Caching and TLBs

Sarah Diesburg
Operating Systems
CS 3430
Caching

- Store copies of data at places that can be accessed more quickly than accessing the original
  - Speed up access to frequently used data
  - At a cost: Slows down the infrequently used data
Caching in Memory Hierarchy

- Provides the illusion of GB storage
- With register access time

<table>
<thead>
<tr>
<th>Memory Level</th>
<th>Component</th>
<th>Access Time</th>
<th>Size</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary memory</td>
<td>Registers</td>
<td>1 clock cycle</td>
<td>~500 bytes</td>
<td>On chip</td>
</tr>
<tr>
<td></td>
<td>Cache</td>
<td>1-2 clock cycles</td>
<td>&lt;10 MB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Main memory</td>
<td>1-4 clock cycles</td>
<td>&lt; 4GB</td>
<td>$0.1/MB</td>
</tr>
<tr>
<td>Secondary memory</td>
<td>Disk</td>
<td>5-50 msec</td>
<td>&lt; 1TB</td>
<td>$0.07/GB</td>
</tr>
</tbody>
</table>
Caching in Memory Hierarchy

- Exploits two hardware characteristics
  - Smaller memory provides faster access times
  - Large memory provides cheaper storage per byte
- Puts frequently accessed data in small, fast, and expensive memory
- Assumption: non-random program access behaviors
Locality in Access Patterns

- **Temporal locality**: recently referenced locations are more likely to be referenced in the near future
  - e.g., files
- **Spatial locality**: referenced locations tend to be clustered
  - e.g., listing all files under a directory
Caching

- Storing a small set of data in cache
  - Provides the following illusions
    - Large storage
    - Speed of small cache
  - Does not work well for programs with little localities
    - e.g., scanning the entire disk
      - Leaves behind cache content with no localities (cache pollution)
Generic Issues in Caching

- Effective metrics
  - Cache hit: a lookup is resolved by the content stored in cache
  - Cache miss: a lookup cannot be resolved by the content stored in cache

- Effective access time:
  \[ P(\text{hit}) \times \text{(hit\_cost)} + P(\text{miss}) \times \text{(miss\_cost)} \]
Effective Access Time

- Cache hit rate: 99%
  - Cost: 2 clock cycles
- Cache miss rate: 1%
  - Cost: 4 clock cycles
- Effective access time:
  - $99\% \times 2 + 1\% \times (2 + 4)$
  - $= 1.98 + 0.06 = 2.04 \text{ (clock cycles)}$
Implications

- 10 MB of cache
  - Illusion of 4 GB of memory
  - Running at the speed of hardware cache
Reasons for Cache Misses

- **Compulsory misses:** data brought into the cache for the first time
  - e.g., booting

- **Capacity misses:** caused by the limited size of a cache
  - A program may require a hash table that exceeds the cache capacity
    - Random access pattern
    - No caching policy can be effective
Reasons for Cache Misses

- Misses due to competing cache entries: a cache entry assigned to two pieces of data
  - When both active
  - Each will preempt the other

- Policy misses: caused by cache replacement policy, which chooses which cache entry to replace when the cache is full
Design Issues of Caching

- How is a cache entry lookup performed?
- Which cache entry should be replaced when the cache is full?
- How to maintain consistency between the cache copy and the real data?
Caching Applied to Address Translation

- Process references the same page repeatedly
  - Translating each virtual address to physical address is wasteful
- *Translation lookaside buffer (TLB)*
  - Track frequently used translations
  - Avoid translations in the common case
Caching Applied to Address Translation

Virtual addresses → TLB → Translation table → Physical addresses

Data offsets into blocks (untranslated)

In TLB
## Example of the TLB Content

<table>
<thead>
<tr>
<th>Virtual page number (VPN)</th>
<th>Physical page number (PPN)</th>
<th>Control bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>Valid, rw</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>Invalid</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>Valid, rw</td>
</tr>
</tbody>
</table>
TLB Lookups

- Sequential search of the TLB table
- **Direct mapping:** assigns each virtual page to a specific slot in the TLB
  - e.g., use upper bits of VPN to index TLB
Direct Mapping

```c
if (TLB[UpperBits(vpn)].vpn == vpn) {
    return TLB[UpperBits(vpn)].ppn;
} else {
    ppn = PageTable[vpn];
    TLB[UpperBits(vpn)].control = INVALID;
    TLB[UpperBits(vpn)].vpn = vpn;
    TLB[UpperBits(vpn)].ppn = ppn;
    TLB[UpperBits(vpn)].control = VALID | RW
    return ppn;
}
```
Direct Mapping

- When use only high order bits
  - Two pages may compete for the same TLB entry
    - May toss out needed TLB entries

- When use only low order bits
  - TLB reference will be clustered
    - Failing to use full range of TLB entries

- Common approach: combine both
TLB Lookups

- Sequential search of the TLB table
- **Direct mapping:** assigns each virtual page to a specific slot in the TLB
  - e.g., use upper bits of VPN to index TLB
- **Set associativity:** use N TLB banks to perform lookups in parallel
Two-Way Associative Cache
Two-Way Associative Cache
Two-Way Associative Cache

If miss, translate and replace one of the entries
TLB Lookups

- **Direct mapping**: assigns each virtual page to a specific slot in the TLB
  - e.g., use upper bits of VPN to index TLB
- **Set associativity**: use N TLB banks to perform lookups in parallel
- **Fully associative cache**: allows looking up all TLB entries in parallel
Fully Associative Cache
Fully Associative Cache
If miss, translate and replace one of the entries

Fully Associative Cache
Typically
- TLBs are small and fully associative
- Hardware caches use direct mapped or set-associative cache
Replacement of TLB Entries

- Direct mapping
  - Entry replaced whenever a VPN mismatches

- Associative caches
  - Random replacement
  - LRU (least recently used)
  - MRU (most recently used)
  - Depending on reference patterns
Replacement of TLB Entries

- **Hardware-level**
  - TLB replacement is mostly random
    - Simple and fast

- **Software-level**
  - Memory page replacements are more sophisticated
    - CPU cycles vs. cache hit rate
Consistency Between TLB and Page Tables

- Different processes have different page tables
  - TLB entries need to be invalidated on context switches
- Alternatives:
  - Tag TLB entries with process IDs
  - Additional cost of hardware and comparisons per lookup
Relationship Between TLB and HW Memory Caches

- We can extend the principle of TLB
- *Virtually addressed cache*: between the CPU and the translation tables
- *Physically addressed cache*: between the translation tables and the main memory
Relationship Between TLB and HW Memory Caches

Virtually addressed cache

<table>
<thead>
<tr>
<th>VA</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>data</td>
</tr>
</tbody>
</table>

Translation tables

Physically addressed cache

<table>
<thead>
<tr>
<th>PA</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
<td>data</td>
</tr>
<tr>
<td>PA</td>
<td>data</td>
</tr>
<tr>
<td>PA</td>
<td>data</td>
</tr>
</tbody>
</table>

Data offsets inside blocks (untranslated)
Two Ways to Commit Data Changes

- **Write-through**: immediately propagates update through various levels of caching
  - For critical data

- **Write-back**: delays the propagation until the cached item is replaced
  - Goal: spread the cost of update propagation over multiple updates
  - Less costly