Branch Prediction & Speculative Execution

Krste Asanovic
Laboratory for Computer Science
Massachusetts Institute of Technology
# 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>20 %</td>
<td></td>
</tr>
<tr>
<td>FPU Mult</td>
<td>13 %</td>
<td></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?
Reducing Control Transfer Penalties

Software solution

- *loop unrolling*
  Increases the run length
- *instruction scheduling*
  Compute the branch condition as early as possible
  (limited)

Hardware solution

- *delay slots*
  replaces pipeline bubbles with useful work
  (requires software cooperation)
- *branch prediction & speculative execution*
  of instructions beyond the branch
Effect of Control Transfer on Pipelined Execution

Control transfer instructions require insertion of bubbles in the pipeline.

The number of bubbles depends upon the number of cycles it takes

• to determine the next instruction address, and

• to fetch the next instruction
# DLX Branches and Jumps

Must know (or guess) both target address and whether taken to execute branch/jump.

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

Next fetch started

Branch executed

PC
I-cache
Fetch Buffer
Fetch
Decode
Issue Buffer
Execute
Func. Units
Result Buffer
Commit
Arch. State
# Branch Penalties in Modern Pipelines

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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PC Generation/Mux</td>
</tr>
<tr>
<td>P</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>I</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>J</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
</tr>
<tr>
<td>E</td>
<td>Integer Execute</td>
</tr>
</tbody>
</table>

- **Branch**
  - Target Address Known
  - Direction & Jump
  - Register Target Known

Remainder of execute pipeline (+ another 6 stages)
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:
• In-order machines: kill instructions following branch in pipeline
• Out-of-order machines: shadow registers and memory buffers for each speculated branch
Static Branch Prediction
(Encode prediction as part of branch instruction)

Probability a branch is taken (~60-70% overall):

Can predict all taken,
or backwards taken/forward not-taken (use offset sign bit)

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

ISA can allow arbitrary choice of statically predicted direction (HP PA-RISC, Intel IA-64)
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)
Branch Prediction Bits

- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!

BP state:

take(right) \times \neg\text{take} \times \text{last prediction right/wrong}
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 bit: H records the direction of the last branch executed by the processor

Two sets of BHT bits (BHT0 & BHT1) per branch instruction

\[ H = 0 \text{ (not taken)} \quad \Rightarrow \quad \text{consult BHT0} \]
\[ H = 1 \text{ (taken)} \quad \Rightarrow \quad \text{consult BHT1} \]
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

0, 0

k

Taken/¬Taken?
```
Limitations of BHTs

Cannot redirect fetch stream until after branch instruction is fetched and decoded, and target address determined.

Correctly predicted taken branch penalty

Jump Register penalty

(UltraSPARC-III fetch pipeline)
Branch Target Buffer (BTB)

- 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

2^k-entry direct-mapped BTB
*can also be associative*
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?
BTB Performance

• 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 different call sites!
Subroutine Return Stack

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

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

Push call address when function call executed

Pop return address when subroutine return decoded

```
&fb()
&fc()
&fd()
```

k entries (typically k=8-16)
Speculating Both Directions

An alternative to branch prediction is to execute both directions of a branch *speculatively*

- resource requirement is proportional to the number of concurrent speculative executions
- only half the resources engage in useful work when both directions of a branch are executed speculatively
- branch prediction takes less resources than speculative execution of both paths

*With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction*
Mispredict Recovery

In-order execution machines:
- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution?
- Multiple instructions following branch in program order can complete before branch resolves
In-Order Commit for Precise Exceptions

- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- Commit (write-back to architectural state, regfile+memory) is in-order

Temporary storage in ROB holds results before commit
ROB for Precise Exceptions

- Add \(<pd, \text{dest}, \text{data}, \text{cause}>\) fields in the instruction template
- Commit instructions to reg file and memory in program order \(\Rightarrow\) buffers can be maintained circularly
- On exception, clear reorder buffer by resetting \(\text{ptr}_1 = \text{ptr}_2\)
  \((\text{stores must wait for commit before updating memory})\)

<table>
<thead>
<tr>
<th>Inst#</th>
<th>use exec</th>
<th>op</th>
<th>p1</th>
<th>src1</th>
<th>p2</th>
<th>src2</th>
<th>pd</th>
<th>dest</th>
<th>data</th>
<th>cause</th>
</tr>
</thead>
</table>

\(\text{ptr}_2\) next to commit

\(\text{ptr}_1\) next available
On mispredict

• Roll back “next available” pointer to just after branch
• Reset use bits
• Flush mis-speculated instructions from pipelines
• Restart fetch on correct branch path
Branch Misprediction in Pipeline

- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order

restart fetch on correct path
Killing Speculative Instructions

- Each instruction tagged with single “speculative” bit, carried throughout all pipelines
- Decode stage stalls if second branch encountered before first speculative branch resolves
- When speculative branch resolves:
  - Prediction incorrect, kill all instructions tagged as speculative
  - Prediction correct, clear speculative bit on all instructions

To allow speculation past multiple branches, add multiple bits per instruction indicating on which outstanding branches it is speculative
- speculation bits reclaimed when corresponding branch resolves and is oldest speculative branch in machine (manage allocation of speculation tag bits circularly)
Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted