CS162
Operating Systems and Systems Programming
Lecture 10
Caches and TLBs

February 26, 2014
Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Goals for Today’s Lecture
• Paging-based, Segmentation-based, and Multi-level Translation Review
• Caching
  – Misses
  – Organization
• Translation Look aside Buffers (TLBs)
• How Caching and TLBs fit into the Virtual Memory Architecture

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.
What happens if stack grows to 1110 0000?

No room to grow! Buffer overflow error or resize segment and move segments around to make room.

What happens if stack grows to 1110 0000?

No room to grow! Buffer overflow error or resize segment and move segments around to make room.

Allocate new pages where room!
**Review: Multi-level Translation**

- A tree of tables
  - Lowest level page table maps memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):
  - Virtual memory view
    - Top-level page table pointer register (2-level page tables)
    - Higher levels often segmented
    - Lowest level page table

**Review: Two-level paging**

- Tree of Page Tables
  - Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
  - Valid bits on Page Table Entries
    - Don’t need every 2nd-level table
    - Even when exist, 2nd-level tables can reside on disk if not in use

**Review: Two-level paging**

- Physical memory view
  - Stack
  - Heap
  - Code

- Virtual memory view
  - Stack
  - Heap
  - Code
Review: Address Translation Comparison

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation</td>
<td>Fast context switching: Segment mapping maintained by CPU</td>
<td>External fragmentation</td>
</tr>
<tr>
<td>Paging (single-level page)</td>
<td>No external fragmentation, fast easy allocation</td>
<td>Large table size ~ virtual memory</td>
</tr>
<tr>
<td>Paged segmentation</td>
<td>Table size ~ # of pages in virtual memory, fast easy allocation</td>
<td>Multiple memory references per page access</td>
</tr>
<tr>
<td>Two-level pages</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Caching Concept

- **Cache**: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching at different levels
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc...
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time = (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Example

- Data in memory, no cache:
  - Access time = 100ns
- Data in memory, 10ns cache:
  - Average Access time = (Hit Rate x HitTime) + (Miss Rate x MissTime)
  - HitRate + MissRate = 1

Review: Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology
**Why Does Caching Help? Locality!**

- **Temporal Locality** (Locality in Time):
  - Keep recently accessed data items closer to processor
- **Spatial Locality** (Locality in Space):
  - Move contiguous blocks to the upper levels

![Diagram of cache memory]

**Sources of Cache Misses**

- **Compulsory** (cold start): first reference to a block
  - “Cold” fact of life: not a whole lot you can do about it
  - Note: When running “billions” of instruction, Compulsory Misses are insignificant
- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- **Conflict** (collision):
  - Multiple memory locations mapped to same cache location
  - Solutions: increase cache size, or increase associativity
- Two others:
  - **Coherence** (Invalidation): other process (e.g., I/O) updates memory
  - **Policy**: Due to non-optimal replacement policy

---

**Caching Questions**

- 8 byte cache
- 32 byte memory
- 1 block = 1 byte
- Assume CPU accesses 01100

1. How do you know whether byte @ 01100 is cached?

![Cache diagram]

---

**Physical Memory**

- 8 byte cache
- 32 byte memory
- 1 block = 1 byte
- Assume CPU accesses 01100

1. How do you know whether byte @ 01100 is cached?
2. If not, at which location in the cache do you place the byte?
Cached to a single cache location.

1. How do you know whether byte @ 01100 is cached?
2. If not, at which location in the cache do you place the byte?
3. If cache full, which cached byte do you evict?

---

Simple Example: Direct Mapped Cache

- Each byte (block) in physical memory is cached to a single cache location
  - Least significant bits of address (last 3 bits) index the cache
  - (00100), (01100), (10100), (11100) cached to 100

1. How do you know which byte is cached?
   - Check tag associated with index 100
2. At which cache location do you place (01100)?
   - 100
3. If cache full, which cached byte do you evict?
   - 100

---

Simple Example: Fully Associative Cache

- Each byte can be stored at any location in the cache

1. How do you know which byte is cached?
   - Tag store entire address of cached byte
Simple Example: Fully Associative Cache

- Each byte can be stored at any location in the cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100</td>
<td>Green</td>
</tr>
</tbody>
</table>

1. How do you know whether (01100) is cached?
   - Check tag of all cache entries
2. At which cache location do you place (01100)?
   - Any
3. If cache full, which cached byte do you evict?
   - Specific eviction policy

Administrivia

- Project #1:
  - Design doc (submit proj1-final-design) and group evals (Google Docs form) due Wed 2/26 at 11:59PM
    - Group evals are anonymous to your group

- Midterm #1 is Wed March 12 4-5:30pm in 245 Li Ka Shing (A-L) and 105 Stanley (M-Z)
  - Closed book, double-sided handwritten page of notes, no calculators, smartphones, Google glass etc.
  - Covers lectures #1-12, readings, handouts, and projects 1 and 2
  - Review session: 245 Li Ka Shing Mon March 12 5:30-7:30pm

- Class feedback is always welcome!
  - https://www.surveymonkey.com/s/ZFDLLYL

Direct Mapped Cache

- Cache index selects a cache block
- "Byte select" selects byte within cache block
  - Example: Block Size=32B blocks
- Cache tag fully identifies the cached data
- Data with same “cache index” shares the same cache entry
  - Conflict misses

Cache Data

Valid Bit

Byte 0

Byte 31

Byte Select

Cache Index

Cache Tag

Ex: 0b01

5min Break
Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operate in parallel
- Example: Two-way set associative cache
  - Two tags in the set are compared in input in parallel
  - Data is selected based on the tag result

Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block

Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache
  - 32-Block Address Space:
  - Direct mapped: block 12 (01100) can go only into block 4 (12 mod 8)
  - Set associative: block 12 can go anywhere in set 0
  - Fully associative: block 12 can go anywhere

Which Block Should be Replaced on a Miss?

- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

Example TLB miss rates:

<table>
<thead>
<tr>
<th>Size</th>
<th>2-way LRU Random</th>
<th>4-way LRU Random</th>
<th>8-way LRU Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
What Happens on a Write?

- **Write through**: The information is written both to the block in the cache and to the block in the lower-level memory.
- **Write back**: The information is written only to the block in the cache.
  - Modified cache block is written to main memory only when it is replaced.
  - Question is block clean or dirty?
- Pros and Cons of each?
  - **WT**:
    - **PRO**: read misses cannot result in writes
    - **CON**: processor held up on writes unless writes buffered
  - **WB**:
    - **PRO**: repeated writes not sent to DRAM
    - **CON**: More complex

What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - If PTE valid, hardware fills TLB and processor never knows
    - If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables:
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - If PTE valid, fills TLB and returns from fault
    - If PTE marked as invalid, internally calls Page Fault handler
- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things
What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!

- Options?
  - Invalidate TLB: simple but might be expensive
    - What if switching frequently between processes?
  - Include ProcessID in TLB
    - This is an architectural solution: needs hardware

- What if translation tables change?
  - For example, to move page from memory to disk or vice versa...
  - Must invalidate TLB entry!
    - Otherwise, might think that page is still in memory!

What TLB organization makes sense?

- Needs to be really fast
  - Critical path of memory access
  - Seems to argue for Direct Mapped or Low Associativity

- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
    - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)

- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - First page of code, data, stack may map to same entry
    - Need 3-way associativity at least?
  - What if use high order bits as index?
    - TLB mostly unused for small programs

TLB organization: include protection

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity

- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info

- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"

- When does TLB lookup occur relative to memory cache access?
  - Before memory cache lookup?
  - In parallel with memory cache lookup?

Reducing translation time further

- As described, TLB lookup is in serial with cache lookup:

  - Machines with TLBs go one step further: they overlap TLB lookup with cache access.
    - Works because offset available early
Overlapping TLB & Cache Access (1/2)

- Main idea:
  - Offset in virtual address exactly covers the "cache index" and "byte select"
  - Thus can select the cached byte(s) in parallel to perform address translation

<table>
<thead>
<tr>
<th>virtual address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical address</td>
<td>tag / page #</td>
</tr>
</tbody>
</table>

Overlapping TLB & Cache Access (1/2)

- Here is how this might work with a 4K cache:

<table>
<thead>
<tr>
<th>32</th>
<th>TLB</th>
<th>assoc lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>index</td>
<td>4K Cache</td>
</tr>
<tr>
<td>10</td>
<td>PA</td>
<td>Data</td>
</tr>
<tr>
<td>2</td>
<td>disp</td>
<td>PA</td>
</tr>
<tr>
<td>4 bytes</td>
<td>Hi/ Miss</td>
<td>Hi/ Miss</td>
</tr>
</tbody>
</table>

- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252

Putting Everything Together: Address Translation

<table>
<thead>
<tr>
<th>Virtual Address:</th>
<th>Physical Address:</th>
<th>Physical Memory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
<td>Offset</td>
<td>Offset</td>
</tr>
<tr>
<td>PageTablePtr</td>
<td>Physical Page #</td>
<td>Offset</td>
</tr>
<tr>
<td>Page Table (1st level)</td>
<td>Page Table (2nd level)</td>
<td>Page Table (1st level)</td>
</tr>
</tbody>
</table>

Putting Everything Together: TLB

<table>
<thead>
<tr>
<th>Virtual Address:</th>
<th>Physical Address:</th>
<th>Physical Memory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
<td>Offset</td>
<td>Offset</td>
</tr>
<tr>
<td>PageTablePtr</td>
<td>Physical Page #</td>
<td>Offset</td>
</tr>
<tr>
<td>Page Table (1st level)</td>
<td>Page Table (2nd level)</td>
<td>Page Table (1st level)</td>
</tr>
<tr>
<td>TLB:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary (1/2)

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    » Temporal Locality: Locality in Time
    » Spatial Locality: Locality in Space

- Three (+1) Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices

Summary (2/2)

- Cache Organizations:
  - Direct Mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent

- TLB is cache on address translations
  - Fully associative to reduce conflicts
  - Can be overlapped with cache access