**Memory 3: Caching and TLBs (Con’t), Demand Paging**

March 10th, 2022
Prof. Anthony Joseph and John Kubiatowicz
http://cs162.eecs.Berkeley.edu

---

### How is the Translation Accomplished?

- The MMU must translate virtual address to physical address on:
  - Every instruction fetch
  - Every load
  - Every store
- What does the MMU need to do to translate an address?
  - 1-level Page Table
    - Read PTE from memory, check valid, merge address
    - Set “accessed” bit in PTE, Set “dirty bit” on write
  - 2-level Page Table
    - Read and check first level
    - Read, check, and update PTE
  - N-level Page Table …
- MMU does **page table Tree Traversal** to translate each address

---

### Where and What is the MMU ?

- The processor requests READ Virtual-Address to memory system
  - Through the MMU to the cache (to the memory)
- Some time later, the memory system responds with the data stored at the physical address (resulting from virtual → physical translation)
  - Fast on a cache hit, slow on a miss
- So what is the MMU doing?
- On every reference (I-fetch, Load, Store) read (multiple levels of) page table entries to get physical frame or FAULT
  - Through the caches to the memory
  - Then read/write the physical location
Recall: CS61c 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 underlies many techniques used today to make computers fast
  - 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 =
  \[(\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})\]

Recall: In Machine Structures (eg. 61C) …

- Caching is the key to memory system performance

  \[
  \text{Average Memory Access Time (AMAT)} = (\text{Hit Rate} \times \text{HitTime}) + (\text{Miss Rate} \times \text{MissTime})
  \]

  \[
  \text{Where HitRate} + \text{MissRate} = 1
  \]

  HitRate = 90% => AMAT = (0.9 x 1) + (0.1 x 101) = 11 ns

  HitRate = 99% => AMAT = (0.99 x 1) + (0.01 x 101) = 2.01 ns

  \[\text{MissTime}_{L1} \text{ includes HitTime}_{L1}\text{+MissPenalty}_{L1} = \text{HitTime}_{L1} + \text{AMAT}_{L2}\]

Another Major Reason to Deal with Caching

- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access?
- Solution? Cache translations!
  - **Translation Cache**: TLB ("Translation Lookaside Buffer")

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
Recall: Memory Hierarchy

- Caching: Take advantage of the principle of locality to:
  - Present the illusion of having as much memory as in the cheapest technology
  - Provide average speed similar to that offered by the fastest technology

<table>
<thead>
<tr>
<th>Core (Programs)</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>L3 Cache</th>
<th>Main Memory (DRAM)</th>
<th>Secondary Storage (SSD)</th>
<th>Secondary Storage (Disk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (ns):</td>
<td>0.3</td>
<td>1</td>
<td>3</td>
<td>10-30</td>
<td>100,000 (0.1 ms)</td>
<td>10,000,000 (15 ms)</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>100Bs</td>
<td>10kBs</td>
<td>100kBs</td>
<td>MBs</td>
<td>GBs</td>
<td>TBs</td>
</tr>
</tbody>
</table>

How do we make Address Translation Fast?

- Cache results of recent translations!
  - Different from a traditional cache
  - Cache Page Table Entries using Virtual Page # as the key

Translation Look-Aside Buffer

- Record recent Virtual Page # to Physical Frame # translation
- If present, have the physical address without reading any of the page tables!!!
  - Even if the translation involved multiple levels
  - Caches the end-to-end result
- Was invented by Sir Maurice Wilkes – prior to caches
  - When you come up with a new concept, you get to name it!
  - People realized “if it’s good for page tables, why not the rest of the data in memory?”
- On a TLB miss, the page tables may be cached, so only go to memory when both miss

Caching Applied to Address Translation

- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds
What kind of Cache for TLB?

- Remember all those cache design parameters and trade-offs?
  - Amount of Data = N * L * K
  - Tag is portion of address that identifies line (w/o line offset)
  - Write Policy (write-thru, write-back), Eviction Policy (LRU, ...)

How might organization of TLB differ from that of a conventional instruction or data cache?

- Let's do some review ...

A Summary on Sources of Cache Misses

- **Compulsory** (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "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 the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity

- **Coherence (Invalidation)**: other process (e.g., I/O) updates memory

How is a Block found in a Cache?

- **Block** is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don’t have data select field

- **Index** Used to Lookup Candidates in Cache
  - Index identifies the set

- **Tag** used to identify actual copy
  - If no candidates match, then declare cache miss
Review: Direct Mapped Cache

- **Direct Mapped** $2^N$ byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = $2^M$)
- Example: 1 KB Direct Mapped Cache with 32 B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block

Review: 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
  - Cache Index selects a “set” from the cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result

Review: 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 can go only into block 4 (12 mod 8)
    - Set associative: block 12 can go anywhere in set 0 (12 mod 4)
    - Fully associative: block 12 can go anywhere
### Lec 15.21

**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)

**Miss rates for a workload:**

<table>
<thead>
<tr>
<th>Size</th>
<th>LRU Random</th>
<th>LRU Random</th>
<th>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>

### Lec 15.22

**Write through vs Write back**

- **Write through:** The information is written to both 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
  - Read miss may require writeback of dirty data

### Lec 15.23

**Physically-Indexed vs Virtually-Indexed Caches**

- **Physically-Indexed Caches**
  - Address handed to cache after translation
  - Page Table holds physical addresses
  - Benefits:
    - Every piece of data has single place in cache
    - Cache can stay unchanged on context switch
  - Challenges:
    - TLB is in critical path of lookup!
    - Pretty Common today (e.g. x86 processors)

- **Virtually-Indexed Caches**
  - Address handed to cache before translation
  - Page Table holds virtual addresses (one option)
  - Benefits:
    - TLB not in critical path of lookup, so can be faster
  - Challenges:
    - Same data could be mapped in multiple places of cache
    - May need to flush cache on context switch

- We will stick with Physically Addressed Caches for now!

### Lec 15.24

**Administrivia**

- Prof Joseph’s office hours: Tuesdays 1-2pm and Thursdays 12-1 (Soda 447A)
- Project 2 design docs are due TOMORROW Friday 3/11
- Midterm 2: Coming up on next Thursday 3/17 7-9pm
  - Topics: up until Lecture 16: Scheduling, Deadlock, Address Translation, Virtual Memory, Caching, TLBs, Demand Paging
- Review Session: Wednesday 3/16 (Details TBA)
What TLB Organization Makes Sense?

- Needs to be really fast
  - Critical path of memory access
    » In simplest view: before the cache
    » Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high! (PT traversal)
  - Cost of Conflict (Miss Time) is high
    » Hit Time – dictated by clock cycle
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of virtual page number 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 (larger now)
  - Not very big, can support higher associativity
- Small TLBs 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”
- Example for MIPS R3000:

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFA00</td>
<td>0x0003</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>R/W 34</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x0010</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R 0</td>
</tr>
<tr>
<td>0x0041</td>
<td>0x0011</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R 0</td>
</tr>
</tbody>
</table>

Example: R3000 pipeline includes TLB “stages”

MIPS R3000 Pipeline

- Inst Fetch
- Dcache/Reg
- ALU / E.A
- Memory
- Write Reg
- TLB
- I-Cache
- RF
- Operation
- WB
- E.A
- TLB
- D-Cache

TLB
- 64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space

- 0xx User segment (caching based on PT/TLB entry)
- 100 Kernel physical space, cached
- 101 Kernel physical space, uncached
- 11x Kernel virtual space

Allows context switching among
- 64 user processes without TLB flush

Reducing translation time for physically-indexed caches

- As described, TLB lookup is in serial with cache lookup
  - Consequently, speed of TLB can impact speed of access to cache
- Machines with TLBs go one step further: overlap TLB lookup with cache access
  - Works because offset available early
  - 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

Virtual address: Virtual Page # Offset

Physical address: tag / page # index byte
Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:
  - What if cache size is increased to 8KB?
    - Overlap not complete
    - Need to do something else. See CS152/252
  - Another option: Virtual Caches would make this faster
    - Tags in cache are virtual addresses
    - Translation only happens on cache misses

Current Example: Memory Hierarchy

- Caches (all 64 B line size)
  - L1 I-Cache: 32 KiB/core, 8-way set assoc.
  - L1 D Cache: 32 KiB/core, 8-way set assoc., 4-5 cycles load-to-use, Write-back policy
  - L2 Cache: 1 MiB/core, 16-way set assoc., Inclusive, Write-back policy, 14 cycles latency
  - L3 Cache: 1.375 MiB/core, 11-way set assoc., shared across cores, Non-inclusive victim cache, Write-back policy, 50-70 cycles latency
- TLB
  - L1 ITLB, 128 entries; 8-way set assoc. for 4 KB pages
    - 8 entries per thread; fully associative, for 2 MiB / 4 MiB page
  - L1 DTLB 64 entries; 4-way set associative for 4 KB pages
    - 4 entries; 4-way associative, 2 MiB / 4 MiB page translations:
  - L2 STLB: 1536 entries; 12-way set assoc. 4 KiB + 2 MiB pages
    - 16 entries; 4-way set associative, 1 GiB page translations:

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!
    - Called “TLB Consistency”
- Aside: with Virtually-Indexed cache, need to flush cache!
  - Remember, everyone has their own version of the address “0”!
Putting Everything Together: Address Translation

Virtual Address:
- Offset
- Virtual P2 index
- Virtual P1 index
- Page Table Pointer

Page Table (1st level)
- Page Table (2nd level)

Physical Address:
- Offset
- Physical Page #

Putting Everything Together: TLB

Virtual Address:
- Offset
- Virtual P2 index
- Virtual P1 index

Page Table Pointer
- Page Table (1st level)
- Page Table (2nd level)
- Physical Page #

Putting Everything Together: Cache

Virtual Address:
- Offset
- Virtual P2 index
- Virtual P1 index

Page Table Pointer
- Page Table (1st level)
- Page Table (2nd level)

Physical Address:
- Offset
- Tag
- Index
- Byte

Cache: tag

TLB:

Putting Everything Together: TLB

Virtual Address:
- Offset
- Virtual P2 index
- Virtual P1 index

Page Table Pointer
- Page Table (1st level)
- Page Table (2nd level)
- Physical Page #

TLB:

Page Fault

- The Virtual-to-Physical Translation fails
  - PTE marked invalid, Priv. Level Violation, Access violation, or does not exist
  - Causes an Fault / Trap  
    » Not an interrupt because synchronous to instruction execution
  - May occur on instruction fetch or data access
  - Protection violations typically terminate the instruction
- Other Page Faults engage operating system to fix the situation and retry the instruction
  - Allocate an additional stack page, or
  - Make the page accessible - Copy on Write,
  - Bring page in from secondary storage to memory – demand paging
- Fundamental inversion of the hardware / software boundary
Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don’t use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user’s code to be in memory
- Solution: use main memory as “cache” for disk

```
Processor
  Control
  Datapath
  Second Level Cache (SRAM)
  Main Memory (DRAM)
  Secondary Storage (Disk)
  Tertiary Storage (Tape)
```

Page Fault \(\Rightarrow\) Demand Paging

```
Process
virtual address

Instruction

MMU

physical address

Page Fault Handler

Update PT entry

Load page from disk

Scheduler

retry

Exception

Frame

Offset

Load page from disk
```

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

Summary (2/2)

- “Translation Lookaside Buffer” (TLB)
  - Small number of PTEs and optional process IDs (< 512)
  - Often Fully Associative (Since conflict misses expensive)
  - On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  - On change in page table, TLB entries must be invalidated
- Demand Paging: Treating the DRAM as a cache on disk
  - Page table tracks which pages are in memory
  - Any attempt to access a page that is not in memory generates a page fault, which causes OS to bring missing page into memory
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past