Branch Prediction
and
Speculative Execution

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Based on the material prepared by Krishna Asanovic and Arvind
Outline

• Control transfer penalty

• Branch prediction schemes

• Branch misprediction recovery schemes
Phases of Instruction Execution

- **PC**
  - *Fetch:* Instruction bits retrieved from cache.

- **I-cache**
  - *Decode:* Instructions placed in appropriate issue (aka “dispatch”) stage buffer

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

- **Issue Buffer**

- **Func. Units**

- **Result Buffer**
  - *Commit:* Instruction irrevocably updates architectural state (aka “graduation” or “completion”).

- **Arch. State**
Fetch Stage

PC

Instruction Cache

Hit?

Opcode  Rd  Rsrc1  Rsrc2/Imm

Instructions

Fetch Buffer

To Decode Stage
Decode & Rename Stage

(Renaming is shown only for Rsrc2, similar for Rsrc1)

Committed Architectural Regfile

R31
R30
R0

Opcode

Tag
R31 R30
Rename Table

V Tag
R0

1 X
0 1 0 1 0 1

1

0 1
0 1

ImmSel

0 1

Rd
Data1
Data2
Pd
P2
P1
Tag1
Tag2
E
U

October 26, 2005
Execute Stage

- Arbiter selects one ready instruction (P1=1 AND P2=1) to execute
- Instruction reads operands from ROB, executes, and broadcasts tag and result to waiting instructions in ROB. Also saves result and exception flags for commit.
Commit Stage

- When instruction at ptr2 (commit point) has completed, write back result to architectural state and check for exceptions.
- Check if rename table entry for architectural register written matches tag, if so, clear valid bit in rename table.
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!
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
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

October 26, 2005
MIPS Branches and Jumps

Need to know (or guess) both target address and whether the branch/jump is taken or not

<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>

October 26, 2005
## Branch Penalties in Modern Pipelines

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

<table>
<thead>
<tr>
<th>Branch</th>
<th>PC Generation/Mux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>Address Known</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td></td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td></td>
<td>Complete Decode</td>
</tr>
<tr>
<td></td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td></td>
<td>Register File Read</td>
</tr>
<tr>
<td></td>
<td>Integer Execute</td>
</tr>
<tr>
<td>Register Target Known</td>
<td>Remainder of execute pipeline (+ another 6 stages)</td>
</tr>
<tr>
<td>Branch Direction &amp; Jump</td>
<td></td>
</tr>
<tr>
<td>Register Target Known</td>
<td></td>
</tr>
</tbody>
</table>

October 26, 2005
Outline

- Control transfer penalty
- Branch prediction schemes
- Branch misprediction recovery schemes
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

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

- **backward**: 90%
- **forward**: 50%

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!

\[
BP \text{ state: } (\text{predict } \text{take/} \neg \text{take}) \times (\text{last prediction right/wrong})
\]

October 26, 2005
Branch History Table

I-Cache

Instruction

Opcode

offset

Branch?

Target PC

Taken/¬Taken?

Fetch PC

BHT Index

$2^k$-entry BHT, 2 bits/entry

4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

October 26, 2005
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?
Exploiting Spatial Correlation
Yeh and Patt, 1992

```plaintext
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

- \( H = 0 \) (not taken) \( \Rightarrow \) consult BHT0
- \( H = 1 \) (taken) \( \Rightarrow \) consult BHT1
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

BP bits are stored with the predicted target address.

IF stage: If (BP=taken) then nPC=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 = 236
Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

Is this a common occurrence?
Can we avoid these bubbles?

October 26, 2005
BTB should be for Control Transfer instructions only

BTB contains useful information for branch and jump instructions only
⇒ it should not be updated 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 (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

October 26, 2005
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

BTB/BHT only updated after branch resolves in E stage
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 different 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.

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

Push call address when function call executed

Pop return address when subroutine return decoded

- \&fd()
- \&fc()
- \&fb()

k entries (typically k=8-16)

October 26, 2005
Outline

• Control transfer penalty

• Branch prediction schemes

• Branch misprediction recovery schemes

Five-minute break to stretch your legs
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, i.e., regfile & memory, is in-order)

Temporary storage needed in ROB to hold results before commit

October 26, 2005
Extensions for Precise Exceptions

<table>
<thead>
<tr>
<th>Inst#</th>
<th>use</th>
<th>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>

- `ptr_2`: next to commit
- `ptr_1`: next available

Reorder buffer

- add `<pd, dest, data, cause>` fields in the instruction template
- commit instructions to reg file and memory in program order ⇒ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting `ptr_1 = ptr_2`
  
  (stores must wait for commit before updating memory)

October 26, 2005
Branch Misprediction Recovery

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 by killing all the instructions in ROB that follow a mispredicted branch
Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted.
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*
Thank you!