Complex Pipelines and Branch Prediction

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Processor Performance

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}
\]

\[\text{CPI} = \text{t}_{CK}\]

- Pipelining lowers \(t_{CK}\). What about CPI?

- \(\text{CPI} = \text{CPI}_{\text{ideal}} + \text{CPI}_{\text{hazard}}\)
  - \(\text{CPI}_{\text{ideal}}\): cycles per instruction if no stall

- \(\text{CPI}_{\text{hazard}}\) contributors
  - Data hazards: long operations, cache misses
  - Control hazards: branches, jumps, exceptions
5-Stage Pipelined Processors

- **Advantages**
  - $\text{CPI}_{\text{ideal}} = 1$ (pipelining)
  - Simple and elegant
    - Still used in ARM & MIPS processors

- **Shortcomings**
  - Upper performance bound is CPI=1
  - High-latency instructions not handled well
    - 1 stage for access to large caches or multiplier
    - Long clock cycle time
  - Unnecessary stalls due to rigid pipeline
    - If one instruction stalls, anything behind stalls
Improving 5-Stage Pipeline Performance

- Increase clock frequency: deeper pipelines
  - Overlap more instructions

- Reduce CPI_{ideal}: wider pipelines
  - Each pipeline stage processes multiple instructions

- Reduce impact of data hazards: out-of-order execution
  - Execute each instruction as soon as its source operands are available

- Reduce impact of control hazards: branch prediction
  - Predict both target and direction of jumps and branches
Deeper Pipelines

- Break up datapath into N pipeline stages
  - Ideal $t_{CK} = 1/N$ compared to non-pipelined
  - So let’s use a large N!

- Advantage: Higher clock frequency
  - The workhorse behind multi-GHz processors
  - Opteron: 11, Power5: 17, Pentium4: 34; Nehalem: 16

- Cost
  - More pipeline registers
  - Complexity: more bypass paths and control logic

- Disadvantages
  - More overlapping $\Rightarrow$ more dependencies
    - $CPI_{\text{hazard}}$ grows due to data and control hazards
  - Clock overhead becomes increasingly important
  - Power consumption
Wider (aka Superscalar) Pipelines

- Each stage operates on up to $W$ instructions each clock cycle

- Advantage: Lower $CPI_{\text{ideal}} = 1/W$
  - Opteron: 3, Pentium4: 3, Nehalem: 4, Power7: 8

- Cost
  - Need wider path to instruction cache
  - Need more ALUs, register file ports, ...
  - Complexity: more bypass & stall cases to check

- Disadvantages
  - Parallel execution $\Rightarrow$ more dependencies
    - $CPI_{\text{hazard}}$ grows due to data and control hazards
Resolving Hazards

- **Strategy 1**: Stall. Wait for the result to be available by freezing earlier pipeline stages

- **Strategy 2**: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

- **Strategy 3**: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly $\rightarrow$ do nothing
    - Guessed incorrectly $\rightarrow$ kill & restart with correct value

- **Strategy 4**: Find something else to do
Out-of-Order Execution

- Consider the expression \( D = 3(a - b) + 7ac \)

**Sequential code**
- ld a
- ld b
- sub a-b
- mul 3(a-b)
- ld c
- mul ac
- mul 7ac
- add 3(a-b)+7ac
- st d

Out-of-order execution runs instructions as soon as their inputs become available.
Out-of-Order Execution Example

- If `ld b` takes a few cycles (e.g., cache miss), can execute instructions that do not depend on `b`

**Sequential code**

```plaintext
ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d
```

**Dataflow graph**

![Dataflow graph](image)

- Completed
- Executing
- Not ready

November 27, 2018
A Modern Out-of-Order Superscalar Processor

Reconstruct dataflow graph

Execute each instruction as soon as it source operands are available

Write back results in program order

Why is this needed?
Control Flow Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!

- How much work is lost every time pipeline does not follow correct instruction flow?

  Loop length x Pipeline width

- One branch every 5-20 instructions... performance impact of mispredictions?
RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  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>JAL</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JALR</td>
<td>After Inst. Decode</td>
<td>After Inst. Execute</td>
</tr>
<tr>
<td>Branches</td>
<td>After Inst. Execute</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>
Resolving Hazards

- **Strategy 1: Stall.** Wait for the result to be available by freezing earlier pipeline stages

- **Strategy 2: Bypass.** Route data to the earlier pipeline stage as soon as it is calculated

- **Strategy 3: Speculate**
  - **Guess a value and continue executing anyway**
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

- **Strategy 4: Find something else to do**

Predict jump/branch target and direction
Static Branch Prediction

- Probability a branch is taken is ~60-70%, but:
  - Some ISAs attach preferred direction hints to branches, e.g., Motorola MC88110
    - bne0 (*preferred taken*)
    - beq0 (*not taken*)
  - Achieves ~80% accuracy
Dynamic Branch Prediction
Learning from 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)
Predicting the Target Address: Branch Target Buffer (BTB)

- BTB is a cache for targets: Remembers last target PC for taken branches and jumps
  - If hit, use stored target as predicted next PC
  - If miss, use PC+4 as predicted next PC
  - After target is known, update if prediction is wrong
Integrating the BTB in the Pipeline

Predict next PC immediately

Correct misprediction when the right outcome is known

Tight loop
Unlike caches, it is fine if the BTB produces an invalid next PC
  - It’s just a prediction!

Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)
  - Storing only a subset of target PC bits (fill missing bits from current PC)
  - Not storing valid bits

Even small BTBs are very effective!
interface BTB;
    method Addr predict(Addr pc);
    method Action update(Addr pc, Addr nextPC, Bool taken);
endinterface

- \textit{predict}: Simple lookup to predict nextPC in Fetch stage
- \textit{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.

A BTB is a good way to improve performance on parts 2 and 3 of the design project.
Better Branch Direction Prediction

- Consider the following loop:

  ```
  loop: ...
  addi a1, a1, -1
  bnez a1, loop
  ```

- How many mispredictions does the BTB incur per loop?
  - One on loop exit
  - Another one on first iteration
Two-Bit Direction Predictor
Smith 1981

- Use two bits per BTB entry instead of one valid bit
- Manage them as a 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 wrong predictions

- How many mispredictions per loop? 1
Modern Processors Combine Multiple Specialized Predictors

- **Fetch**: Predict next PC immediately
- **Decode**: Instruction type & branch/JAL target known
- **RegRead**: Branch direction & JALR target known
- **Execute**: Best predictors reflect program behavior
- **WriteBack**: Correct mispred

**BTB** (Branch Target Buffer):
- Branch dir predictor
- Return addr predictor
- Loop predictor
Summary

- Modern processors rely on a handful of techniques
  - Deep pipelines $\rightarrow$ Multi-GHz frequency
  - Wide (superscalar) pipelines $\rightarrow$ Multiple instructions/cycle
  - Out-of-order execution $\rightarrow$ Reduce impact of data hazards
  - Branch prediction $\rightarrow$ Reduce impact of control hazards

- Main objective is high sequential performance
  - High costs (area, power)
  - Requires high memory bandwidth and low latency
  - Simple to use (but knowing these techniques often needed to write high-performance programs!)
Thank you!

Next lecture: Bypassing in Bluespec  
+ Design Project Tips