CS162
Operating Systems and
Systems Programming
Lecture 14
Caching and Demand Paging

March 13, 2006
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Memory Hierarchy of a Modern Computer System
• 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

Review: A Summary on 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

Review: 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
Review: Other Caching Questions

- What line gets replaced on cache miss?
  - Easy for Direct Mapped: Only one possibility
  - Set Associative or Fully Associative:
    » Random
    » LRU (Least Recently Used)
- What happens on a write?
  - Write through: The information is written to both 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?

Goals for Today

- Finish discussion of TLBs
- Concept of Paging to Disk
- Page Faults and TLB Faults
- Precise Interrupts
- Page Replacement Policies

Quick Aside: Protection without Hardware

- Does protection require hardware support for translation and dual-mode behavior?
  - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
- Protection via Strong Typing
  - Restrict programming language so that you can’t express program that would trash another program
  - Loader needs to make sure that program produced by valid compiler or all bets are off
  - Example languages: LISP, Ada, Modula-3 and Java
- Protection via software fault isolation:
  - Language independent approach: have compiler generate object code that provably can’t step out of bounds
    » Compiler puts in checks for every “dangerous” operation (loads, stores, etc). Again, need special loader.
    » Alternative, compiler generates “proof” that code cannot do certain things (Proof Carrying Code)
  - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)

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
TLB organization

- 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”

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</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFA00</td>
<td>0x0003</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>R/W</td>
<td>34</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x0010</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>0x0041</td>
<td>0x0011</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>

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

Example: R3000 pipeline includes TLB “stages”

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Dec/ Reg</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>E.A.</td>
</tr>
</tbody>
</table>

MIPS R3000 Pipeline

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

<table>
<thead>
<tr>
<th>ASID</th>
<th>V. Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

- 0x 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

Overlapping TLB & Cache Access

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

- Exam Update
- Make sure to come to sections!
  - There will be a lot of information about the projects that I cannot cover in class
  - Also supplemental information and detail that we don’t have time for in class

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

Demand Paging is Caching

- 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!
Review: What is in a PTE?

- 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</th>
<th>Free (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>S</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Physical Page Number)</td>
<td></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>
</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: 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 \(\Rightarrow\) Page in memory, PTE points at physical page
  - Not Valid \(\Rightarrow\) 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

Software-Loaded TLB

- MIPS/Snake/Nachos TLB is loaded by software
  - High TLB hit rate \(\Rightarrow\) ok to trap to software to fill the TLB, even if slower
  - Simpler hardware and added flexibility: software can maintain translation tables in whatever convenient format
- How can a process run without access to page table?
  - Fast path (TLB hit with valid=1):
    » Translation to physical page done by hardware
  - Slow path (TLB hit with valid=0 or TLB miss)
    » Hardware receives a "TLB Fault"
- What does OS do on a TLB Fault?
  » Traverse page table to find appropriate PTE
  » If valid=1, load page table entry into TLB, continue thread
  » If valid=0, perform "Page Fault" detailed previously
  » Continue thread
- Everything is transparent to the user process:
  » It doesn’t know about paging to/from disk
  » It doesn’t even know about software TLB handling

Transparent Exceptions

- How to transparently restart faulting instructions?
  » Could we just skip it?
    » No: need to perform load or store after reconnecting physical page
  » 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?
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 S/370 and VAX solution: execute twice - once read-only

- What about “RISC” processors?
  - For instance delayed branches?
    » Example: `bne somewhere`
      1d r1, (sp)
    » Precise exception state consists of two PCs: PC and nPC
  - Delayed exceptions:
    » Example: `div r1, r2, r3`
      1d r1, (sp)
    » What if takes many cycles to discover divide by zero, but load has already caused page fault?

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 designers to forsake 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

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    » The cost of being wrong is high: must go to disk
    » Must keep important pages in memory, not toss them out

- What about MIN?
  - Replace page that won’t be used for the longest time
  - Great, but can’t really know future ...
  - Makes good comparison case, however

- What about RANDOM?
  - Pick random page for every replacement
    » Typical solution for TLB’s. Simple hardware
    » Pretty unpredictable - makes it hard to make real-time guarantees

- What about FIFO?
  - Throw out oldest page. Be fair - let every page live in memory for some amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages

Replacement Policies (Con’t)

- What about LRU?
  - Replace page that hasn’t been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.

- How to implement LRU? Use a list!
  - On each use, remove page from list and place at head
  - LRU page is at tail

- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list ...
  - Many instructions for each hardware access

- In practice, people approximate LRU (more later)
Summary

- TLB is cache on translations
  - Fully associative to reduce conflicts
  - Can be overlapped with cache access
- Demand Paging:
  - Treat memory as cache on disk
  - Cache miss ⇒ get page from disk
- Transparent Level of Indirection
  - User program is unaware of activities of OS behind scenes
  - Data can be moved without affecting application correctness
- Software-loaded TLB
  - Fast Path: handled in hardware (TLB hit with valid=1)
  - Slow Path: Trap to software to scan page table
- Precise Exception specifies a single instruction for which:
  - All previous instructions have completed (committed state)
  - No following instructions nor actual instruction have started
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: replace page that will be used farthest in future
  - LRU: Replace page that hasn’t be used for the longest time