CS 261 Fall 2021

Mike Lam, Professor



Caching

(get it??)

Topics

- Caching
- Cache implementations
- Cache policies
- Cache performance
- Performance improvement strategies

Motivation

- Caching is ubiquitous in modern computing:
 - L1-L3 memory
 - TLB and virtual memory (next week)
 - Disk controller buffers
 - Network controller buffers
 - Browser caches
 - Content delivery networks

Caching

- A cache is a small, fast memory that acts as a buffer or staging area for a larger, slower memory
 - Fundamental CS system design concept
 - Data is transferred in blocks or lines
 - Slower caches use larger block sizes
 - Cache hit vs. cache miss
 - Hit ratio: # hits / # memory accesses



Cache implementations

- What data structure can we use to implement caches?
 - Need **FAST** lookups and containment checks
 - From CS 240: use a hash table!
 - Cache slot = "real address" % CACHE_SIZE

Multiple addresses may map to the same cache slot! (this is called a conflict)



- Suppose we have a sixteen-element cache, with slots labeled starting at zero. Which slot would we use to store a cached version of a data element stored at address 0x4d6?
 - Reminder: cache slot = "real address" % CACHE_SIZE
 - ⁻ Hint: $2^4 = 16$, and four bits = one hex digit

Cache implementations

- A cache line is a block or sequence of bytes that is moved between memory levels in a single operation
- A cache set is a collection of one or more cache lines
 - Each cache line contains a tag to identify the source address and a valid flag/bit indicating whether the value is up-to-date



Cache size: $C = B \times E \times S$ data bytes

Cache implementations

- General cache organization:
 - **S** = # of cache sets = 2^{s}
 - s = # of bits for set index
 - **E** = # of lines per cache set
 - Level of associativity
 - **B** = block (cache line) size = 2^b
 - · Essentially bytes per line
 - b = # of bits for block offset
 - **m** = # of bits for memory address
 - M = size of memory in bytes = 2^m
 - C = total cache capacity = $S \times E \times B$
 - sets x lines/set x bytes/line
 - t = # of tag bits = m s b



Cache size: $C = B \times E \times S$ data bytes



Types of caches

- Direct-mapped (E = 1)
 - One line per set

- Set-associative (1 < E < C/B)
 - Multiple lines per set

- Fully-associative (E = C/B)
 - All lines in one set







Cache implementations

• Direct-mapped (E = 1) caches





- Suppose we have a direct-mapped cache (S=16, B=1), with sets labeled starting at zero. Which set would we use to store a cached version of a data element stored at address 0x4d6?
 - Hint: S=16 so the number of bits for the set index is four
 - Hint: B=1 so the number of bits for the block offset is zero



Cache implementations

• Set-associative (1 < E < C/B) caches





• Suppose we have a four-way set-associative cache (S=16, E=4, B=1), with sets labeled starting at zero. Which set would we use to store a cached version of a data element stored at address 0x4d6?

Cache implementations

• Fully-associative (E = C/B) caches





• Suppose we have a fully-associative cache (B=1) with sets labeled starting at zero. Which set would we use to store a cached version of a data element stored at address 0x4d6?

Cache implementations

- In general, we use the middle bits for the set index
 - Contiguous memory blocks should map to different cache sets



Cache misses ("Three C's")

- Compulsory / cold miss
 - First cache miss due to an "empty" cache
 - As the cache loads data, it is warmed up
- Conflict miss
 - Cache miss due to multiple lines in working set mapping to the same cache line
 - Repeated conflict misses for the same cache lines or blocks is called thrashing
- Capacity miss
 - The working set (amount of memory accessed in a given time interval) is too large to fit in cache

Cache policies

- If a cache set is full, a cache miss in that set requires lines to be replaced or evicted
- Policies:
 - Random replacement
 - Least recently used
 - Least frequently used
- These policies require additional overhead
 - More important for lower levels of the memory hierarchy

Cache policies

- How should we handle writes to a cached value?
 - Write-through: immediately update to lower level
 - Typically used for higher levels of memory hierarchy
 - Write-back: defer update until replacement/eviction
 - Typically used for lower levels of memory hierarchy
- How should we handle write misses?
 - Write-allocate: load then update
 - Typically used for write-back caches
 - No-write-allocate: update without loading
 - Typically used for write-through caches

Performance impact

• Metrics

- Hit rate/ratio: # hits / # memory accesses (1 miss rate)
 - Hit time: delay in accessing data for a cache hit
- Miss rate/ratio: # misses / # memory accesses
 - Miss penalty: delay in loading data for a cache miss
- Read throughput (or "bandwidth"): the rate that a program reads data from a memory system
- General observations:
 - Larger cache = higher hit rate but higher hit time
 - Lower miss rates = higher read throughput

Case study: matrix multiply





Case study: matrix multiply



Optimization strategies

- Focus on the common cases
- Focus on the code regions that dominate runtime
- Focus on inner loops and minimize cache misses
- Favor repeated local accesses (temporal locality)
- Favor stride-1 access patterns (spatial locality)

Tip: You can use Valgrind to detect cache misses (look up a tool called cachegrind)

Core theme

- Cache system design involves tradeoffs
 - Larger caches => higher hit rate but higher hit time
 - Size vs. speed
 - Larger blocks => higher hit rate for programs with good spatial locality, but lower hit rate for others
 - Favor spatial vs. temporal locality
 - Higher associativity => lower chance of thrashing but expensive to implement w/ possibly increased hit time
 - Hit time vs. miss penalty
 - More writes => simpler to implement but lower performance
 - Write-through vs. write-back

Next time

- Virtual memory: an OS-level memory cache
 - Bridge between module 4 (machine architectures) and module 5 (operating systems)

Cache architecture

- Example: Intel Core i7
- Per-core:
 - Registers
 - L1 d-cache and i-cache
 - Data and instructions
 - L2 unified cache
- Shared:
 - L3 unified cache
 - Main memory



- As the **working set size** of a loop decreases, what generally happens to the read throughput?
 - A) It increases
 - B) It decreases
 - C) It remains the same
 - D) There is no correlation
 - E) Not enough information to determine

Temporal locality

• Working set size vs. throughput



- As the **stride** of a loop increases, what generally happens to the read throughput?
 - A) It increases
 - B) It decreases
 - C) It remains the same
 - D) There is no correlation
 - E) Not enough information to determine

Spatial locality

• Stride vs. throughput



Memory mountain (CS:APP)

• Stride and WSS vs. read throughput



Memory mountain (stu, 2017)



Output of **Iscpu**:

Architecture:	x86_64
Byte Order:	Little Endian
CPU(s):	24
Thread(s) per core:	2
Core(s) per socket:	6
Socket(s):	2
Vendor ID:	Intel
Model name:	
<pre>Intel(R) Xeon(R) CPU</pre>	E5-2640
CPU max MHz:	3000.0000
CPU min MHz:	1200.0000
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	15360K

Memory mountain (stu, 2018)



Output of **Iscpu**:

Architecture:	x86_64
Byte Order:	Little Endian
CPU(s):	48
Thread(s) per core:	2
Core(s) per socket:	12
Socket(s):	2
Vendor ID:	Intel
Model name:	
<pre>Intel(R) Xeon(R) CPU</pre>	E5-2680
CPU max MHz:	3300.0000
CPU min MHz:	1200.0000
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	30720K

Memory mountain (stu, 2021)



Output of **Iscpu**:

x86_64
Little Endian
48
2
12
2
Intel
E5-2680 V3
3300.0000
1200.0000
768K
768K
6M
60M

Note: new per-user resource limits put in place Fall 2021 may be interfering

- Assume the following cache:
 - **S** = **8** sets (so s=3 bits for set index)
 - **E** = 1 line per set (so direct-mapped)
 - **B** = 4 bytes per line (so b=2 bits for block offset)
- What is the set index, tag, and block offset for address 227?
 - Hint: 227 in binary is 11100011



- Assume the following cache:
 - **S = 8** sets (so s=3 bits for set index)
 - **E** = 1 line per set (so direct-mapped)
 - **B** = 4 bytes per line (so b=2 bits for block offset)
- Address 227 (binary: 11100011)
 - Set index = 000_2 (0)
 - $\text{Tag} = 111_2$ (7)
 - Block offset = 11_2 (3)
 - Is this a hit?

No! Need to load the line into cache:

Set	Valid	Tag	block[0]	block[1]	block[2]	block[3]
0 (000)	1	111	m[224]	m[225]	m[226]	m[227]

- Assume the following cache:
 - **S** = **8** sets (so s=3 bits for set index)
 - **E** = 1 line per set (so direct-mapped)
 - **B** = 4 bytes per line (so b=2 bits for block offset)
- What is the set index, tag, and block offset for address 226? Is it a hit?
 - Hint: 226 in binary is 11100010

