a given number of "s" the larger G the better; for a given "G" length, the larger the number of "s" the better.
- Note that the result of a branch might not be known when the GA (or PA) needs to be used again (because we might issue several instructions per cycle). It must be speculatively updated (and corrected if need be).
- Ditto for PHT but less in a hurry?

Performance

- Hybrid predictor consisting of a local predictor of size *s1* and a global predictor of size *s2* seems to perform better than a local or global predictor of size $s > s1 + s2$
- Use machine learning (AI) techniques?
  - Start with a "quick and dirty" predictor yielding a prediction in one cycle
  - Concurrently use a slower, more accurate predictor. If its prediction disagrees with the fast predictor, roll back the computation.

Summary: Anatomy of a Branch Predictor



Return jump stack

- Indirect jumps difficult to predict except returns from procedures (but luckily returns are about 85% of indirect jumps)
- If returns are entered with their target address in BTB, most of the time it will be the wrong target
  - Procedures are called from many different locations
- Hence addition of a small "return stack"; 4 to 8 entries are enough (1 in MIPS R10000, 4 in Alpha 21064, 4 in Sparc64, 12 in Alpha 21164)
  - Checked during IF, in parallel with BTB.

Resume buffer

- In some “old” machines (e.g., IBM 360/91 circa 1967), branch prediction was implemented by fetching both paths (limited to 1 branch)
- Similar idea: “resume buffer” in MIPS R10000.
  - If branch predicted taken, it takes one cycle to compute and fetch the target
  - During that cycle save the Not-Taken sequential instruction in a buffer (4 entries of 4 instructions each).
  - If mispredict, reload from the “resume buffer” thus saving one cycle