IC220 Set #17:
Caching Finale and Virtual Reality
(Chapter 5)

ADMIN

- Reading – finish Chapter 5
  - Sections 5.7 (pgs 446-451 are optional), 5.8, 5.15, 5.16
Cache Performance

• Simplified model:
  \[
  \text{execution time} = (\text{execution cycles} + \text{stall cycles}) \times \text{cycle time}
  = \text{execTime} + \text{stallTime}
  \]

  \[
  \text{stall cycles} = \frac{\text{MemoryAccesses}}{\text{Program}} \cdot \text{MissRate} \cdot \text{MissPenalty}
  \]

  \[
  \text{(or)} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Misses}}{\text{Instruction}} \cdot \text{MissPenalty}
  \]

• Two typical ways of improving performance:
  – decreasing the miss rate
  – decreasing the miss penalty

  *What happens if we increase block size?*

  *Add associativity?*

Performance Example

• Suppose processor has a CPI of 1.5 given a perfect cache. If there are 1.2 memory accesses per instruction, a miss penalty of 20 cycles, and a miss rate of 10%, what is the effective CPI with the real cache?
Split Caches

- Instructions and data have different properties
  - May benefit from different cache organizations (block size, assoc…)

![Cache Diagram](image)

- Why else might we want to do this?

Cache Complexities

- Not always easy to understand implications of caches:

![Theoretical vs. Observed Behavior](image)
Cache Complexities

- Here is why:
  - Memory system performance is often critical factor
    - multilevel caches, pipelined processors, make it harder to predict outcomes
    - Compiler optimizations to increase locality sometimes hurt ILP
  - Difficult to predict best algorithm: need experimental data

---

Program Design for Caches – Example 1

- Option #1
  ```
  for (j = 0; j < 20; j++)
      for (i = 0; i < 200; i++)
          x[i][j] = x[i][j] + 1;
  ```

- Option #2
  ```
  for (i = 0; i < 200; i++)
      for (j = 0; j < 20; j++)
          x[i][j] = x[i][j] + 1;
  ```
Program Design for Caches – Example 2

• Why might this code be problematic?
  int A[1024][1024];
  int B[1024][1024];
  for (i = 0; i < 1024; i++)
    for (j = 0; j < 1024; j++)
      A[i][j] += B[i][j];

• How to fix it?

VIRTUAL MEMORY
Virtual memory summary (part 1)

Data access without virtual memory:

| Memory address | 31 30 29 28 27  .......... | 15 14 13 12 11 10 9 8  .......... | 3 2 1 0 |

Disk → Cache → Memory

Virtual memory summary (part 2)

Data access with virtual memory:

| Virtual address | 31 30 29 28 27  .......... | 15 14 13 12 11 10 9 8  .......... | 3 2 1 0 |

<table>
<thead>
<tr>
<th>Virtual page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>29 28 27  ..........</td>
<td>15 14 13 12 11 10 9 8  ..........</td>
</tr>
</tbody>
</table>

Disk → Cache → Memory
Virtual Memory

• Main memory can act as a cache for the secondary storage (disk)

  ![Virtual Memory Diagram](image)

  - Advantages:
    - Illusion of having more physical memory
    - Program relocation
    - Protection

• Note that main point is caching of disk in main memory but will affect all our memory references!

Address Translation

Terminology:
- Cache block
- Cache miss
- Cache tag
- Byte offset

![Address Translation Diagram](image)
Pages: virtual memory blocks

- Page faults: the data is not in memory, retrieve it from disk
  - huge miss penalty (slow disk), thus
    - pages should be fairly
  - Replacement strategy:
    - can handle the faults in software instead of hardware

- Writeback or write-through?

Page Tables

![Diagram](image)
Example – Address Translation Part 1

- Our virtual memory system has:
  - 32 bit virtual addresses
  - 28 bit physical addresses
  - 4096 byte page sizes
- How to split a virtual address?

<table>
<thead>
<tr>
<th>Virtual page #</th>
<th>Page offset</th>
</tr>
</thead>
</table>

- What will the physical address look like?

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>Page offset</th>
</tr>
</thead>
</table>

- How many entries in the page table?

Example – Address Translation Part 2

Translate the following addresses:
1. C0001560

<table>
<thead>
<tr>
<th>Physical Page or Disk Block #</th>
</tr>
</thead>
<tbody>
<tr>
<td>A204</td>
</tr>
<tr>
<td>A200</td>
</tr>
<tr>
<td>FB00</td>
</tr>
<tr>
<td>8003</td>
</tr>
<tr>
<td>7290</td>
</tr>
<tr>
<td>5600</td>
</tr>
<tr>
<td>F5C0</td>
</tr>
</tbody>
</table>

2. C0006123

3. C0002450

...
Making Address Translation Fast

- A cache for address translations: translation lookaside buffer

![Diagram of translation lookaside buffer and page table]

Typical values: 16-512 PTEs (page table entries),
mis-rate: .01% - 1%
mis-penalty: 10 – 100 cycles

Protection and Address Spaces

- Every program has its own “address space”
  - Program A’s address 0xc000 0200 not same as program B’s
  - OS maps every virtual address to distinct physical addresses

- How do we make this work?
  - Page tables –
  - TLB –

- Can program A access data from program B? Yes, if...
  1. OS can map different virtual page #’s to same physical page #’s
     - So A’s 0xc000 0200 = B’s 0xb320 0200
  2. Program A has read or write access to the page
  3. OS uses supervisor/kernel protection to prevent user programs
     from modifying page table/TLB
Integrating Virtual Memory, TLBs, and Caches

TLBs and Caches

What happens after translation?
## Modern Systems

### Concluding Remarks

- **Fast memories are small, large memories are slow**
  - We really want fast, large memories
  - Caching gives this illusion
- **Principle of locality**
  - Programs use a small part of their memory space frequently
- **Memory hierarchy**
  - $L_1$ cache $\leftrightarrow L_2$ cache $\leftrightarrow \ldots \leftrightarrow$ DRAM memory $\leftrightarrow$ disk
- **Memory system design is critical for multiprocessors**

### Memory Characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>ARM Cortex-A7</th>
<th>Intel Nehalem</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache organization</td>
<td>Split instruction and data caches</td>
<td>Split instruction and data caches</td>
</tr>
<tr>
<td>L1 cache size</td>
<td>32 KB each for instructions/data per core</td>
<td>32 KB each for instructions/data per core</td>
</tr>
<tr>
<td>L1 cache associativity</td>
<td>4-way (Bi) / 4-way (Bi) set associative</td>
<td>4-way (Bi) / 8-way (Bi) set associative</td>
</tr>
<tr>
<td>L1 replacement</td>
<td>Random</td>
<td>Approximate (B)</td>
</tr>
<tr>
<td>L2 block size</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 write policy</td>
<td>Write-back, Write-allocation</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td>L1 hit time (clocks)</td>
<td>3 clock cycles</td>
<td>4 clock cycles, predicted</td>
</tr>
<tr>
<td>L2 cache organization</td>
<td>Unified (instruction and data)</td>
<td>Unified (instruction and data) per core</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>32 KB to 1 MB</td>
<td>256 KB to 256 MB</td>
</tr>
<tr>
<td>L2 cache associativity</td>
<td>8-way set associative</td>
<td>8-way set associative</td>
</tr>
<tr>
<td>L2 replacement</td>
<td>Random</td>
<td>Approximate (B)</td>
</tr>
<tr>
<td>L2 block size</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L2 write policy</td>
<td>Write-back, Write-allocation</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td>L3 hit time (clocks)</td>
<td>3 clock cycles</td>
<td>3 clock cycles</td>
</tr>
<tr>
<td>L3 cache organization</td>
<td>Unified (instruction and data)</td>
<td>Unified (instruction and data)</td>
</tr>
<tr>
<td>L3 cache size</td>
<td>–</td>
<td>8 MB, shared</td>
</tr>
<tr>
<td>L3 cache associativity</td>
<td>–</td>
<td>16-way set associative</td>
</tr>
<tr>
<td>L3 replacement</td>
<td>–</td>
<td>Approximate (B)</td>
</tr>
<tr>
<td>L3 block size</td>
<td>–</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L3 write policy</td>
<td>–</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td>L3 hit time</td>
<td>–</td>
<td>3 clock cycles</td>
</tr>
</tbody>
</table>