18-447 Lecture 10: Branch Prediction

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today
  – understand how to guess your way through control flow and why it works so well

• Notices
  – HW 2, past due
  – Lab 2, status check this week, due next week
  – HW 3, due Mon 2/28
  – Midterm 1, Wed 3/2, covers up to today L10

• Readings
  – P&H Ch 4
Branch Prediction 101: PC+4

In general as long as:
1. prediction is always checked
2. correct target is fetched after a misprediction
3. wrong path instructions removed

**ANY** predictor will work, including RNG, PC-4
Prediction and Resolution in General

- “Trust (1), but verify (2)”
- When wrong, (3) clean up mistake and (4) update predictor to improve next guess

```
nextPC

“ANY” branch predictor

pred. taken? pred. target

fetched PC Inst

actual target

update

mispredict?

compute actual outcome

rewind

1 3 3

2

??

kill kill kill

4

update

1

pred. taken?

```
Tagged BTB (from last lecture)

Only add control-flow inst to BTB; non-control-flow always miss, always PC+4

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

~30% not taken
Sum Up So Far

• Given current PC, speculate most likely next PC

• The easy part: target
  – same PC always same instruction
  – nextPC always PC+4 for non-control-flow inst
  – target of PC-offset control-flow always same

  BTB from last slide works very well

• The not so easy part: taken?
  – branch decision is dynamically data dependent
  – so far, either 1. always-predict-not-taken (PC+4) or
    2. always-predict-taken (BTB)
Branch Direction Prediction

• Already 100% correct on non-control-flow inst
• Improve on always-predict-taken (70% correct)?
  – ~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 (forward+backward) if guessed to repeat the outcome last time
  – IPC = \( 1 / \left[ 1 + (0.20 \times 0.15) \times 2 \right] = 0.94 \)

if not repeat
“Adaptive” History-Based Prediction

Branch History Table entry (1 bit) is updated with actual outcome after branch is executed

IPC = 1 / [ 1 + (0.20*0.15) * 2 ] = 0.94 if not repeat
Branch History State Machine

Predict same as last outcome
2-Bit Saturation Counter

![Diagram of 2-bit saturation counter with states labeled as follows:
- pred taken 11
- pred !taken 01
- pred taken 10
- pred !taken 00
- actually taken
- actually !taken
- "weakly taken"
- "strongly taken"

The diagram shows transitions between states based on predicted and actual taken status.

How is this better?]
2-Bit “Hysteresis” Counter

Change prediction after 2 consecutive mistakes
Per-Branch Counter-Based BP

- 2-bit counter can get >90% correct
  - IPC = \( \frac{1}{1 + (0.20 \times 0.10) \times 2} \) = 0.96
  - any “reasonable” 2-bit counter works
  - adding more bits to counter does not help much

- Major branch behaviors exploited
  - almost always repeat the same (>80%)
    - 1-bit and 2-bit counters equally effective
    - occasionally do the opposite once (5~10%)
    - 2 misprediction with a 1-bit counter
    - 1 misprediction with a 2-bit counter

- Need more elaborate predictors for other behaviors

Is it worth the cost? Will it slow down the clock?
The cost of misprediction

- Misprediction penalty increases with
  - number of pipeline stages
  - width of superscalarity
  - number of nested predictions and rewind cost

["The microarchitecture of the Pentium 4 processor," Intel Technology Journal, 2001.]
Multiple shots at better predictions

- more time & info in later stages
- early “correction” based on better guesses

fetch
- instruction cache
- BHT
- BTAC
- $+2$
- $+4$
- FAR

decode
- Prediction Logic (4 instructions)
- Target
- Seq Addr

dispatch
- Prediction Logic (4 instructions)
- Target
- Seq Addr

branch
- Prediction Logic (4 instructions)
- Target
- Seq Addr

execute
- Target
- Exception Logic

complete
- PC
- $+$

[PowerPC 604]
Two-level Prediction [Yeh & Patt]

- **tag**
- **BHT idx**

- **m-bit “local” branch history**
  - e.g., if $m=6$
    - 000000
    - 111111
    - 111110
    - 000001
    - 101010
    - 010101
    - 110001

- **isBranch?**
- **taken?**

what a branch did last $m$ times

what happened for a pattern? (adaptive)
Path History

• Branch outcome may be correlated to other branches

• Equntott, 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 to capture this information?
Gshare Branch Prediction [McFarling]

Global Branch History Shift Register tracks the outcomes of the last M branch instructions (dynamic)
Return Address Stack

• A register-indirect jump can have different target
  – same target only if fxn called repeatedly from same call-site
  – but, function call and return behavior easily tracked by a last-in-first-out queue

• Return Address Stack
  – return address is pushed when a link instruction (i.e., JAL x1...) is executed
  – when encountering PC of a return instruction (i.e., JALR ...x1) predict nPC from top of stack and pop

  What happens when the stack overflows?
  How do you know when to follow RAS vs BTB?
• 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
Superscalar Complications

• “Superscalar” processors need to fetch multiple instructions per cycle

• Consider 2-way superscalar fetch scenario
  
  (case 1) both instructions are not taken control-flow
  – \( nPC = PC + 8 \)

  (case 2) one inst is a **taken** control-flow inst
  – \( nPC = \) predicted target addr
    
    note: both instructions could be control-flow;
    target is for younger of predicted taken
  – if 1\(^{st}\) instruction is predicted taken, nullify 2\(^{nd}\) instruction fetched
2-way Branch Predictor Sketch

- Tag Table
- Branch History Table (BHT)
- Branch Target Buffer (BTB)

- last inst in cache block?
- PC+4
- PC+8
- predPC
- first?
- taken?
- hit
- cache block offset
- taken?
Trace Caching

- Static 90%
- Dynamic 10%

Compiler static

Hardware dynamic

i-cache block boundaries

trace cache block boundaries
Intel P4 Trace Cache

• A 12K-uop trace cache in place of L1 I-cache
• 6-uop per trace block, can include branches
• Trace cache returns 3-uop per cycle
• IA-32 decoder can be simpler and slower <<<
Ways SW can Help

• Associate static branch “hints” with opcodes
  – taken vs. not-taken
  – whether to allocate entry in dynamic BP hardware

• Give SW and HW joint control of BP hardware
  – Intel Itanium BRP (branch prediction) instruction
    issued ahead of branch to preset BTB state

• TAR (Target Address Register, Itanium)
  – a small, fully-associative BTB
  – controlled entirely by BRP instructions
  – a hit in TAR overrides all other predictors

Relieves “urgency” by not wait to compute branch condition and target as last inst in basic block
Predicated Execution

- Intel Itanium example
  - predicate register file (64 by 1-bit)
  - each instruction has a predicate reg argument
  - instruction is NOP if predicate is false at runtime
- Converting control flow into dataflow

Make sense if processors have lots of spare resources and BP is hard
Interrupt Control Transfer

- **Basic Part:** an “unplanned” fxn call to a “third-party” routine; and later return control back to point of interruption
- **Tricky Part:** interrupted thread cannot anticipate/prepare for this control transfer
  - must be 100% transparent
  - not enough to impose all callee-save convention
- **Puzzling Part:** why is there a hidden routine running invisibly?