Lecture 14
Fundamental Memory Concepts (Part 2)

Xuan ‘Silvia’ Zhang
Washington University in St. Louis

http://classes.engineering.wustl.edu/ese566/
Memory Structure and Technology

- Register Files
Memory Structure and Technology

- SRAM (cache, on-chip)
Memory Structure and Technology

- DRAM
Memory Structure and Technology

- **DRAM**

Adapted from [Foss, "Implementing Application-Specific Memory." ISSCC'96]
Memory Structure and Technology

- **Disk**
  - magnetic hard drives require rotating platters resulting in long random access times with have hardly improved over several decades

- **Flash**
  - solid-state drives using flash have 100x lower latencies, but also lower density and higher cost
Memory Technology Trade-offs

Latches & Registers

Register Files

SRAM

DRAM

Flash & Disk

Low Capacity
Low Latency
High Bandwidth
(more and wider ports)

High Capacity
High Latency
Low Bandwidth
Latency Numbers:
every programmer (architect) should know

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>1 ns</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>3 ns</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>4 ns</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>17 ns</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
</tr>
<tr>
<td>Send 2KB over commodity network</td>
<td>250 ns</td>
</tr>
<tr>
<td>Compress 1KB with zip</td>
<td>2 us</td>
</tr>
<tr>
<td>Read 1MB sequentially from main memory</td>
<td>9 us</td>
</tr>
<tr>
<td>SSD random read</td>
<td>16 us</td>
</tr>
<tr>
<td>Read 1MB sequentially from SSD</td>
<td>156 us</td>
</tr>
<tr>
<td>Round trip in datacenter</td>
<td>500 us</td>
</tr>
<tr>
<td>Read 1MB sequentially from disk</td>
<td>2 ms</td>
</tr>
<tr>
<td>Disk random read</td>
<td>4 ms</td>
</tr>
<tr>
<td>Packet roundtrip from CA to Netherlands</td>
<td>150 ms</td>
</tr>
</tbody>
</table>

find updated at [https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html](https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html)
Cache Memories in Computer Architecture

• Three key questions
  - how much data is aggregated in a cache line
  - how do we organize multiple lines in cache
  - what data is replaced to make room for new data when cache is full

• Categorizing misses

• Write policies

• Multi-level cache
Typical Access Pattern
instruction vs data access, temporal vs spatial locality
Understand Locality

Examine each of the following assembly programs and rank each program based on the level of **temporal and spatial locality** in both the **instruction and data address stream** on a scale from 0 to 5 with 0 being no locality and 5 being very significant locality.

<table>
<thead>
<tr>
<th></th>
<th>Inst Temp</th>
<th>Inst Temp</th>
<th>Data Temp</th>
<th>Data Temp</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loop:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>r1, 0(r2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>r3, 0(r4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td>r5, r1, r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>r5, 0(r6)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td>r2, r2, 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td>r4, r4, 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td>r4, r4, 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td>r6, r6, 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td>r7, r7, -1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>r7, r0, loop</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Q1: How Much Data is Stored in a Cache Line?
Q2: How to Organize Multiple Lines in the Cache?

Four-line direct-mapped cache with 4B cache lines

Example execution worksheet and table for direct-mapped cache

<table>
<thead>
<tr>
<th>Dynamic Transaction Stream</th>
<th>0x000</th>
<th>0x004</th>
<th>0x008</th>
<th>0x00c</th>
<th>0x010</th>
<th>0x014</th>
<th>0x018</th>
<th>0x01c</th>
<th>0x020</th>
<th>0x024</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd 0x000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd 0x004</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd 0x008</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd 0x010</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd 0x00c</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd 0x010</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd 0x000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd 0x004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

V  Tag  Data

Set 0
Set 1
Set 2
Set 3

hit

4 Sets
Increase Cache Associativity

Four-line two-way set-associative cache with 4B cache lines

Four-line fully-associative cache with 4B cache lines
Increase Cache Line Size
Q3: What Data to be Replaced When Cache is Full?

- No choice in a direct-mapped cache
- Random
  - good average case performance, but difficult to implement
- Least recently used (LRU)
  - replace cache line which has not been accessed recently
  - exploits temporal locality
  - LRU cache state must be updated on every access
  - two-way cache can use a single “last used bit”
  - pseudo-LRU uses binary tree to approximate LRU for higher associativity
- First-In First-Out (FIFO, Round Robin)
  - simpler implementation without exploiting temporal locality
  - potentially useful in large fully-associative caches
Categorize Misses: The Three C’s

- **Compulsory**
  - first-reference to a block

- **Capacity**
  - cache is too small to hold all of the data

- **Conflict**
  - collisions in a specific set
Classify Misses as a Sequence of Three Questions

• Q1: would this miss occur in a cache with infinite capacity?
  - if “yes”, then this is a compulsory miss
  - if “no”, consider Q2

• Q2: would this miss occur in a fully-associate cache with the desired capacity?
  - if “yes”, then this is a capacity miss
  - if “no”, then consider Q3

• Q3: would this miss occur in a cache with the desired capacity and associativity?
  - if “yes”, then this is a conflict miss
  - if “no”, then this is not a miss—it is a hit!
Examples

Assume we have a direct-mapped cache with two 16B lines, each with four 4B words for a total cache capacity of 32B. We will need four-bits for the offset, one bit for the index, and the remaining bits for the tag.

<table>
<thead>
<tr>
<th></th>
<th>tag</th>
<th>idx</th>
<th>h/m</th>
<th>type</th>
<th>Set 0</th>
<th>Set 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>0x000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>0x020</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>0x000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>0x020</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume we have a direct-mapped cache with two 16B lines, each with four 4B words for a total cache capacity of 32B. We will need four-bits for the offset, one bit for the index, and the remaining bits for the tag.

<table>
<thead>
<tr>
<th></th>
<th>tag</th>
<th>idx</th>
<th>h/m</th>
<th>type</th>
<th>Set 0</th>
<th>Set 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>0x000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>0x020</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>0x030</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>0x000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Write Policies

- **Write-through with no write allocate**
  - on write miss, write memory but do not bring line into cache
  - on write hit, write both cache and memory
  - require more memory bandwidth, but simpler implementation

- **Write-back with write allocate**
  - on write miss, bring cache line into cache then write
  - on write hit, only write cache, do not write memory
  - only update memory when a dirty cache line is evicted
  - more efficient, but more complicated to implement
Translation, Protection, and Virtualization

- **Translation**
  - mapping of virtual address to physical address
- **Protection**
  - permission to access address in memory
- **Virtualization**
  - transparent extension of memory space using disk

Most modern systems provide support for all three functions with a single paged-based memory management unit (MMU).
Analyze Memory Performance

\[
\frac{\text{Time}}{\text{Mem Access Sequence}} = \frac{\text{Mem Accesses}}{\text{Sequence}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- mem access/sequence depends on program and translation
- time/cycle depends on microarchitecture and implementation
- also known as average memory access latency (AMAL)
Analyze Memory Performance

\[
\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left( \frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)
\]

- average cycles/hit is called the hit latency; it depends on microarchitecture
- number of misses/number of access is called the miss rate; it depends on microarchitecture
- average extra cycles/miss is called the miss penalty; it depends on microarchitecture, rest of memory system
Analyze Memory Performance

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Hit Latency</th>
<th>Extra Accesses for Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM Cache</td>
<td>&gt;1</td>
<td>1+</td>
</tr>
<tr>
<td>Pipelined Cache</td>
<td>≈1</td>
<td>1+</td>
</tr>
<tr>
<td>Pipelined Cache + TLB</td>
<td>≈1</td>
<td>≈0</td>
</tr>
</tbody>
</table>
Transactions and Steps, Now for Memory

- Executing a memory access involves a sequence of steps
  - check tag: check one or more tags in cache
  - select victim: select victim line from cache using replacement policy
  - evict victim: evict victim line from cache and write victim to memory
  - refill: refill requested line by reading line from memory
  - write mem: write requested word to memory
  - access data: read or write requested word in cache
Memory Microarchitecture Overview

Diagram showing the flow of data between memory management units, control units, and main memory. The diagram includes signals such as mmureq, mmuresp, cachereq, cacheresp, mreq, mresp, and data path components like tag array and data array.
High-level Idea for FSM Cache
FSM Cache Datapath

MT: Check tag
MRD: Read data array, return cacheresp
R0: Send refill memreq, get memresp
R1: Write data array with refill cache line
MWD: Send write memreq, write data array
High-level Idea for Pipelined Cache

FSM

Pipelined
Pipeline Cache Datapath
Questions?

Comments?

Discussion?
Acknowledgement

Cornell University, ECE 4750