Caching in Operating Systems Design & Systems Programming

David E. Culler
CS162 – Operating Systems and Systems Programming
Lecture 17
October 8, 2014

Reading: A&D 9.1-5,7
HW 3 due monday
Proj 1 submit today
**In Machine Structures (eg. 61C) ...**

- Caching is the key to memory system performance

![Diagram showing memory hierarchy with processor, main memory (DRAM), and second level cache (SRAM) connections.](image)

- Average Access time = \((\text{Hit Rate} \times \text{HitTime}) + (\text{Miss Rate} \times \text{MissTime})\)
- \(\text{HitRate} + \text{MissRate} = 1\)
- \(\text{HitRate} = 90\% \Rightarrow \text{Average Access Time} = 19 \text{ ns}\)
- \(\text{HitRate} = 99\% \Rightarrow \text{Average Access Time} = 10.9\text{ ns}\)
Review: Memory Hierarchy

- 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>
<th>Main Memory (DRAM)</th>
<th>Secondary Storage (SSD)</th>
<th>Secondary Storage (Disk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Registers</td>
<td>L1 Cache</td>
<td>L2 Cache</td>
<td>L3 Cache (shared)</td>
</tr>
<tr>
<td>Core Registers</td>
<td>L1 Cache</td>
<td>L2 Cache</td>
<td></td>
</tr>
</tbody>
</table>

- Speed (ns): 0.3 1 3 10-30 100 100,000 (0.1 ms) 10,000,000 (10 ms)
- Size (bytes): 100Bs 10kBs 100kBs MBs GBs 100GBs TBs
Why Does Caching Work? 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
Design issues for caches

• In Computer Architecture we are focused on cache design as a transparent memory accelerator
  – reduce average MAT (latency), increase BW
• implemented directly in hardware
• Issues:
  – cache size
  – block size
  – associativity (direct mapped, set assoc, fully assoc)
  – placement, replacement
  – number of levels of caches
• trade-offs among all of these
Quick Review of 61C Caches

Review: Direct-Mapped Cache

- All fields are read as unsigned integers.
- **Index**
  - specifies the cache index (or "row"/block)
- **Tag**
  - distinguishes between the addresses that map to the same location
- **Offset**
  - specifies which byte within the block we want

```
+-------------------+-------------------+-------------------+
| tag               | index             | byte              |
| to check if have  | to select within  |
| correct block     | block             |
```

TIO Dan’s great cache mnemonic

```
AREA (cache size, B) = \(2^{2(H+W)} = 2^H \times 2^W\)
```

Caching Terminology

- When reading memory, 3 things can happen:
  - **cache hit**: cache block is valid and contains proper address, so read desired word
  - **cache miss**: nothing in cache in appropriate block, so fetch from memory
  - **cache miss, block replacement**: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

Cache Terms

- **Hit rate**: fraction of access that hit in the cache
- **Miss rate**: 1 – Hit rate
- **Miss penalty**: time to replace a block from lower level in memory hierarchy to cache
- **Hit time**: time to access cache memory (including tag comparison)

```
Abbreviation: “$” = cache (A Berkeley innovation!)
```
Direct Mapped Cache

- Cache index selects a cache block
- “Byte select” selects byte within cache block
  - Example: Block Size=32B blocks
- Cache tag fully identifies the cached data
- Data with same “cache index” shares the same cache entry
  - Conflict misses
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**
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result

```
  31 8 4 0
  |   |   |
  |   |   |
  |   |   |
  |   |   |
  |   |   |

Cache Tag  Cache Index  Byte Select

Valid  Cache Tag  Cache Data  Cache Data  Cache Tag  Valid

Cache Block 0  Cache Block 0
```

- **Compare**
- **Sel1**
- **Mux**
- **Sel0**
- **OR**
- **Hit**
- **Cache Block**
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
Cache Design Issues

• Organization
  – cache size, block size
  – 1-way, n-way, associative

• Write Policy
  – write-through, write-back

• Replacement policy
  – given n-way associativity, which of the n gets replaced
  – FIFO, Random, LRU, Clock

• Coherence Policy (multi-processor)
  – write-invalidate, write-update
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 lots

Each TLB entry is for a whole page of blocks !!!
Where does caching arise in Operating Systems?
# Hardware Design Trade-offs

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Associativity</th>
<th>Time Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>100Bs</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>10kBs</td>
<td></td>
<td>100,000 (0.1 ms)</td>
</tr>
<tr>
<td>100kB</td>
<td></td>
<td>10,000,000 (10 ms)</td>
</tr>
</tbody>
</table>

### Block Size:

### Associativity:

### Time Constraint:
Where does caching arise in Operating Systems?

• Direct use of caching techniques
  – paged virtual memory (mem as cache for disk)
  – TLB (cache of PTEs)
  – 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?
Where does caching arise in Operating Systems?

• Indirect - dealing with cache effects

• Process scheduling
  – which and how many processes are active?
  – large memory footprints versus small ones?
  – priorities?

• Impact of thread scheduling on cache performance
  – rapid interleaving of threads (small quantum) may degrade cache performance
    • increase ave MAT !!!

• Designing operating system data structures for cache performance
Where does caching arise in Operating Systems?

• Maintaining the correctness of various caches

• TLB consistent with PT across context switches?

• Across updates to the PT?

• Shared pages mapped into VAS of multiple processes?
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 WS is active
- Transitions from one WS to the next
- Capacity, Conflict, Compulsory misses
- Applicable to memory caches and pages. Others?

Hit Rate vs. Cache Size

- New working set fits

10/8/14
Another model of Locality: Zipf

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

- Likelihood of accessing item of rank $r$ is $\alpha_1/r^a$
- 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 one
Going into detail on TLB
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
  – 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
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!
What TLB organization makes sense?

- Needs to be really fast
  - Critical path of memory access
  - 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

• 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 relative to memory cache access?
  – Before memory cache lookup?
  – In parallel with memory cache lookup?
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
Here is how this might work with a 4K cache:

- Overlap not complete
- Need to do something else. See CS152/252
Putting Everything Together: Address Translation

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

Page Table (1st level)

Page Table (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:

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

TLB:

Physical Memory:

- Physical Address:
  - Physical Page #
  - Offset

Putting Everything Together: TLB

Virtual Address:

- Virtual P1 index
- Virtual P2 index
- Offset

Page Table:

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

TLB:

Physical Memory:

- Physical Address:
  - Physical Page #
  - Offset
Putting Everything Together: Cache

Virtual Address:

Page Table
(1st level)

Page Table
(2nd level)

Page TablePtr

Physical Memory:

Physical Address:

Tag
Index
Byte

Cache:

Tag
Block

TLB: