3810 Review Session
Spring 2024
Disk Basics

- Disk access remains very slow – mechanical head that has to move to the correct “ring” of data – order of milli-seconds – high enough that a context-switch is best
- Focus on other metrics, especially reliability
- A sector on the disk is associated with a cyclic redundancy code (CRC) – a hash that tells us if the read data is correct or not – it is simply an error detector, not an error corrector
- To correct the error, RAID is commonly used
- Reliability measures continuous service accomplishment and is usually expressed as mean time to failure (MTTF)
- Availability is measured as MTTF/(MTTF+MTTRecovery)
RAID

- RAID 0: no redundancy
- RAID 1: mirroring
- RAID 2 and 6: memory-style ECC and rarely deployed
- RAID 3: bit-interleaved, lower cost, but no query-level parallelism
- RAID 4: block-interleaved, lower cost, query-level parallelism, but write bottleneck
- RAID 5: block-interleaved, lower cost, query-level parallelism, write parallelism
- Parity and XOR!
<table>
<thead>
<tr>
<th>Circuit Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of full circuit:</td>
</tr>
<tr>
<td>Length of each stage:</td>
</tr>
<tr>
<td>No hazards</td>
</tr>
</tbody>
</table>

**Unpipelined processor**
- **CPI:**
- **Clock speed:**
- **Throughput:**

**Pipelined processor**
- **CPI:**
- **Clock speed:**
- **Throughput:**

**Pipeline Performance**
No Bypassing
(for the 5-stage pipeline)
Point of production: always RW middle
Point of consumption: always D/R middle

<table>
<thead>
<tr>
<th>I1</th>
<th>add:</th>
<th>IF</th>
<th>DR</th>
<th>AL</th>
<th>DM</th>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>add:</td>
<td>IF</td>
<td>DR</td>
<td>DR</td>
<td>DR</td>
<td>AL</td>
</tr>
</tbody>
</table>

* PoP

<table>
<thead>
<tr>
<th>I1</th>
<th>add:</th>
<th>IF</th>
<th>DR</th>
<th>AL</th>
<th>DM</th>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>add:</td>
<td>IF</td>
<td>DR</td>
<td>AL</td>
<td>DM</td>
<td>RW</td>
</tr>
</tbody>
</table>

* PoC

Bypassing
Point of production:
  add, sub, etc.: end of ALU
  lw: end of DM

Point of consumption:
  add, sub, lw: start of ALU
  sw $1, 8($2): start of ALU for $2, start of DM for $1

<table>
<thead>
<tr>
<th>I1</th>
<th>add:</th>
<th>IF</th>
<th>DR</th>
<th>AL</th>
<th>DM</th>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>add:</td>
<td>IF</td>
<td>DR</td>
<td>AL</td>
<td>DM</td>
<td>RW</td>
</tr>
</tbody>
</table>

* PoP

<table>
<thead>
<tr>
<th>I1</th>
<th>add:</th>
<th>IF</th>
<th>DR</th>
<th>AL</th>
<th>DM</th>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>add:</td>
<td>IF</td>
<td>DR</td>
<td>AL</td>
<td>DM</td>
<td>RW</td>
</tr>
</tbody>
</table>

* PoC

Data Hazards
Control Hazards

Assumptions

100 instructions
20 branches
14 Not-Taken, 6 Taken
Branch resolved in 6\textsuperscript{th} cycle (penalty of 5)

Approach 1: Panic and wait

Approach 2: Fetch-next-instr

Approach 3: Branch Delay Slot
Option A: always useful
Option B: useful when the branch goes along common fork
Option C: useful when the branch goes along uncommon fork
Option D: no-op, always non-useful

Approach 4: Branch predictor
Accuracy of 90%
Out of Order Processor

Branch prediction and instr fetch

R1 ← R1+R2
R2 ← R1+R3
BEQZ R2
R3 ← R1+R2
R1 ← R3+R2

Instr Fetch Queue

Decode & Rename

R1 ← R1+R2
R2 ← R1+R3
BEQZ R2
R3 ← R1+R2
R4 ← R3+R2

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 6

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 6

T1 ← R1+R2
T2 ← T1+R3
BEQZ T2
T4 ← T1+T2
T5 ← T4+T2

T1
T2
T3
T4
T5
T6

Reorder Buffer (ROB)

Register File R1-R32

ALU
ALU
ALU

Results written to ROB and tags broadcast to IQ

Out of Order Processor
Assumptions

1000 instructions, 1000 cycles, no stalls with L1 hits

# loads/stores:
% of loads/stores that show up at L2:
% of loads/stores that show up at L3:
% of loads/stores that show up at mem:
L2 acc = 10 cyc, L3 acc = 25 cyc, mem acc = 200 cyc
Assumptions

512KB cache, 8-way set-associative, 64-byte blocks, 32-bit addresses

Data array size = \#sets \times \#ways \times \text{blocksize}
Tag array size = \#sets \times \#ways \times \text{tagsize}
Offset bits = \log(\text{blocksize})
Index bits = \log(\#sets)
Tag bits + index bits + offset bits = \text{addresswidth}
Assumptions

16 sets, 1 way, 32-byte blocks

Access pattern:  4  40  400  480  512  520  1032  1540

Offset = address % 32  (address modulo 32, extract last 5)
Index = address/32 % 16  (shift right by 5, extract last 4)
Tag = address/512       (shift address right by 9)

<table>
<thead>
<tr>
<th>32-bit address</th>
<th>23 bits tag</th>
<th>4 bits index</th>
<th>5 bits offset</th>
<th>H/M</th>
<th>Evicted address</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>400</td>
<td>0</td>
<td>12</td>
<td>16</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>480</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>520</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>H</td>
<td>-</td>
</tr>
<tr>
<td>1032</td>
<td>2</td>
<td>0</td>
<td>8</td>
<td>M</td>
<td>512</td>
</tr>
<tr>
<td>1540</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>M</td>
<td>1024</td>
</tr>
</tbody>
</table>
Show how the following addresses map to the cache and yield hits or misses. The cache is direct-mapped, has 16 sets, and a 64-byte block size. Addresses: 8, 96, 32, 480, 976, 1040, 1096

Offset = address % 64  (address modulo 64, extract last 6)  
Index = address/64 % 16     (shift right by 6, extract last 4)  
Tag = address/1024          (shift address right by 10)
Consider a 3-processor multiprocessor connected with a shared bus that has the following properties:

(i) centralized shared memory accessible with the bus, (ii) snooping-based MSI cache coherence protocol,
(iii) write-invalidate policy. Also assume that the caches have a writeback policy. Initially, the caches all
have invalid data. The processors issue the following three requests, one after the other. Similar to slide
17 of lecture 25, fill in the following table to indicate what happens for every request. Also indicate
if/when memory writeback is performed. (8 points)

<table>
<thead>
<tr>
<th>Request</th>
<th>Cache Hit/Miss</th>
<th>Request on the bus</th>
<th>Who responds</th>
<th>State in Cache 1</th>
<th>State in Cache 2</th>
<th>State in Cache 3</th>
<th>State in Cache 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: Read X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3: Read X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Request</th>
<th>Cache Hit/Miss</th>
<th>Request on the bus</th>
<th>Who responds</th>
<th>State in Cache 1</th>
<th>State in Cache 2</th>
<th>State in Cache 3</th>
<th>State in Cache 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: Read X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3: Read X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Questions to ask yourself:
How does Meltdown work?
How does Spectre work?
How can you force a footprint? (the relevant code sequence)
How can you examine footprints? (exploiting the side channel)
How can you defend against these attacks?
<table>
<thead>
<tr>
<th><strong>Questions to ask yourself:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>What does the programmer/compiler deal with?</td>
</tr>
<tr>
<td>What does the OS deal with?</td>
</tr>
<tr>
<td>How is translation done efficiently?</td>
</tr>
</tbody>
</table>
Questions to ask yourself:
Why do multiprocs need to deal with prog. models, coherence, synchronization, consistency?
What are race conditions?
What is an example synchronization primitive and how is it implemented?
What consistency model is assumed by a programmer?
Why is it slow?
How do I make life easier for the programmer and provide high performance?
Questions to ask yourself:
What are the central philosophies in a GPU?
In what ways does the GPU design differ from a CPU?
What are the different ways that disks provide high reliability?
Can you explain how parity is used to recover lost data?