CS61C
Cache Memory

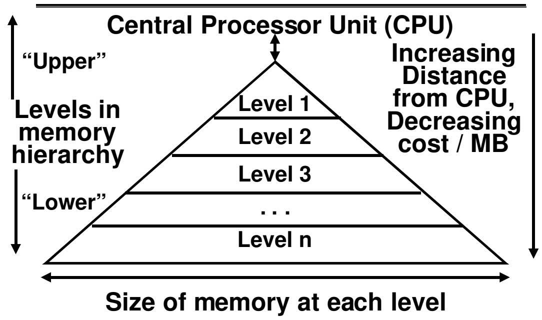
Lecture 17

March 31, 1999

Dave Patterson
(http.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs61c/schedule.html

Review 1/3: Memory Hierarchy Pyramid



Review 2/3: Hierarcgy Analogy: Library

- Term Paper: Every time need a book
  - Leave some books on desk after fetching them
  - Only go to shelves when need a new book
  - When go to shelves, bring back related books in case you need them; sometimes you'll need to return books not used recently to make space for new books on desk
  - Return to desk to work
  - When done, replace books on shelves, carrying as many as you can per trip
- Illusion: whole library on your desktop

Review 3/3

- Principle of Locality (locality in time, locality in space) + Hierarchy of Memories of different speed, cost; exploit locality to improve cost-performance
- Hierarchy Terms: Hit, Miss, Hit Time, Miss Penalty, Hit Rate, Miss Rate, Block, Upper level memory, Lower level memory
- Review of Big Ideas (so far):
  - Abstraction, Stored Program, Pliable Data, compilation vs. interpretation, Performance via Parallelism, Performance Pitfalls
  - Applies to Processor, Memory, and I/O

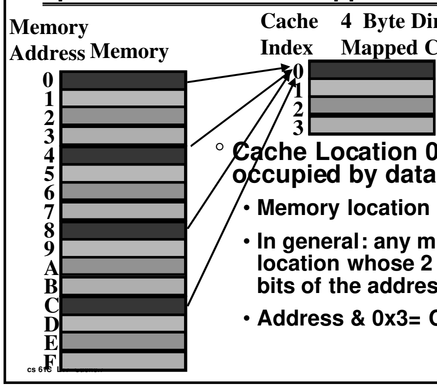
Outline

- Review
- Direct Mapped Cache
- Example
- Administrivia, Midterm results, Some survey results
- Block Size to Reduce Misses
- Associative Cache to Reduce Misses
- Conclusion

Cache: 1st Level of Memory Hierarchy

- How do you know if something is in the cache?
- How find it if it is in the cache?
- In a <u>direct mapped cache</u>, each memory address is associated with one possible block (also called "<u>line</u>") within the cache
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache

Simplest Cache: Direct Mapped



° Cache Location 0 can be occupied by data from:

- Memory location 0, 4, 8, ...
- In general: any memory location whose 2 rightmost bits of the address are 0s
- Address & 0x3= Cache index

Direct Mapped Questions

° Which memory block is in the cache?

° Also, What if block size is > 1 byte?

° Divide Memory Address into 3 portions: tag, index, and byte offset within block

tttttttttttttttt	iiiiiiiiii	oooo

° The index tells where in the cache to look, the offset tells which byte in block is start of the desired data, and the tag tells if the data in the cache corresponds to the memory address being looking for

Size of Tag, Index, Offset fields

° If

- 32-bit Memory Address
- Cache size = 2^N bytes
- Block (line) size = 2^M bytes

° Then

- The leftmost (32 - N) bits are the Cache Tag
- The rightmost M bits are the Byte Offset
- Remaining bits are the Cache Index

31 N N-1 M M-1 0

ttttttttttttttttt	iiiiiiiiii	oooo

Accessing data in a direct mapped cache

° So lets go through accessing some data in a direct mapped, 16KB cache

- 16 byte blocks x 1024 cache blocks

° 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

° 000000000000000000 0000000001 0100

° 000000000000000000 0000000001 1100

° 000000000000000000 0000000011 0100

° 000000000000000010 0000000001 0100

Tag Index Offset

16 KB Direct Mapped Cache, 16B blocks

° Valid bit ⇒ no address match when power on cache
(Not valid ⇒ no match even if tag = addr)

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1						
2						
3						
4						
5						
6						
7						
...				...		
1022						
1023						

Read 000000000000000000 0000000001 0100

° 000000000000000000 0000000001 0100
Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1						
2						
3						
4						
5						
6						
7						
...				...		
1022						
1023						

So we read block 1 (0000000001)

° 000000000000000000 0000000001 0100

Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1						
2						
3						
4						
5						
6						
7						
...			...			
1022						
1023						

No valid data

° 000000000000000000 0000000001 0100

Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1						
2						
3						
4						
5						
6						
7						
...			...			
1022						
1023						

So load that data into cache, setting tag, valid

° 000000000000000000 0000000001 0100

Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3						
4						
5						
6						
7						
...			...			
1022						
1023						

Read from cache at offset, return word b

° 000000000000000000 0000000001 0100

Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3						
4						
5						
6						
7						
...			...			
1022						
1023						

Read 000000000000000000 0000000001 1100

° 000000000000000000 0000000001 1100

Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3						
4						
5						
6						
7						
...			...			
1022						
1023						

Data valid, tag OK, so read offset return word d

° 000000000000000000 0000000001 1100

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3						
4						
5						
6						
7						
...			...			
1022						
1023						

Read 000000000000000000 0000000011 0100

- 000000000000000000 0000000011 0100
  Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3						
4						
5						
6						
7						
...			...			
1022						
1023						

So read block 3

- 000000000000000000 0000000011 0100
  Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3						
4						
5						
6						
7						
...			...			
1022						
1023						

No valid data

- 000000000000000000 0000000011 0100
  Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3						
4						
5						
6						
7						
...			...			
1022						
1023						

Load that cache block, return word f

- 000000000000000000 0000000011 0100
  Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3	1	0	e	f	g	h
4						
5						
6						
7						
...			...			
1022						
1023						

Read 000000000000000010 0000000001 0100

- 000000000000000010 0000000001 0100
  Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3	1	0	e	f	g	h
4						
5						
6						
7						
...			...			
1022						
1023						

So read Cache Block 1, Data is Valid

- 000000000000000010 0000000001 0100
  Tag field / Index field / Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3	1	0	e	f	g	h
4						
5						
6						
7						
...			...			
1022						
1023						

Cache Block 1 Tag does not match (0 != 2)

° 000000000000000010 0000000001 0100
Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	0	a	b	c	d
2						
3	1	0	e	f	g	h
4						
5						
6						
7						
...			...			
1022						
1023						

Miss, so replace block 1 with new data & tag

° 000000000000000010 0000000001 0100
Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	2	i	j	k	l
2						
3	1	0	e	f	g	h
4						
5						
6						
7						
...			...			
1022						
1023						

And return word j

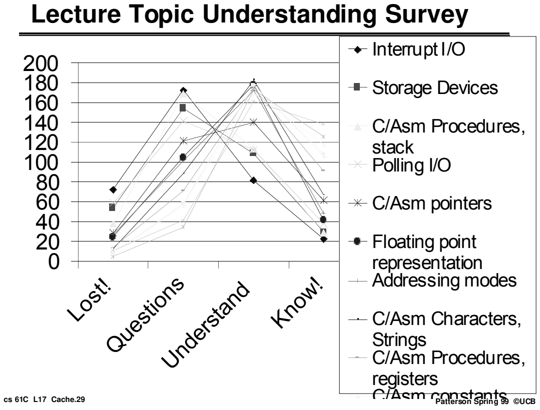
° 000000000000000010 0000000001 0100
Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0						
1	1	2	i	j	k	l
2						
3	1	0	e	f	g	h
4						
5						
6						
7						
...			...			
1022						
1023						

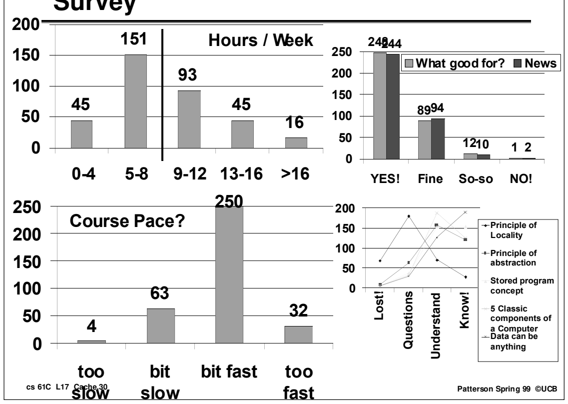
Administrivia

- Project 5: Due 4/14: design and implement a cache (in software) and plug into instruction simulator
- 6th homework: Due 4/7 7PM
  - Exercises 7.7, 7.8, 7.20, 7.22, 7.32
- Readings 7.2, 7.3, 7.4
- Midterm results
  - Average
  - Median
  - Max 50
  - Min

Lecture Topic Understanding Survey



Survey



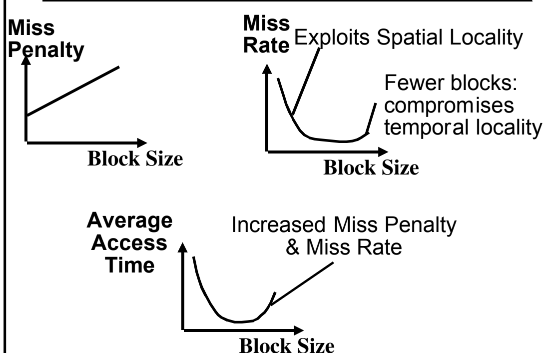
Block Size Tradeoff

- In general, larger block size take advantage of spatial locality BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
    - Too few cache blocks
- <u>In general, minimize Average Access Time:</u>
  - <u>= Hit Time x (1 - Miss Rate) + Miss Penalty x Miss Rate</u>

Extreme Example: single big block(!)

- Cache Size = 4 bytes Block Size = 4 bytes
  - Only <u>ONE</u> entry in the cache!
- If item accessed, likely accessed again soon
  - But unlikely will be accessed again immediately!
- The next access will likely to be a miss again
  - Continually loading data into the cache but discard data (force out) before use it again
  - Nightmare for cache designer: Ping Pong Effect

Block Size Tradeoff

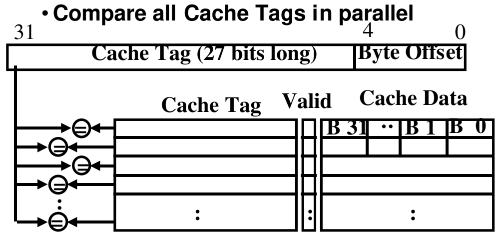


One Reason for Misses, and Solutions

- "Conflict Misses" are misses caused by different memory locations mapped to the same cache index accessed almost simultaneously
- Solution 1: Make the cache size bigger
- Solution 2: Multiple entries for the same Cache Index?

Extreme Example: Fully Associative

- <u>Fully Associative Cache</u> (e.g., 32 B block)
  - Forget about the Cache Index
  - Compare all Cache Tags in parallel

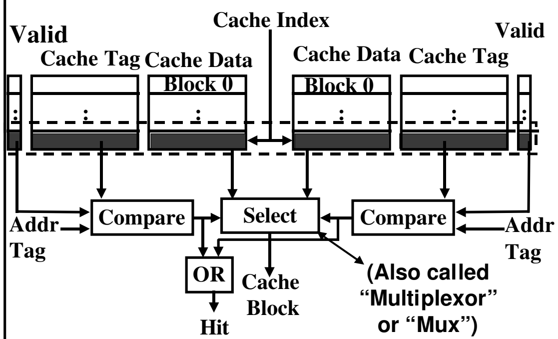


- By definition: Conflict Misses = 0 for a fully associative cache

Compromise: N-way Set Associative Cache

- <u>N-way set associative</u>: N entries for each Cache Index
  - N direct mapped caches operate in parallel
  - Select the one that gets a hit
- Example: 2-way set associative cache
  - Cache Index selects a "set" from the cache
  - The 2 tags in set are compared in parallel
  - Data is selected based on the tag result (which matched the address)

Compromise: 2-way Set Associative Cache



Block Replacement Policy

- N-way Set Associative or Fully Associative have choice where to place a block, (which block to replace)
    - Of course, if there is an invalid block, use it
- Whenever get a cache hit, record the cache block that was touched
- When need to evict a cache block, choose one which hasn't been touched recently: “Least Recently Used” (LRU)
    - Past is prologue: history suggests it is least likely of the choices to be used soon
    - Flip side of temporal locality

Block Replacement Policy: Random

- Sometimes hard to keep track of LRU if lots choices
- Second Choice Policy: pick one at random and replace that block
- Advantages
    - Very simple to implement
    - Predictable behavior
    - No worst case behavior

“And in Conclusion ...”

- Tag, index, offset to find matching data, support larger blocks, reduce misses
- Where in cache? Direct Mapped Cache
    - Conflict Misses if memory addresses compete
- Fully Associative to let memory data be in any block: no Conflict Misses
- Set Associative: Compromise, simpler hardware than Fully Associative, fewer misses than Direct Mapped
- LRU: Use history to predict replacement
- Next: Cache Improvement, Virtual Memory