
Cache Memory

CSE 410, Spring 2004
Computer Systems

<http://www.cs.washington.edu/education/courses/410/04sp/>

Reading and References

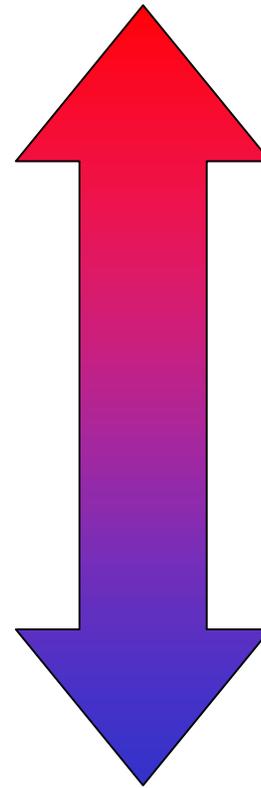
- Reading
 - » Sections 7.1-7.3, *Computer Organization and Design, Patterson and Hennessy*
- Reference
 - » Chapter 4, *See MIPS Run*, D. Sweetman

The Quest for Speed - Memory

- If all memory accesses (IF/lw/sw) accessed main memory, programs would run 20 times slower
- And it's getting worse
 - » processors speed up by 50% annually
 - » memory accesses speed up by 9% annually
 - » it's becoming harder and harder to keep these processors fed

A Solution: Memory Hierarchy

- Keep copies of the active data in the small, fast, expensive storage
- Keep all data in the big, slow, cheap storage



*fast, small,
expensive
storage*

*slow, large,
cheap storage*

Memory Hierarchy

Memory Level	Fabrication Tech	Access Time (ns)	Typ. Size (bytes)	\$/MB
Registers	Registers	<0.5	256	1000
L1 Cache	SRAM	2	8K	100
L2 Cache	SRAM	10	1M	100
Memory	DRAM	50	128M	0.75
Disk	Magnetic Disk	10M	32G	0.0035

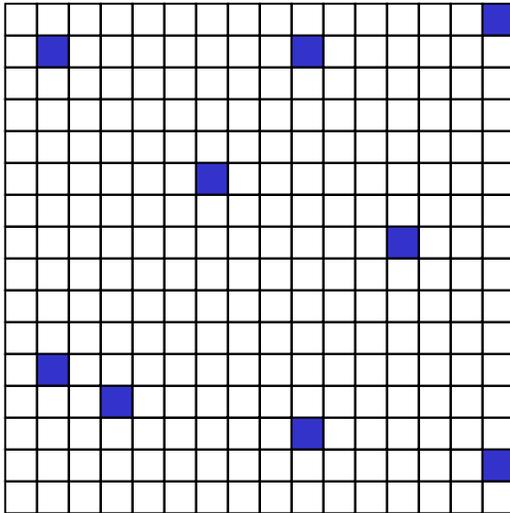
What is a Cache?

- A cache allows for fast accesses to a subset of a larger data store
- Your web browser's cache gives you fast access to pages you visited recently
 - » faster because it's stored locally
 - » subset because the web won't fit on your disk
- The memory cache gives the processor fast access to memory that it used recently
 - » faster because it's fancy and usually located on the CPU chip
 - » subset because the cache is smaller than main memory

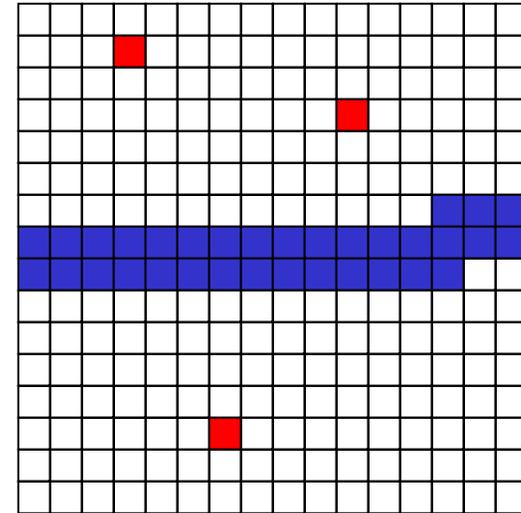
Locality of reference

- Temporal locality - nearness in time
 - » Data being accessed now will probably be accessed again soon
 - » Useful data tends to continue to be useful
- Spatial locality - nearness in address
 - » Data near the data being accessed now will probably be needed soon
 - » Useful data is often accessed sequentially

Memory Access Patterns



- Memory accesses **don't** usually look like this
 - » random accesses



- Memory accesses **do** usually look like this
 - hot variables
 - step through arrays

Cache Terminology

- **Hit and Miss**
 - » the data item is in the cache or the data item is not in the cache
- **Hit rate and Miss rate**
 - » the percentage of references that the data item is in the cache or not in the cache
- **Hit time and Miss time**
 - » the time required to access data in the cache (cache access time) and the time required to access data not in the cache (memory access time)

Effective Access Time

$$t_{\text{effective}} = (h) t_{\text{cache}} + (1-h) t_{\text{memory}}$$

↑ effective access time

↑ cache access time

↑ memory access time

cache hit rate

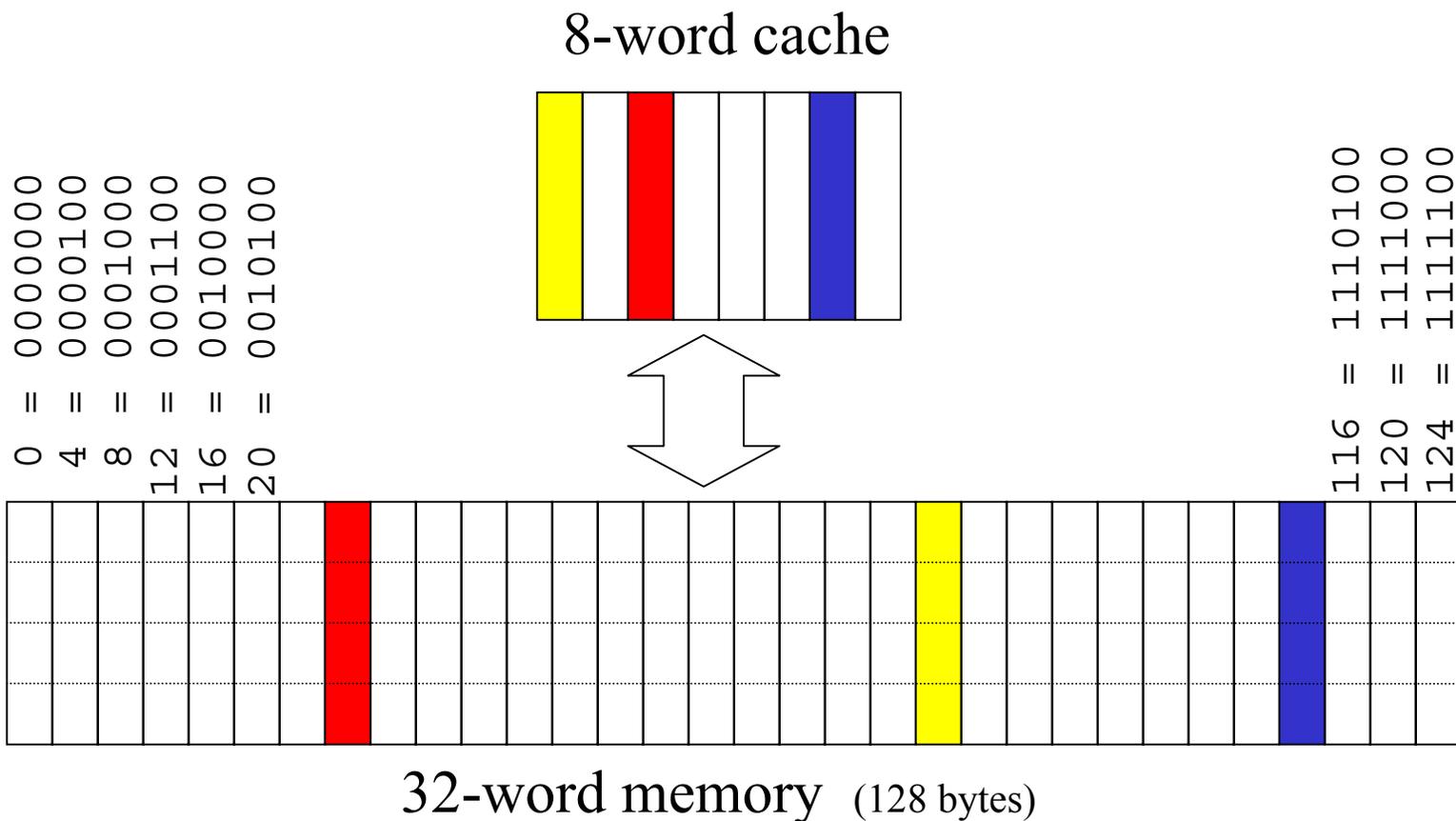
cache miss rate

aka, Average Memory Access Time (AMAT)

Cache Contents

- When do we put something in the cache?
 - » when it is used for the first time
- When do we overwrite something in the cache?
 - » when we need the space in the cache for some other entry
 - » all of memory won't fit on the CPU chip so not every location in memory can be cached

A small two-level hierarchy



Fully Associative Cache

- In a fully associative cache,
 - » any memory word can be placed in any **cache line**
 - » each cache line stores an address and a data value
 - » accesses are slow (but not as slow as you might think)

Address	Valid	Value
0010100	Y	0x00000001
0000100	N	0x09D91D11
0100100	Y	0x00000410
0101100	Y	0x00012D10
0001100	N	0x00000005
1101100	Y	0x0349A291
0100000	Y	0x000123A8
1111100	N	0x00000200

Direct Mapped Caches

- Fully associative caches are often too slow
- With direct mapped caches the address of the item determines where in the cache to store it
 - » In our example, the lowest order two bits are the byte offset within the word stored in the cache
 - » The next three bits of the address dictate the location of the entry within the cache
 - » The remaining higher order bits record the rest of the original address as a tag for this entry

Address Tags

- A *tag* is a label for a cache entry indicating where it came from
 - » The upper bits of the data item's address

7 bit Address
1011101

Tag (2)	Index (3)	Byte Offset (2)
10	111	01

Direct Mapped Cache

Cache Contents

Memory Address	Tag	Valid	Value	Cache Index
11 <u>000</u> 00	11	Y	0x00000001	$000_2 = 0$
10 <u>001</u> 00	10	N	0x09D91D11	$001_2 = 1$
01 <u>010</u> 00	01	Y	0x00000410	$010_2 = 2$
00 <u>011</u> 00	00	Y	0x00012D10	$011_2 = 3$
10 <u>100</u> 00	10	N	0x00000005	$100_2 = 4$
11 <u>101</u> 00	11	Y	0x0349A291	$101_2 = 5$
00 <u>110</u> 00	00	Y	0x000123A8	$110_2 = 6$
10 <u>111</u> 00	10	N	0x00000200	$111_2 = 7$

N-way Set Associative Caches

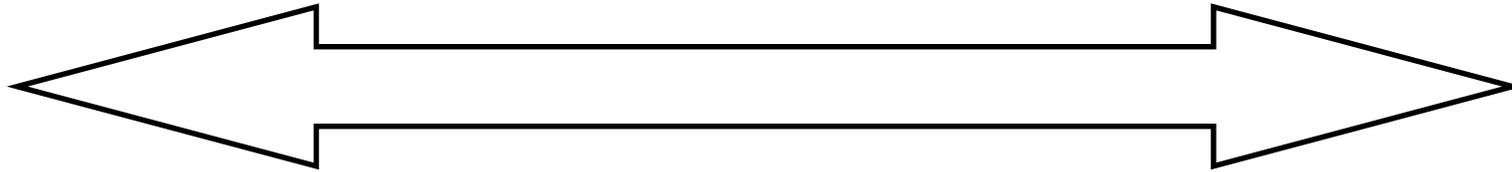
- Direct mapped caches cannot store more than one address with the same index
- If two addresses collide, then you overwrite the older entry
- 2-way associative caches can store two different addresses with the same index
 - » 3-way, 4-way and 8-way set associative designs too
- Reduces misses due to conflicts
- Larger sets imply slower accesses

2-way Set Associative Cache

Index	Tag	Valid	Value	Tag	Valid	Value
000	11	Y	0x00000001	00	Y	0x00000002
001	10	N	0x09D91D11	10	N	0x0000003B
010	01	Y	0x00000410	11	Y	0x000000CF
011	00	Y	0x00012D10	10	N	0x000000A2
100	10	N	0x00000005	11	N	0x00000333
101 ⇒	11	Y	0x0349A291	10	Y	0x00003333
110	00	Y	0x000123A8	01	Y	0x0000C002
111	10	N	0x00000200	10	N	0x00000005

The highlighted cache entry contains values for addresses $10101xx_2$ and $11101xx_2$.

Associativity Spectrum



Direct Mapped

Fast to access

Conflict Misses

N-way Associative

Slower to access

Fewer Conflict Misses

Fully Associative

Slow to access

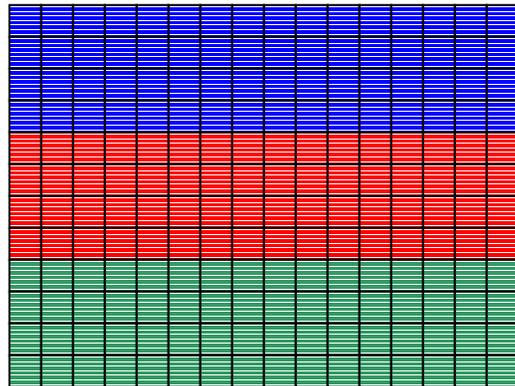
No Conflict Misses

Spatial Locality

- Using the cache improves performance by taking advantage of temporal locality
 - » When a word in memory is accessed it is loaded into cache memory
 - » It is then available quickly if it is needed again soon
- This does nothing for spatial locality

Memory Blocks

- Divide memory into **blocks**
- If any word in a block is accessed, then load an entire block into the cache



Block 0 0x00000000–0x0000003F

Block 1 0x00000040–0x0000007F

Block 2 0x00000080–0x000000BF

Cache line for 16 word block size

tag	valid	w ₀	w ₁	w ₂	w ₃	w ₄	w ₅	w ₆	w ₇	w ₈	w ₉	w ₁₀	w ₁₁	w ₁₂	w ₁₃	w ₁₄	w ₁₅
-----	-------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

Address Tags Revisited

- A cache block size > 1 word requires the address to be divided differently
- Instead of a byte offset into a word, we need a byte offset into the block
- Assume we have 10-bit addresses, 8 cache lines, and 4 words (16 bytes) per cache line block...

10 bit Address		
0101100111		
Tag (3)	Index (3)	Block Offset (4)
010	110	0111

The Effects of Block Size

- Big blocks are good
 - » Fewer first time misses
 - » Exploits spatial locality
- Small blocks are good
 - » Don't evict as much data when bringing in a new entry
 - » More likely that all items in the block will turn out to be useful

Reads vs. Writes

- Caching is essentially making a copy of the data
- When you read, the copies still match when you're done
- When you write, the results must eventually propagate to both copies
 - » Especially at the lowest level of the hierarchy, which is in some sense the permanent copy

Write-Through Caches

- Write all updates to both cache and memory
- Advantages
 - » The cache and the memory are always consistent
 - » Evicting a cache line is cheap because no data needs to be written out to memory at eviction
 - » Easy to implement
- Disadvantages
 - » Runs at memory speeds when writing (can use write buffer to reduce this problem)

Write-Back Caches

- Write the update to the cache only. Write to memory only when cache block is evicted
- Advantage
 - » Runs at cache speed rather than memory speed
 - » Some writes never go all the way to memory
 - » When a whole block is written back, can use high bandwidth transfer
- Disadvantage
 - » complexity required to maintain consistency

Dirty bit

- When evicting a block from a write-back cache, we could
 - » always write the block back to memory
 - » write it back only if we changed it
- Caches use a “dirty bit” to mark if a line was changed
 - » the dirty bit is 0 when the block is loaded
 - » it is set to 1 if the block is modified
 - » when the line is evicted, it is written back only if the dirty bit is 1

i-Cache and d-Cache

- There usually are two separate caches for instructions and data.
 - » Avoids structural hazards in pipelining
 - » The combined cache is twice as big but still has an access time of a small cache
 - » Allows both caches to operate in parallel, for twice the bandwidth

Cache Line Replacement

- How do you decide which cache block to replace?
- If the cache is direct-mapped, it's easy
 - » only one slot per index
- Otherwise, common strategies:
 - » Random
 - » Least Recently Used (LRU)

LRU Implementations

- LRU is very difficult to implement for high degrees of associativity
- 4-way approximation:
 - » 1 bit to indicate least recently used pair
 - » 1 bit per pair to indicate least recently used item in this pair
- We will see this again at the operating system level

Multi-Level Caches

- Use each level of the memory hierarchy as a cache over the next lowest level
- Inserting level 2 between levels 1 and 3 allows:
 - » level 1 to have a higher miss rate (so can be smaller and cheaper)
 - » level 3 to have a larger access time (so can be slower and cheaper)

Cache Comparisons

L1 i-Cache	Alpha 21164	MIPS R10000	Pentium Pro	UltraSparc 1
	8KB direct-mapped 32B block	32KB 2-way (LRU) 64B block	8KB 4-way 32B block	16KB pseudo 2-way 32B block
L1 d-Cache	Alpha 21164	MIPS R10000	Pentium Pro	UltraSparc 1
	8KB direct-mapped 32B block	32KB 2-way (LRU) 32B block	8KB 2-way 32B block	16KB direct-mapped 32B block
L2 unified Cache	Alpha 21164		Pentium Pro	
	96KB 3-way 64B block on chip		256KB 4-way 32B block same package	

Summary: Classifying Caches

- Where can a block be placed?
 - » Direct mapped, N-way Set or Fully associative
- How is a block found?
 - » Direct mapped: by index
 - » Set associative: by index and search
 - » Fully associative: by search
- What happens on a write access?
 - » Write-back or Write-through
- Which block should be replaced?
 - » Random
 - » LRU (Least Recently Used)