CS162
Operating Systems and Systems Programming
Lecture 14
Caching and Demand Paging
March 4, 2010
Ion Stoica
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

<table>
<thead>
<tr>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
</tr>
<tr>
<td>Datapath</td>
</tr>
<tr>
<td>Cache</td>
</tr>
<tr>
<td>Main Memory (DRAM)</td>
</tr>
<tr>
<td>Secondary Storage (Disk)</td>
</tr>
<tr>
<td>Tertiary Storage (Tape)</td>
</tr>
</tbody>
</table>

- Speed (ns): 1s, 10s-100s, 100s, 10,000,000s (10s-100s, 10s sec)
- Size (bytes): 1s, Ks-Ms, Ms, Gs, Ts

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
  - HitRate = 90% → Average Access Time = 19ns
  - HitRate = 99% → Average Access Time = 10.9ns

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: 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

Review: Where does a Block Get Placed in a Cache?

- **Example**: Block 12 placed in 8 block cache

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)

Goals for Today

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

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.
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

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 (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
- **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

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!
Administrative

- Midterm next week:
  - Tuesday, 3/9, 3:30-6:30pm, 277 Cory Hall
  - Should be 2 hour exam with extra time
  - Closed book, one page of hand-written notes (both sides)
- No class on day of Midterm
  - Extra Office Hours: Tuesday 10-11am and 1:00-3:00pm
- Midterm Topics
  - Topics: Everything up to today (3/4)
  - History, Concurrency, Multithreading, Synchronization, Protection/Address Spaces, TLBs
- Project 2
  - Initial Design Document due today (Thursday 3/4)
  - Look at the lecture schedule to keep up with due dates!

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!
  - 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
- 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>

- 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?
  - Before cache lookup?
  - In parallel with cache lookup?

Reducing translation time further

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

Virtual Address

<table>
<thead>
<tr>
<th></th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y page no.</td>
<td>10</td>
</tr>
</tbody>
</table>

TLB Lookup

<table>
<thead>
<tr>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

Physical Address

<table>
<thead>
<tr>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

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

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

- 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
- 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

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!
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

Software-Loaded TLB

- MIPS/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
      » ld r1, (sp)`
    » Precise exception state consists of two PCs: PC and nPC
    » Delayed exceptions:
      » Example: `div r1, r2, r3`
      » ld r1, (sp)`
      » What if takes many cycles to discover divide by zero, but load has already caused page fault?
Precise Exceptions

- Precise $\Rightarrow$ 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 $\Rightarrow$ 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 same 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 $\Rightarrow$ 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 been used for the longest time