Reducing pipeline stalls and misprediction penalties

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Code for the lecture is available on the course website under the code tab
In case of a misprediction we squash the instruction in D and loose one more cycle because of the conflict between F and EX_R.

In case the instruction in D has a RAW hazard with an instruction in WB or Ex, we will introduce one to two stalls in the pipeline.

Can we do better?
How frequent are branches?
RISC-V [S. Zhang 2018]

Spec INT 2006

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RV64G</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>branch %</td>
<td>load %</td>
<td>store %</td>
<td>other %</td>
</tr>
<tr>
<td>astar</td>
<td>2.80E+11</td>
<td>19.8</td>
<td>38.4</td>
<td>3.5</td>
<td>38.3</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.21E+11</td>
<td>12.7</td>
<td>23.8</td>
<td>6.5</td>
<td>57</td>
</tr>
<tr>
<td>gcc</td>
<td>6.46E+10</td>
<td>21.6</td>
<td>23.6</td>
<td>7.5</td>
<td>47.4</td>
</tr>
<tr>
<td>gobmk</td>
<td>2.50E+11</td>
<td>16.8</td>
<td>24.1</td>
<td>7.7</td>
<td>51.4</td>
</tr>
<tr>
<td>h264ref</td>
<td>5.71E+11</td>
<td>7.8</td>
<td>31.4</td>
<td>3.9</td>
<td>56.9</td>
</tr>
<tr>
<td>hmmer</td>
<td>9.91E+11</td>
<td>17.9</td>
<td>31.6</td>
<td>7.6</td>
<td>42.9</td>
</tr>
<tr>
<td>libquantum</td>
<td>1.24E+12</td>
<td>32.1</td>
<td>21.1</td>
<td>3.6</td>
<td>43.2</td>
</tr>
<tr>
<td>mcf</td>
<td>2.78E+11</td>
<td>24.6</td>
<td>41.8</td>
<td>6.4</td>
<td>27.1</td>
</tr>
<tr>
<td>omnetpp</td>
<td>5.84E+11</td>
<td>23.4</td>
<td>27.4</td>
<td>9.6</td>
<td>39.7</td>
</tr>
<tr>
<td>sjeng</td>
<td>2.75E+12</td>
<td>18.3</td>
<td>22.9</td>
<td>5.6</td>
<td>53.2</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>1.05E+12</td>
<td>30.7</td>
<td>31.9</td>
<td>3.5</td>
<td>33.9</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>21.7</td>
<td>27</td>
<td>5.5</td>
<td>45.8</td>
</tr>
</tbody>
</table>

Every 5th instruction is a branch
Static Branch Prediction

- Since most instructions do not result in a control transfer, pc+4 is a good predictor.
- 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
    - reported as ~80% accurate

... but our ISA is fixed!
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)

Large number of branch prediction schemes have been proposed and implemented; we will study a simple but effective one ⇒ BTB
Next Address Predictor: Branch Target Buffer (BTB)

Even small BTBs are very effective

Zero-size BTB behaves like the pc+4 nap

- BTB remembers recent target PC’s for a set of control instructions
  - Fetch: looks for the pc and the associated target in BTB; if pc is not found then ppc is pc+4
  - Execute: checks prediction, if wrong it poisons the wrong-path instructions; updates the BTB for jumps and taken-branches

BTB permits ppc to be determined before the instruction is decoded

May 3, 2018

L21-6
Next Addr Predictor interface

interface AddrPred;

method Addr nap(Addr pc);
method Action update(Addr pc, Addr nextPC, Bool taken);

dendinterface

**Predictor training:** On a pc misprediction, pc and epoch are updated and the relevant information is passed to the next address predictor

- **nap:** simple look up
- **update:** On a pc misprediction, if the jump or branch at the pc was taken then the BTB is updated with the new (pc, nextPC) otherwise the pc entry is deleted

You will implement a BTB module and integrate in the pipeline
Branch Prediction Bits

Remember how a branch was resolved previously

- Assume 2 BP bits per conditional branch instruction
- Use saturating counter

<table>
<thead>
<tr>
<th>On not taken</th>
<th>On taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Direction prediction changes only after two successive bad predictions

You may want to integrate prediction bits in your BTB for extra credit; training is not so simple
If the search by Decode does not see an instruction in the scoreboard, then its effect must have taken place. This means that any updates to the register file by that instruction must be visible to the subsequent register reads ⇒

- sb.remove and rf.wr should happen atomically
- sb.search and rf.rd1, rf.rd2 should happen atomically
Concurrent Analysis

- Normal register file: \( rd < wr \)
- This implies:
  - \( \text{doFetch} < \text{doExecute} \)
  - \( \text{sb}: \{\text{search, insert}\} < \text{remove} \Rightarrow \) normal \( \text{sb} \)
  - \( \text{d2e}: \) enq \( \{<, \text{CF}\} \) {deq, first} \( \Rightarrow \) CF FIFO
- Performance:
  - Up to two stalls after each RAW hazard

Let us consider \( \text{doExecute} < \text{doFetch} \)
Concurrency and Performance

**doExecute < doFetch**

- **Register File**
  - `rd1` → `rd2` → `redirect` → `wr`
  - `doFetch` → `doExecute`
- **Scoreboard**
  - `search` → `insert` → `d2e` → `remove`

**For doExecute < doFetch:**
- **rf:** \( wr < rd \) ⇒ bypass rf
- **sb:** remove < \{search, insert\} ⇒ bypass sb
- **d2e:** \{first, deq\} \{<, CF\} enq ⇒ pipelined or CF Fifo

**To avoid a stall due to a RAW hazard between successive instructions**
- **sb:** remove < search
- **rf:** \( wr < rd \) bypass rf

Let us assume we can design rf and sb with such properties
Bypassing from WB to Decode

- We just need to instantiate bypass versions of rf and sb and the rest of the code will remain the same.
- This will eliminate one stall but not both, i.e., an instruction with immediate dependency on the preceding instruction will still stall.
Reducing the misprediction penalty

- Currently doExecute_Redirect C doFetch
  - What if doExecute_Redirect < doFetch?
    => one less cycle delay for misprediction

How do you get doExecute_Redirect < doFetch?

- doExRedirect: pc.write$_R$
- doFetch: pc.read$_F$ and pc.write$_F$
- pc.write$_R$ <$> pc.read$_F$ < pc.write$_F$

This can’t be done using ordinary registers, so we introduce a new storage element, EHR, which will allow us to solve this problem

- The same EHR can also be used to implement other modules like, pipeline FIFO, bypass FIFO etc.
Ephemeral History Register (EHR)  Dan Rosenband [MEMOCODE’04]

Register with prioritized writes and a bypass port

- \( w[0].data \)
- \( w[0].en \)
- \( w[1].data \)
- \( w[1].en \)

\( r[1] \) returns:
- The current state if \( w[0] \) is not enabled
- The value being written (\( w[0].data \)) if \( w[0] \) is enabled

\( w[i+1] \) takes precedence over \( w[i] \)

\( r[0] < w[0] \) \( r[1] < w[1] \) \( w[0] < w[1] < ... \)
## Conflict Matrix of EHR

<table>
<thead>
<tr>
<th></th>
<th>EHR.r0</th>
<th>EHR.w0</th>
<th>EHR.r1</th>
<th>EHR.w1</th>
</tr>
</thead>
<tbody>
<tr>
<td>EHR.r0</td>
<td>CF</td>
<td>&lt;</td>
<td>CF</td>
<td>&lt;</td>
</tr>
<tr>
<td>EHR.w0</td>
<td>&gt;</td>
<td>C</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>EHR.r1</td>
<td>CF</td>
<td>&gt;</td>
<td>CF</td>
<td>&lt;</td>
</tr>
<tr>
<td>EHR.w1</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>

### Register

<table>
<thead>
<tr>
<th></th>
<th>reg.r</th>
<th>reg.w</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg.r</td>
<td>CF</td>
<td>&lt;</td>
</tr>
<tr>
<td>reg.w</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>
Reducing the misprediction penalty

- Declare the pc to be an EHR
- Redirect rule will write into port 0 of pc
- Fetch rule will read the bypass port of the pc and update port 1 of pc
Making One-Element FIFO into a *Pipelined FIFO*

```verilog
module mkFifo (Fifo#(1, t));
    Reg#(t)  d <= mkRegU;
    Ehr#(2, Bool) v <= mkEhr(False);
method Action enq(t x) if (!v[1]);
    v[1] <= True; d <= x;
endmethod
method Action deq if (v[0]);
    v[0] <= False;
endmethod
method t first if (v[0]);
    return d;
endmethod
endmodule
```

Pipelined FIFO behavior

```
deq < enq
first < deq
first < enq
```

Enq and deq can be ready simultaneously and can be executed concurrently.

Similarly, we can design Bypass FIFO and CF FIFO without an internal cycle.

Takeaway

- By proper use of bypasses (EHRs) and a good branch predictor we can reduce the cycles lost to stalls and squashes in the 4-stage pipeline to as low as 10%, provided the memory system is pipelined and can deliver a response in one cycle.
- Designing good memory system is as challenging as designing processors
- Designs with EHRs are harder to debug!