Branch Prediction

Joel Emer
Computer Science and Artificial Intelligence Laboratory
M.I.T.

http://www.csg.csail.mit.edu/6.823
Phases of Instruction Execution

Fetch: Instruction bits retrieved from cache.

Decode: Instructions placed in appropriate issue (aka “dispatch”) stage buffer

Execute: Instructions and operands sent to execution units. When execution completes, all results and exception flags are available.

Commit: Instruction irrevocably updates architectural state (aka “graduation” or “completion”).
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!

How much work is lost if pipeline doesn’t follow correct instruction flow?

~ Loop length x pipeline width
Average Run-Length between Branches

Average dynamic instruction mix from SPEC92:

<table>
<thead>
<tr>
<th></th>
<th>SPECint92</th>
<th>SPECfp92</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>39 %</td>
<td>13 %</td>
</tr>
<tr>
<td>FPU Add</td>
<td></td>
<td>20 %</td>
</tr>
<tr>
<td>FPU Mult</td>
<td></td>
<td>13 %</td>
</tr>
<tr>
<td>load</td>
<td>26 %</td>
<td>23 %</td>
</tr>
<tr>
<td>store</td>
<td>9 %</td>
<td>9 %</td>
</tr>
<tr>
<td>branch</td>
<td>16 %</td>
<td>8 %</td>
</tr>
<tr>
<td>other</td>
<td>10 %</td>
<td>12 %</td>
</tr>
</tbody>
</table>

SPECint92:  
compress, eqntott, espresso, gcc, li

SPECfp92:  
doduc, ear, hydro2d, mdijdp2, su2cor

What is the average run length between branches

Roughly 10 instructions
# MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JR</td>
<td>After Inst. Decode</td>
<td>After Reg. Fetch</td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td>After Reg. Fetch*</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>

*Assuming zero detect on register read
Example Branch Penalties

UltraSPARC-III instruction fetch pipeline stages (in-order issue, 4-way superscalar, 750MHz, 2000)

- A: PC Generation/Mux
- P: Instruction Fetch Stage 1
- F: Instruction Fetch Stage 2
- B: Branch Address Calc/Begin Decode
- I: Complete Decode
- J: Steer Instructions to Functional units
- R: Register File Read
- E: Integer Execute
- ... Remainder of execute pipeline (+ another 6 stages)
Reducing Control Flow Penalty

Software solutions

- *Eliminate branches - loop unrolling*
  Increases the run length
- *Reduce resolution time - instruction scheduling*
  Compute the branch condition as early as possible (of limited value)

Hardware solutions

- Bypass – usually results are used immediately
- Change architecture - find something else to do
  - *delay slots* - replace pipeline bubbles with useful work (requires software cooperation)
- *Speculate - branch prediction*
  *Speculative execution* of instructions beyond the branch
Branch Prediction

Motivation:
Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:
Prediction structures:
- Branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:
- Keep result computation separate from commit
- Kill instructions following branch in pipeline
- Restore state to state following branch
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
  bne0 (*preferred taken*)  beq0 (*not taken*)

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
  typically reported as ~80% accurate
Dynamic Prediction

Prediction as a feedback control process

Operations
- Predict
- Update
Predictor Primitive
Emer & Gloy, 1997

- Indexed table holding values

- Operations
  - Predict
  - Update

- Algebraic notation

\[ \text{Prediction} = P[\text{Width}, \text{Depth}](\text{Index}; \text{Update}) \]
Dynamic Branch Prediction
learning based on past behavior

Temporal correlation
The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Spatial correlation
Several branches may resolve in a highly correlated manner (a preferred path of execution)
One-bit Predictor

Simple temporal prediction

\[ A_{21064}(PC; T) = P[1, 2K](PC; T) \]

What happens on loop branches?

At best, mispredicts twice for every use of loop.
Branch Prediction Bits

- Assume 2 BP bits per instruction
- Use saturating counter

<table>
<thead>
<tr>
<th>On non-taken</th>
<th>On taken</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Strongly taken</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Weakly taken</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Weakly non-taken</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Strongly non-taken</td>
</tr>
</tbody>
</table>
Two-bit Predictor

*Smith, 1981*

Counter[\(W,D\)](I; \(T\)) = \(P[W, D](I; \text{if } T \text{ then } P+1 \text{ else } P-1)\)

\(A21164(\text{PC}; T) = \text{MSB(Counter}[2, 2K](\text{PC}; T))\)
Branch History Table

4K-entry BHT, 2 bits/entry, ~80-90% correct predictions
Exploiting Spatial Correlation
Yeh and Patt, 1992

if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;

If first condition false, second condition also false

*History register*, H, records the direction of the last N branches executed by the processor
History Register

\[
\text{History}(\text{PC}; \ T) = \text{P}(\text{PC}; \ P \ || \ T)
\]
Global History

\[
\text{GHist}(;T) = \text{MSB}(\text{Counter}((\text{History}(0, T); T))
\]

\[
\text{Ind-Ghist}(PC;T) = \text{MSB}(\text{Counter}(PC || \text{Hist}((\text{GHist}(;T);T)))
\]

Can we take advantage of a pattern at a particular PC?
Local History

LHist(PC; T) = MSB(Counter(History(PC; T); T))

Can we take advantage of the global pattern at a particular PC?
Two-level Predictor

\[ 2\text{Level}(\text{PC}; T) = \text{MSB}(\text{Counter}(\text{History}(0; T) \| \text{PC}; T)) \]
Two-Level Branch Predictor

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)

Fetch PC

2-bit global branch history shift register

Shift in Taken/¬Taken results of each branch

Taken/¬Taken?
Choosing Predictors

Chooser = MSB(P(PC; P + (A==T) - (B==T)))

or

Chooser = MSB(P(GHist(PC; T); P + (A==T) - (B==T)))
Tournament Branch Predictor
(Alpha 21264)

- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications

Local history table (1,024x10b)
Local prediction (1,024x3b)
Global Prediction (4,096x2b)
Choice Prediction (4,096x2b)
Global History (12b)
TAGE predictor
Seznec & Michaud, 2006

\[ \text{TAGE TREE}[L1, L2, L3](PC; T) = \text{TAGE}[L3](PC, \text{TAGE}[L2](PC, \text{TAGE}[L1](PC, \text{Bimodal}(PC;T); T); T); T) \]
TAGE component

Next Predictor

Counter

Useful

Tag

Prediction

My Guess

Use me?

GHist

PC

March 13, 2017
http://www.csg.csail.mit.edu/6.823
TAGE predictor component

TAGE[L](PC, NEXT; T) =

\[ \text{idx} = \text{hash}(PC, \text{GHIST}[L](;T)) \]
\[ \text{tag} = \text{hash}(PC, \text{GHIST}[L](;T)) \]

TAGE.U = SA(idx, tag; ((TAGE == T) && (NEXT != T))?1:SA)
TAGE.Counter = SA(idx, tag; T?SA+1:SA-1)

use_me = TAGE.U && isStrong(TAGE.Counter)
TAGE = use_me?MSB(TAGE.Counter):NEXT

Notes:
SA is a ‘set associative’ structure
SA allocation occurs on mispredict (not shown)
TAGE.U cleared on global counter saturation
Limitations of branch predictors

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.

UltraSPARC-III fetch pipeline

Correctly predicted taken branch penalty

A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

Jump Register penalty

Remainder of execute pipeline (+ another 6 stages)
Branch Target Buffer (untagged)

BP bits are stored with the predicted target address.

IF stage: \( \text{If (BP=take) then nPC=target else nPC=PC+4} \)

later: \( \text{check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb} \)
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?

BTB prediction = 236
Correct target = 1032

⇒ kill PC=236 and fetch PC=1032

Is this a common occurrence?
Can we avoid these bubbles?
BTB is only for Control Instructions

BTB contains useful information for branch and jump instructions only
   ⇒ Do not update it for other instructions

For all other instructions the next PC is (PC)+4!

*How to achieve this effect without decoding the instruction?*
Branch Target Buffer (tagged)

- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded
Consulting BTB Before Decoding

- The match for PC=1028 fails and 1028+4 is fetched
  ⇒ eliminates false predictions after ALU instructions

- BTB contains entries only for control transfer instructions
  ⇒ more room to store branch targets
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR).
- BHT can hold many more entries and is more accurate.

BHT in later pipeline stage corrects when BTB misses a predicted taken branch.

BTB/BHT only updated after branch resolves in E stage.
Line Prediction
(Alpha 21[234]64)

- For superscalar useful to predict next cache line(s) to fetch

- Line Predictor predicts line to fetch each cycle (tight loop)
  - Untagged BTB structure – Why?
  - 21464 was to predict 2 lines per cycle

- Icache fetches block, and predictors improve target prediction

- PC Calc checks accuracy of line prediction(s)
Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)
  
  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)
  
  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)
  
  BTB works well if usually return to the same place
  
  ⇒ Often one function called from many distinct call sites!

How well does BTB work for each of these cases?
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```plaintext
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

Push call address when function call executed

Pop return address when subroutine return decoded

k entries (typically k=8-16)
Overview of branch prediction

- Need next PC immediately
- Tight loop
- Must speculation check always be correct? No...

**Overview:**

1. **PC**
   - Need next PC immediately
   - Tight loop
   - Must speculation check always be correct? No...

2. **BTB**
   - Instr type, PC relative targets available
   - Loose loop

3. **Decode**
   - Simple conditions, register targets available
   - Loose loop

4. **Reg Read**
   - Complex conditions available
   - Loose loop

5. **Execute**
   - Best predictors reflect program behavior
Thank you!