Branch Prediction

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Reminder: 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”).

The diagram illustrates the flow of instruction execution phases:

1. **PC** (Program Counter)
2. **I-cache** (Instruction Cache)
3. **Fetch Buffer**
4. **Issue Buffer**
5. **Func. Units** (Function Units)
6. **Results Buffer**
7. **Arch. State** (Architectural State)
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?
Average Run-Length between Branches

Average dynamic instruction mix of SPEC CPU 2017 [Limaye and Adegbiya, ISPASS’18]:

<table>
<thead>
<tr>
<th></th>
<th>SPECInt</th>
<th>SPECfp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branches</td>
<td>19 %</td>
<td>11 %</td>
</tr>
<tr>
<td>Loads</td>
<td>24 %</td>
<td>26 %</td>
</tr>
<tr>
<td>Stores</td>
<td>10 %</td>
<td>7 %</td>
</tr>
<tr>
<td>Other</td>
<td>47 %</td>
<td>56 %</td>
</tr>
</tbody>
</table>

SPECInt17: perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, exchange2, xz
SPECfp17: bwaves, cactus, lbm, wrf, pop2, imagick, nab, fotonik3d, roms

What is the average run length between branches?
# 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></td>
<td></td>
</tr>
<tr>
<td>JR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td></td>
<td></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)

Branch
Target
Address
Known

Branch
Direction &
Jump

Register
Target
Known

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 run length between branches
  – 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

Input ➔ Predictor ➔ Prediction ➔ Truth/Feedback
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*)
Predictor Primitive
Emer & Gloy, 1997

- Indexed table holding values

- Operations
  - Predict
  - Update

- Algebraic notation

\[ \text{Prediction} = P[\text{Width}, \text{Depth}](\text{Index}; \text{Update}) \]
One-bit Predictor
aka Branch History Table (BHT)

Simple temporal prediction

1 bit

PC ➔

Prediction

A21064(PC; T) = P[ 1, 2K ](PC; T)

What happens on loop branches?
Two-bit Predictor

Smith, 1981

- Use two bits per entry instead of one bit
- Manage them as a saturating counter:

<table>
<thead>
<tr>
<th>On not-taken</th>
<th>On taken</th>
<th>Status</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 not-taken</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Strongly not-taken</td>
</tr>
</tbody>
</table>

- Direction prediction changes only after two wrong predictions

How many mispredictions per loop? ___
Two-bit Predictor

Smith, 1981

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

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

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

if (\text{x[i]} < 7) \text{ then}
\quad y += 1;
if (\text{x[i]} < 5) \text{ then}
\quad c -= 4;

If first condition false, second condition also false

*History register* records the direction of the last N branches executed by the processor
History Registers
aka Pattern History Table (PHT)

History(PC; T) = P(PC; P || T)
Global-History Predictor

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

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

Can we take advantage of the global pattern at a particular PC?

\[ \text{LHist}(\text{PC}; T) = \text{MSB}(\text{Counter}(\text{History}(\text{PC}; T); T)) \]
Global-History Predictor with Per-PC Counters

\[ \text{GHistPA}(PC; T) = \text{MSB} (\text{Counter}(\text{History}(0; T)||PC; T)) \]
Two-Level Branch Predictor
(Pentium Pro, 1995)

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

\[
\text{Chooser} = \text{MSB}(P(\text{PC}; P + (A==T) - (B==T)))
\]

or

\[
\text{Chooser} = \text{MSB}(P(\text{GHist(PC}; T); P + (A==T) - (B==T)))
\]
Tournament Branch Predictor
(Alpha 21264, 1996)

- 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
TAGE predictor
Seznec & Michaud, 2006

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

\[
TAGE[L](PC, \text{NEXT}; T) = \\
\text{idx} = \text{hash}(PC, \text{GHIST}[L](; T)) \\
\text{tag} = \text{hash'}(PC, \text{GHIST}[L](; T))
\]

\[
\text{TAGE.U} = \text{SA(idx, tag; ((TAGE == T) \&\& (NEXT != T))?1:SA)} \\
\text{TAGE.Counter} = \text{SA(idx, tag; T?SA+1:SA-1)}
\]

\[
\text{use_me} = \text{TAGE.U} \&\& \text{isStrong(TAGE.Counter)} \\
\text{TAGE} = \text{use_me?MSB(TAGE.Counter):NEXT}
\]

Notes:
\begin{itemize}
\item SA is a set-associative structure
\item SA allocation occurs on mispredict (not shown)
\item TAGE.U cleared on global counter saturation
\end{itemize}
Limitations of branch predictors

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

Correctly predicted taken branch penalty

Jump Register penalty

UltraSPARC-III fetch pipeline

PC Generation/Mux
Instruction Fetch Stage 1
Instruction Fetch Stage 2
Branch Address Calc/Begin Decode
Complete Decode
Steer Instructions to Functional units
Register File Read
Integer Execute
Remainder of execute pipeline (+ another 6 stages)
BP bits are stored with the predicted target address.

**IF stage:** If \((BP=\text{taken})\) then \(nPC=\text{target}\) else \(nPC=PC+4\) later: 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 =
- Correct target =

⇒

Is this a common occurrence?
Can we avoid these mispredictions?
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
  \[\Rightarrow\] eliminates false predictions after ALU instructions

- BTB contains entries only for control transfer instructions
  \[\Rightarrow\] 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
Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)
- Dynamic function call (jump to run-time function address)
- Subroutine returns (jump to return address)

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.

```c
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)/
&fd()
&fc()
&fb()
```
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)
Overview of Branch Prediction

Must speculation check always be correct?

Best predictors reflect program behavior

Tight loop
Loose loop
Loose loop
Loose loop
Next Lecture:
Speculative Execution
& Value Management