CS162
Operating Systems and
Systems Programming
Lecture 14

Caching (Finished),
Demand Paging

October 15th, 2017

Ion Stoica
http://cs162.eecs.Berkeley.edu
Recall: In Machine Structures (eg. 61C) …

- Caching is the key to memory system performance

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

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

\[
\text{HitRate} = 90\% \implies \text{Avg. Access Time} = (0.9 \times 10) + (0.1 \times 100) = 19\text{ns}
\]

\[
\text{HitRate} = 99\% \implies \text{Avg. Access Time} = (0.99 \times 10) + (0.01 \times 100) = 10.9\text{ns}
\]
### 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

<table>
<thead>
<tr>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Ex: 0x50</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid Bit</th>
<th>Cache Tag</th>
<th>Ex: 0x50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 31</td>
</tr>
<tr>
<td>Byte 63</td>
</tr>
<tr>
<td>Byte 1023</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>31</td>
</tr>
</tbody>
</table>
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
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

```
0 4 31

Cache Tag

Cache Data
Cache Index
Byte Select

Valid

Cache Tag

Cache Block 0

Cache Data
Cache Block 0

Cache Tag

Valid

Compare

Sel1

Mux 0 Sel0

OR

Hit

Cache Block
```
**Where does a Block Get Placed in a Cache?**

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

- **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
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</th>
<th>2-way Random</th>
<th>4-way LRU</th>
<th>4-way Random</th>
<th>8-way LRU</th>
<th>8-way Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
<td>5.3%</td>
<td>4.4%</td>
<td>5.0%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
<td>1.7%</td>
<td>1.4%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
<td>1.13%</td>
<td>1.12%</td>
<td>1.12%</td>
</tr>
</tbody>
</table>
What happens on a write?

• Write through (WT): The information is written to both the block in the cache and to the block in the lower-level memory.
  – Modified cache block is written to main memory only when it is replaced.
  – Question is block clean or dirty?
• Write back (WB): The information is written only to the block in the cache.

  – 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 processor not held up on writes
    » CON: More complex
      Read miss may require writeback of dirty data
• 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?

- 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

- 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

- 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
• 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
Transparent Exceptions: TLB/Page fault (2/2)

User

TLB Faults

OS

- 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: \texttt{mov (sp)+,10}
    » What if page fault occurs when write to stack pointer?
    » Did \texttt{sp} get incremented before or after the page fault?
  – Example 2: \texttt{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: \texttt{bne somewhere}
      \texttt{ld r1,(sp)}
    » Precise exception state consists of two PCs: PC and nPC (next PC)
  – Delayed exceptions:
    » Example: \texttt{div r1, r2, r3}
      \texttt{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 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
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>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Page #</td>
<td>tag / page #</td>
</tr>
<tr>
<td>Offset</td>
<td>index</td>
</tr>
<tr>
<td></td>
<td>byte</td>
</tr>
</tbody>
</table>
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

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

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

Page TablePtr

Physical Address:
- Physical Page #
- Offset

Physical Memory:
Putting Everything Together: TLB

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

Page Table (1st level):

Page Table (2nd level):

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:
Putting Everything Together: Cache

Virtual Address:

- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

- Page TablePtr

Page Table (2nd level)

TLB:

Physical Address:

- Physical Page #
- Offset

- tag
- index
- byte

cache:

- tag:
- block:

Physical Memory:
BREAK
Next Up: What happens when …

Process

virtual address

instruction

MMU

physical address

type

PT

frame#

offset

frame#

offset

update PT entry

Page Fault Handler

Scheduler

retry

exception

Page Fault Handler

load page from disk

Operating System
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

$P \text{ access}(\text{rank}) = \frac{1}{\text{rank}}$

- Likelihood of accessing item of rank $r$ is $\alpha \frac{1}{r^a}$
- 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!
Summary: Steps in Handling a Page Fault

1. Reference
2. Trap
3. Page is on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction

Diagram:
- Load M
- Operating system
- Page table
- Free frame
- Physical memory
Summary

• A cache of translations called a “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