CS 152: Discussion Section 7
Branch Predictor and VLIW

Albert Ou, Yue Dai
03/013/2020
Administrivia

- Problem Set 3 due 10:30am on Mon, March 16
- Lab 3 released today, due 10:30am on Mon, April 6
- Midterm 1 scores are available on Gradescope
  - One week to submit regrade requests
  - Regrade window opens at 4pm today
  - Solutions posted on course webpage
Agenda

● Branch Prediction
  ○ Branch History Table
  ○ Branch Target Buffer
● Load/Store Queue
● VLIW
  ○ Software Pipelining
● Lab 3 overview
Branch Prediction - BHT

- Exploit temporal correlation
- How to learn based on spatial correlation?

![Diagram of branch prediction with BHT (Branch History Table)]
Branch Prediction - BHT

- Use history register
- Worksheet Q1
- Q: what’s the limitation of just using BHT?
Branch Prediction - BTB

- Index by branch PC but need checking whether PC matches, and contains branch PC and target PC in the same line.
- Q: What target PC should be stored? Should we store the not-taken target PC?
- Q: Which happens earlier? BTB check or BHT check?
- Q: When should BTB be updated?
Branch Prediction - BTB update

- Here we assume using both BTB and BHT. BTB check in IF stage, BHT check in decode stage.
- But in a real design, the fetch stage may be pipelined, which makes BHT check occur in a later stage of IF.

*FIGURE 3.26 The steps involved in handling an instruction with a branch-target buffer.*
Load/Store Queue

- We would like to speculatively issue loads without violating in-order semantics and precise exceptions
- Q: What extra structure do you need?
Load/Store Queue

- Speculative Store Buffer
  - Dispatch:
    - Store: allocate an entry in store buffer in program order
    - Load: record the position of youngest store instruction older than this load
  - Execute:
    - Store: update the corresponding address and data in store buffer
    - Load: can only execute when all older store address are known: Find all stores prior to the load; If has, forward the data from the youngest match to load / If not, load from cache
  - Commit:
    - Store: store the data to cache, free that entry
    - Load: commit normally
- Q: What if you want to be more aggressive?
  Speculative load
Load/Store Queue

- Speculative Store Buffer + Load Queue
  - Can execute load instruction without waiting for all previous store address are known
  - Load Queue is used to keep the order of load instructions.
  - When a store address is finished execution, check all load addresses in load queue which is younger than this store.
    - If no match, keep executing normally
    - If has match, flush all instruction executions after the oldest load match

- Problem: too expensive, large penalty for inaccurate address speculation
VLIW

- Compiler
  - VLIW compiler needs to explicitly schedule operations to maximize parallel execution and avoid data hazards
  - Guarantees intra-instruction parallelism
- Q: How to better schedule the code
  - Loop unrolling
  - Software pipelining
  - Trace scheduling
VLIW - Software pipelining

- Software pipelining pays startup/wind-down costs only once per loop, not once per iteration
- Worksheet Q2
VLIW - Trace scheduling

- Find the most frequent branch path and optimize it
- Use profiling feedback
- Add fix up code
VLIW - Predicated execution

- Remove mispredicted branches by using predicated execution with predict register
- Predicate register true: execute; false: nop

```
<table>
<thead>
<tr>
<th>b0:</th>
<th>Inst 1</th>
<th>\textit{if}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inst 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>br a==b, b2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b1:</th>
<th>Inst 3</th>
<th>\textit{else}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inst 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>br b3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b2:</th>
<th>Inst 5</th>
<th>\textit{then}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inst 6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b3:</th>
<th>Inst 7</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inst 8</td>
<td></td>
</tr>
</tbody>
</table>
```

Predicate register
Execute either inst 3 & inst 4 or inst 5 & inst 6
BOOM: Berkeley Out-of-Order Machine

- Open-source, synthesizable, out-of-order superscalar RISC-V core
- Heavily inspired by the MIPS R10000 and Alpha 21264
- Unified physical register file with explicit renaming
- Split ROB / issue window design
- Extensively parameterized:
  - Fetch and issue widths, ROB size, LSU size
  - Functional unit mix, latencies
  - Issue scheduler
  - Composable branch predictors, RAS size, BTB size
  - Commit map table (R10k rollback vs Alpha 21264 single-cycle flush)
  - Maximum in-flight branches
BOOM: Berkeley Out-of-Order Machine
Open-Ended: Branch predictor design

- Implement a branch predictor in C++ that integrates with BOOM
- Objective is to improve accuracy over baseline BHT
- Competition:
  - Winning team receives 10% extra credit
  - Limited division: Constrained to 64 KiB of storage, plus 2048 bits of additional budget
  - Open division: No restrictions
  - Gradescope autograder will be deployed next week
Open-Ended: Spectre attacks

- **Spectre/Meltdown**: Microarchitectural side-channel attacks that exploit branch prediction, speculative execution, and cache timing to bypass security mechanisms.
- Objective is to recreate Spectre attacks on BOOM.
- Attack scenario:
  - Vulnerable Spectre gadget present in supervisor syscall code.
  - Write user program to infer secret data from protected kernel memory using branch predictor mis-training and cache side effects.
- Team that can guess most bytes correctly receives 10% extra credit.
  - Gradescope autograder will be deployed next week.