18-447 Lecture 10: Branch Prediction

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
  - apply the locality principle to predict what is on the midterm
  - understand how to guess your way through control flow and why it works so well

- Notices
  - Lab 2, status check this week, due next week
  - HW2, due Wed
  - Midterm, next Monday

- Readings
  - P&H Ch 4
The Locality Principle

- **One’s recent** past is a very good predictor of one’s **near** future.
- **Temporal Locality:** If you just did something, it is very likely that you will do the same thing again **soon**
  - since you are here today, there is a good chance you will be here again and again regularly
  - inverse is also true
- **Spatial Locality:** If you just did something, it is very likely you will do something **similar/related**
  - if you are in this room, I expect to find you in the same seat (or nearby)
  - I expect to find your class buddies also in this room
  - Programs are even more predictable than people
  - 18-447 midterms exhibit strong locality too

Control Speculation 101: PC+4

VERY IMPORTANT: as long as
1. prediction is always checked
2. correct target is fetched after a misprediction,

**ANY** predictor will work, including RNG, PC-4, etc.

Inst$_h$, Inst$_i$, Inst$_j$, Inst$_k$, Inst$_l$

Inst$_l$ is a taken branch

\[ IPC = \frac{1}{1 + (0.20 \times 0.7) \times 2} \]
\[ = 0.78 \]
Branch Resolution in a Pipeline

- “Trust, but verify”
- Update branch predictor to improve next guess

Speculative Execution Summary

- Each control flow instruction must carry the predicted nextPC down the pipeline
- As soon as the control flow outcome of an instruction is certain, the predicted nextPC is checked
- If nextPC was predicted correctly
  - update BHT (reinforce prediction)
  - do nothing more
- If nextPC was predicted incorrectly
  - update BHT and/or BTB
  - flush all younger instructions in the pipeline
  - restart fetching at the correct target

Relatively easy in in-order pipelines, but not so easy in deeply pipelined, superscalar, out-of-order processors (multiple nested predictions, many 10s of inst’s later)
Branch Target Buffer (magic version)

- BTB
  - a giant table indexed by PC
  - returns the “guess” for nextPC
- When encountering a PC for the first time, after decoding, record in BTB . . .
  - PC + 4 if ALU/LD/ST
  - PC+offset if Branch or Jump
  - ?? if JR
- Effectively guessing branches are always taken (and where to)

\[ \text{IPC} = \frac{1}{1 + (0.20 \times 0.3) \times 2} = 0.89 \]

BTB (Reality)

- “Hash” PC into a 2^N entry table
- On collision, BTB returns a guess for some unrelated PC

How big should this table be?
Tagged BTB

Only store branch instructions (save 80% storage)
Update tag and BTB for new branch after collision (good strategy?)

Even Better Guess

- Already 100% correct on non-branch instructions
- Can we do better than 70% on branch instructions?
  - ~90% correct on backward branch (dynamic)
  - only ~50% correct on forward branch (dynamic)
  What pattern to leverage on forward branches?

- A given static branch instruction is likely to be biased in one direction (either taken or not taken)
  - 80~90% correct if we always guessed the same outcome as the last time the same branch was executed
  - IPC = 1 / [ 1 + (0.20*0.15) * 2 ] = 0.94
The 1-bit BHT entry is updated with the actual outcome after each branch is executed.

Branch History State Machine

- predict not taken
- predict taken
- actually taken
- actually not taken
2-Bit Saturation Counter

2-Bit “Hysteresis” Counter

Change prediction after 2 consecutive mistakes
Per-Branch History-Based Predictors

- 2-bit predictor can get >90% correct
  - $\text{IPC} = \frac{1}{[1 + (0.20 \times 0.10) \times 2]} = 0.96$
  - any “reasonable” 2-bit predictor does about the same
- Adding more bits to counters does not help much more
- Major branch behaviors exploited
  - almost always do the same thing again and again (>80%)
    - 1-bit and 2-bit predictors equally effective
  - occasionally do the opposite once (5~10%)
    - 2 misprediction with a 1-bit predictor
    - 1 misprediction with a 2-bit predictor
  - miscellaneous (<10%)
    - some could be captured with more elaborate predictors
    - what does Amdahl’s law say about this? (be careful!!)

Path History

- Branch outcome can be correlated to other branches
- Equantott, SPEC92
  
  ```
  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

How do you capture this information?
Gshare Branch Prediction [McFarling]

<table>
<thead>
<tr>
<th>tag</th>
<th>BTB idx</th>
</tr>
</thead>
</table>

- N-bit
- M-bit
- BHSR

Global BHSR (Branch History Shift Register) tracks the outcomes of the last M branch instructions

Return Address Stack

- The targets of register-indirect jumps have weak locality
  - history-based predictors don’t work
  - but a simple “stack” captures the usage pattern of function call and return very well
- Return Address Stack (RAS)
  - the return address is pushed when a link instruction (e.g., JAL) is executed
  - when the PC of a return instruction (e.g., JR) is encountered predict nPC from the top of the stack and pop

What happens when the stack overflows?
How do you know when to follow RAS vs BTB?
Alpha 21264 Tournament Predictor

- Make separate predictions using local history (per branch) and global history (correlating all branches) to capture different branch behaviors
- A meta-predictor decides which predictor to believe

Better than 97% correct

Multiple Predictors: PPC 604

[Diagram showing the flow of instructions through the fetch, decode, dispatch, branch, execute, and complete stages with prediction logic for each stage]
Superscalar Complications

- “Superscalar” processors
  - attempts to execute more than 1 instruction-per-cycle
  - must fetch multiple instructions per cycle
- 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 youngest predicted taken
    - If the 1st instruction is the predicted taken branch
      → nullify 2nd instruction fetched

2-way Branch Predictor Example

Note on BHT: need 1 extra bit in addition to history bits to disambiguate whether predicted-taken branch was in primary or secondary position
Trace Caching

- Static 90% dynamic 10%
- 10% static 90% dynamic

Trace-cache line boundaries

I-cache line boundaries

Intel P4 Trace Cache

- A 12K-uop trace cache replaces the L1 I-cache
- 6-uop per trace line, can include branches
- Trace cache returns 3-uop per cycle
- IA-32 decoder can be simpler and slower
Involving SW in Branch Prediction

- Static branch “hints” can be associated with opcodes
  - taken vs. not-taken
  - whether to allocate an entry in the dynamic BP hardware
- SW and HW has joint control of BP hardware
  - Intel Itanium “brp” (branch prediction) instruction can be issued ahead of the actual branch to preset the BTB state
- TAR (Target Address Register, Itanium)
  - a small, fully-associative BTB
  - controlled entirely by “prepare-to-branch” instructions
  - a hit in TAR overrides all other predictions

Why wait until the last instruction in the basic block to calculate branch condition and target?

Predicated Execution: If-conversion

- Example: predication in Intel Itanium
  - 64 one-bit predicate registers
    - each instruction carries a 6-bit predicate field
  - each instruction can be separately predicated
  - an instruction is effectively a NOP if its predicate is false
- Converts control flow into dataflow

Make sense if processors have lots of spare resources and BP is hard
Branch Prediction: the bottom-line

- Given current PC, how to determine the next PC waiting for anymore information would need stalls
- The easy part
  - the same PC always points to the same instruction (barring self-modifying code)
  - nextPC is always PC+4 for non-control-flow instructions,
  - the target of a PC-offset control-flow is always the same
    A memoization table can get these nearly 100% right
- The not so easy part
  - taken versus not-taken decision is not static
    - 90% of backward branches are taken (loops)
    - 50% of forward branches are taken (if-then-else)
  - a given branch “almost” “always” repeats itself