Recall: 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

Recall: Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates 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

Recall: 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 (block = 1 byte)

32-Block Address Space:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

- **Direct mapped:** block 12 can go only into block 4 (12 mod 8)
  - Tag: 9b
  - Index: 3b

- **Set associative:** block 12 can go anywhere in set 0 (12 mod 4)
  - Tag: 10b
  - Index: 2b

- **Fully associative:** block 12 can go anywhere
  - Tag: 12b

**Review: 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>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>

**Review: What happens on a write?**

- **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

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

- Caching is the key to memory system performance

```
Processor <-> Second Level Cache (SRAM) <-> Main Memory (DRAM)
```

Average Access time = (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

<table>
<thead>
<tr>
<th></th>
<th>100ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Access time = 100ns</td>
</tr>
<tr>
<td>Processor</td>
<td>Second Level Cache (SRAM)</td>
</tr>
</tbody>
</table>

Hit Rate + Miss Rate = 1

Hit Rate = 90% => Average Access Time = 10.9 ns
Hit Rate = 99% => Average Access Time = 10.9 ns
Recall: 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

Recall: What Actually Happens on a TLB Miss? (1/3)

• 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

Recall: What Actually Happens on a TLB Miss? (2/3)

• Software traversed Page tables (like MIPS)
  – 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

Recall: What Actually Happens on a TLB Miss? (3/3)

• Most chip sets provide hardware traversal
  – Modern operating systems tend to have more TLB faults since they use translation for many things
  – Examples:
    » shared segments
    » user-level portions of an operating system
**Transparent Exceptions: TLB/Page fault (1/2)**

- **How to transparently restart faulting instructions?**
  - (Consider load or store that gets TLB or Page fault)
  - Could we just skip faulting instruction?
    » No: need to perform load or store after reconnecting physical page

**Consider weird things that can happen**

- What if an instruction has side effects?
  - Options:
    » Unwind side-effects (easy to restart)
    » Finish off side-effects (messy?)
  - Example 1: `mov (sp)+,10`
    » What if page fault occurs when write to stack pointer?
    » Did `sp` get incremented before or after the page fault?
  - Example 2: `strcpy (r1), (r2)`
    » Source and destination overlap: can't unwind in principle!
    » IBM 5/370 and VAX solution: execute twice – once read-only
- What about “RISC” processors?
  - For instance delayed branches?
    » Example: `bne someplace`
    » Precise exception state consists of two PCs: PC and nPC (next PC)
  - Delayed exceptions:
    » Example: `div r1, r2, r3`
    » What if takes many cycles to discover divide by zero, but load has already caused page fault?

**Transparent Exceptions: TLB/Page fault (2/2)**

- Hardware must help out by saving:
  - Faulting instruction and partial state
    » Need to know which instruction caused fault
    » Is single PC sufficient to identify faulting position????
  - Processor State: sufficient to restart user thread
    » Save/restore registers, stack, etc
- What if an instruction has side-effects?

**Precise Exceptions**

- Precise: state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position
- Imprecise: system software has to figure out what is where and put it all back together
- Performance goals often lead to forsaking precise interrupts
  - System software developers, user, markets etc. usually wish they had not done this
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts
Recall: TLB Organization

• 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!
  – 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

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

virtual address: Virtual Page # Offset
physical address: tag / page # index byte

Overlapping TLB & Cache Access (2/2)

• 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
    – Tags in cache are virtual addresses
    – Translation only happens on cache misses

Reducing translation time further

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

Virtual Address

Physical Address

• Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  – Works because offset available early

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
    – Tags in cache are virtual addresses
    – Translation only happens on cache misses
Putting Everything Together: Address Translation

Physical Address: Offset

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

Virtual Address: Offset

Page TablePtr

Putting Everything Together: TLB

Physical Address: Offset

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

Virtual Address: Offset

Page TablePtr

Putting Everything Together: Cache

Physical Address: Offset

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

Virtual Address: Offset

Page TablePtr

Next Up: What happens when ...

Process

virtual address

MMU

PT

frame#

generic

offset

frame#

Load page from disk

scheduler

Exception

Page Fault Handler

update PT entry

retry

page fault

load page from disk
**Administrivia**

- Midterm 2 coming up on **Tue 3/21 7:00-8:30PM**
  - All topics up to and including Lecture 15
    » Focus will be on Lectures 11 – 15 and associated readings
    » Projects 1 and 2
    » Homework 0 – 2
  - Closed book
  - 2 pages hand-written notes both sides
  - Room assignment
    » A-H 1 100 Genetics and Plant Biology Building, I-Z 1 Pimentel

- Ion out of Wednesday (3/15) at NSF in Washington, DC
  - Nathan will teach the lecture

---

**Where are all places that caching arises in OSes?**

- Direct use of caching techniques
  - TLB (cache of PTEs)
  - Paged virtual memory (memory as cache for disk)
  - File systems (cache disk blocks in memory)
  - DNS (cache hostname => IP address translations)
  - Web proxies (cache recently accessed pages)

- Which pages to keep in memory?
  - All-important “Policy” aspect of virtual memory
  - Will spend a bit more time on this in a moment

---

**Impact of caches on Operating Systems (1/2)**

- Indirect - dealing with cache effects (e.g., sync state across levels)
  - Maintaining the correctness of various caches
  - E.g., TLB consistency:
    » With PT across context switches ?
    » Across updates to the PT ?

- Process scheduling
  - Which and how many processes are active? Priorities?
  - Large memory footprints versus small ones?
  - Shared pages mapped into VAS of multiple processes?
Impact of caches on Operating Systems (2/2)

• Impact of thread scheduling on cache performance
  – Rapid interleaving of threads (small quantum) may degrade
    cache performance
    » Increase average memory access time (AMAT) !!!

• Designing operating system data structures for cache
  performance

Working Set Model

• As a program executes it transitions through a sequence of
  “working sets” consisting of varying sized subsets of the address
  space

Cache Behavior under WS model

• Amortized by fraction of time the Working Set is active
• Transitions from one WS to the next
• Capacity, Conflict, Compulsory misses
• Applicable to memory caches and pages. Others?

Another model of Locality: Zipf

• Likelihood of accessing item of rank r is $\alpha \frac{1}{r^2}$
• Although rare to access items below the top few, there are so many
  that it yields a “heavy tailed” distribution
• Substantial value from even a tiny cache
• Substantial misses from even a very large cache
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

Illusion of Infinite Memory (1/2)

- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
  » More programs fit into memory, allowing more concurrency

Illusion of Infinite Memory (2/2)

- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
  » Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
  » Performance issue, not correctness issue

Since Demand Paging is Caching, Must Ask…

- What is block size?
  - 1 page
- What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
  - Fully associative: arbitrary virtual → physical mapping
- How do we find a page in the cache when look for it?
  - First check TLB, then page-table traversal
- What is page replacement policy? (i.e. LRU, Random…)
  - This requires more explanation… (kinda LRU)
- What happens on a miss?
  - Go to lower level to fill miss (i.e. disk)
- What happens on a write? (write-through, write back)
  - Definitely write-back – need dirty bit!
Recall: What is in a Page Table Entry

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called “Directories”

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- P: Present (same as “valid” bit in other architectures)
- W: Writeable
- U: User accessible
- PWT: Page write transparent: external cache write-through
- PCD: Page cache disabled (page cannot be cached)
- A: Accessed: page has been accessed recently
- D: Dirty (PTE only): page has been modified recently
- L=1⇒4MB page (directory only).
  - Bottom 22 bits of virtual address serve as offset

Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - Resulting trap is a “Page Fault”
      - What does OS do on a Page Fault:
        » Choose an old page to replace
        » If old page modified ("D=1"), write contents back to disk
        » Change its PTE and any cached TLB to be invalid
        » Load new page into memory from disk
        » Update page table entry, invalidate TLB for new entry
        » Continue thread from original faulting location
        - TLB for new page will be loaded when thread continued!
        - While pulling pages off disk for one process, OS runs another process from ready queue
        » Suspended process sits on wait queue

Summary: Steps in Handling a Page Fault

- A cache of translations called “Translation Lookaside Buffer” (TLB)
  - Relatively small number of PTEs and optional process IDs (< 512)
  - 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
  - TLB is logically in front of cache (need to overlap with cache access)
- Precise Exception specifies a single instruction for which:
  - All previous instructions have completed (committed state)
  - No following instructions nor actual instruction have started
- Can manage caches in hardware or software or both
  - Goal is highest hit rate, even if it means more complex cache management