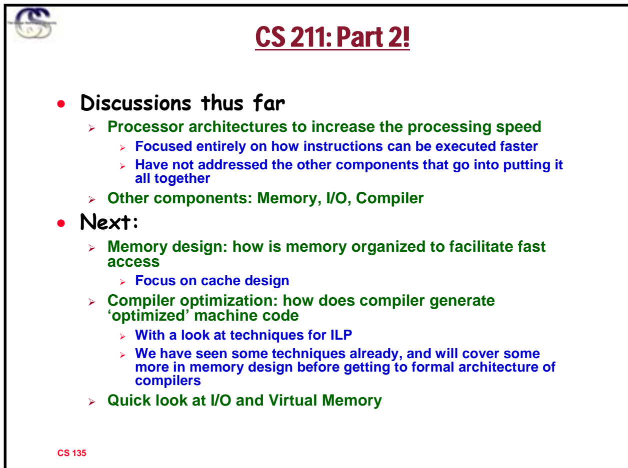
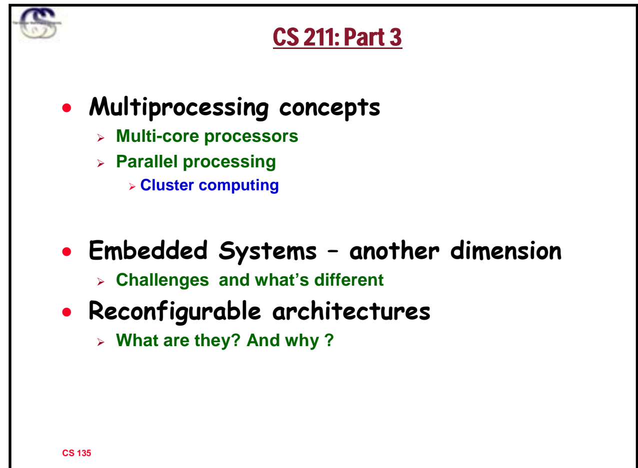
CS 211: Computer Architecture
Cache Memory Design

Course Objectives: Where are we?
CS 135



CS 211: Part 2!
Discussions thus far
Processor architectures to increase the processing speed
Focused entirely on how instructions can be executed faster
Have not addressed the other components that go into putting it all together
Other components: Memory, I/O, Compiler
Next:
Memory design: how is memory organized to facilitate fast access
Focus on cache design
Compiler optimization: how does compiler generate 'optimized' machine code
With a look at techniques for ILP
We have seen some techniques already, and will cover some more in memory design before getting to formal architecture of compilers
Quick look at I/O and Virtual Memory
CS 135



CS 211: Part 3
Multiprocessing concepts
Multi-core processors
Parallel processing
Cluster computing
Embedded Systems – another dimension
Challenges and what's different
Reconfigurable architectures
What are they? And why ?
CS 135

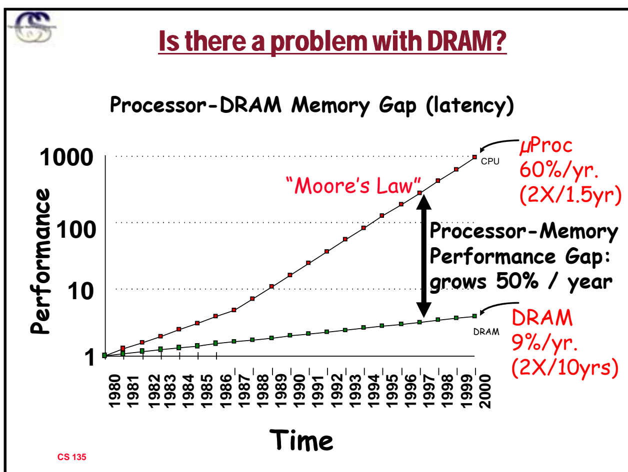
Memory

- In our discussions (on MIPS pipeline, superscalar, EPIC) we've constantly been assuming that we can access our operand from memory in 1 clock cycle…
  - This is possible, but its complicated
  - We'll now discuss how this happens
- We'll talk about…
  - Memory Technology
  - Memory Hierarchy
    - Caches
    - Memory
    - Virtual Memory

Memory Technology

- Memory Comes in Many Flavors
  - SRAM (Static Random Access Memory)
    - Like a register file; once data written to SRAM its contents stay valid – no need to refresh it
  - DRAM (Dynamic Random Access Memory)
    - Like leaky capacitors – data stored into DRAM chip charging memory cells to max values; charge slowly leaks and will eventually be too low to be valid – therefore refresh circuitry rewrites data and charges back to max
    - Static RAM is faster but more expensive
      - Cache uses static RAM
  - ROM, EPROM, EEPROM, Flash, etc.
    - Read only memories – store OS
  - Disks, Tapes, etc.
- Difference in speed, price and "size"
  - Fast is small and/or expensive
  - Large is slow and/or expensive

Is there a problem with DRAM?



Why Not Only DRAM?

- Can lose data when no power
  - Volatile storage
- Not large enough for some things
  - Backed up by storage (disk)
  - Virtual memory, paging, etc.
- Not fast enough for processor accesses
  - Takes hundreds of cycles to return data
  - OK in very regular applications
    - Can use SW pipelining, vectors
  - Not OK in most other applications

The principle of locality…

- …says that most programs don't access all code or data uniformly
  - *e.g.* in a loop, small subset of instructions might be executed over and over again…
  - …& a block of memory addresses might be accessed sequentially…
- This has led to "memory hierarchies"
- Some important things to note:
  - Fast memory is expensive
  - Levels of memory usually smaller/faster than previous
  - Levels of memory usually "subset" one another
    - All the stuff in a higher level is in some level below it

Levels in a typical memory hierarchy

CPU
Registers
Cache
Memory bus
Memory
I/O bus
I/O devices
Register reference
Cache reference
Memory reference
Disk memory reference

	Register reference	Cache reference	Memory reference	Disk memory reference
Size:	500 bytes	64 KB	512 MB	100 GB
Speed:	0.25 ns	1 ns	100 ns	5 ms

© 2003 Elsevier Science (USA). All rights reserved.

Memory Hierarchies

- Key Principles
  - Locality – most programs do not access code or data uniformly
  - Smaller hardware is faster
- Goal
  - Design a memory hierarchy "with cost almost as low as the cheapest level of the hierarchy and speed almost as fast as the fastest level"
    - This implies that we be clever about keeping more likely used data as "close" to the CPU as possible
- Levels provide subsets
  - Anything (data) found in a particular level is also found in the next level below.
  - Each level maps from a slower, larger memory to a smaller but faster memory

The Full Memory Hierarchy "always reuse a good idea"

Capacity
Access Time
Cost

CPU Registers
100s Bytes
<10s ns

Cache
K Bytes
10-100 ns
1-0.1 cents/bit

Main Memory
M Bytes
200ns- 500ns
$.0001-.00001 cents /bit

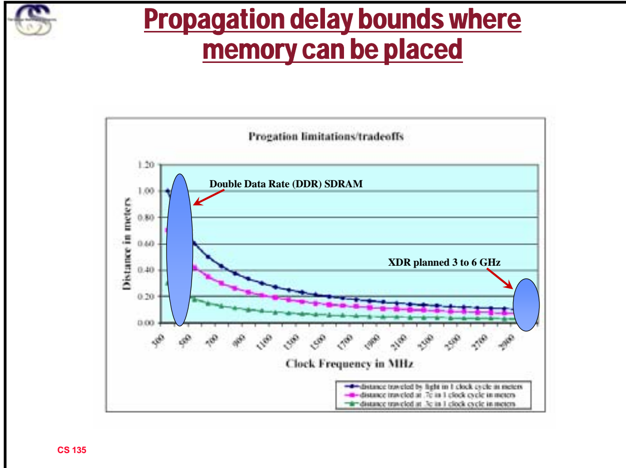
Disk
G Bytes, 10 ms
(10,000,000 ns)
10^{-5} - 10^{-6} cents/bit

Tape
infinite
sec-min
10^{-8}

Registers
Instr. Operands
Our current focus
Cache
Blocks
Memory
Pages
Disk
Files
Tape

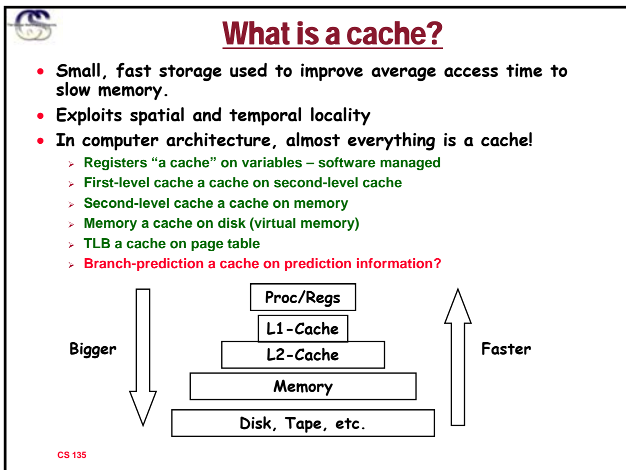
Staging Xfer Unit
prog./compiler 1-8 bytes
cache cntl 8-128 bytes
OS 4K-16K bytes
user/operator Mbytes

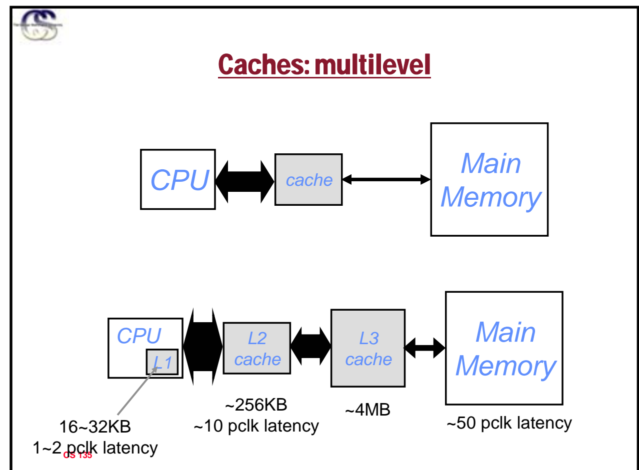
Upper Level
faster
Larger
Lower Level



Cache: Terminology

- Cache is name given to the first level of the memory hierarchy encountered once an address leaves the CPU
  - Takes advantage of the principle of locality
- The term cache is also now applied whenever buffering is employed to reuse items
- Cache controller
  - The HW that controls access to cache or generates request to memory
- No cache in 1980 PCs to 2-level cache by 1995..!





A brief description of a cache

- Cache = next level of memory hierarchy up from register file
  - All values in register file should be in cache
- Cache entries usually referred to as "blocks"
  - Block is minimum amount of information that can be in cache
  - fixed size collection of data, retrieved from memory and placed into the cache
- Processor generates request for data/inst, first look up the cache
- If we're looking for item in a cache and find it, have a cache hit; if not then a

Terminology Summary

- Hit: data appears in block in upper level (i.e. block X in cache)
  - Hit Rate: fraction of memory access found in upper level
  - Hit Time: time to access upper level which consists of
    - RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieved from a block in the lower level (i.e. block Y in memory)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Extra time to replace a block in the upper level +
    - Time to deliver the block the processor
- Hit Time << Miss Penalty (500 instructions on Alpha 21264)

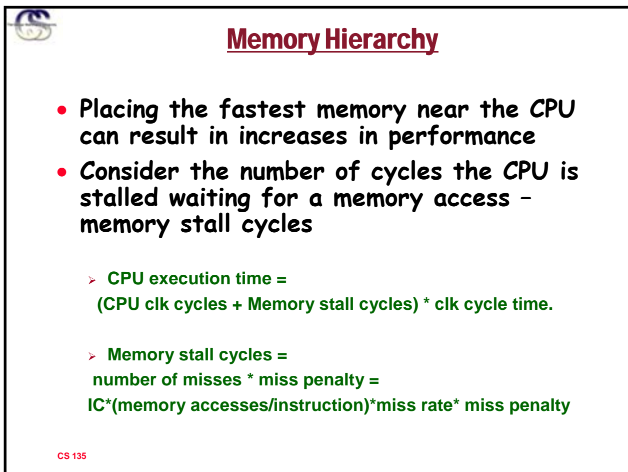
To Processor
From Processor
Upper Level Memory
Blk X
Lower Level Memory
Blk Y

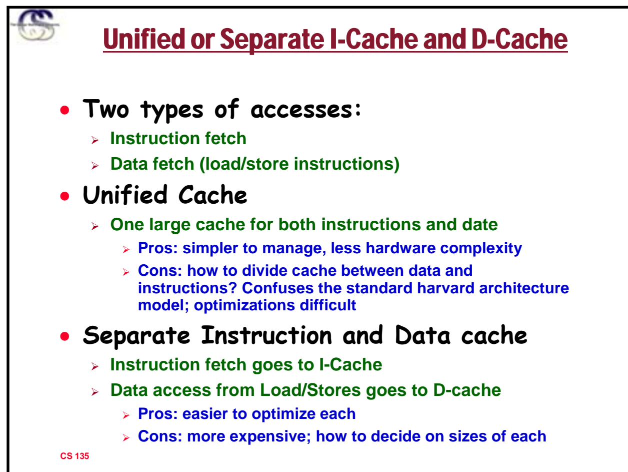
Definitions

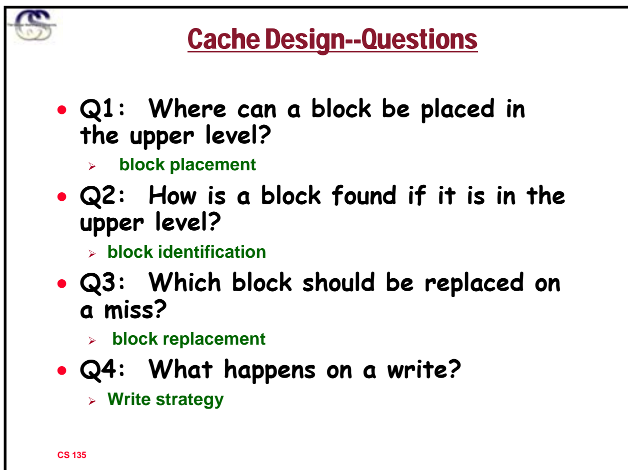
- Locating a block requires two attributes:
  - Size of block
  - Organization of blocks within the cache
- Block size (also referred to as line size)
  - Granularity at which cache operates
  - Each block is contiguous series of bytes in memory and begins on a naturally aligned boundary
  - Eg: cache with 16 byte blocks
    - each contains 16 bytes
    - First byte aligned to 16 byte boundaries in address space
      - Low order 4 bits of address of first byte would be 0000
  - Smallest usable block size is the natural word size of the processor
    - Else would require splitting an access across blocks and slows down translation

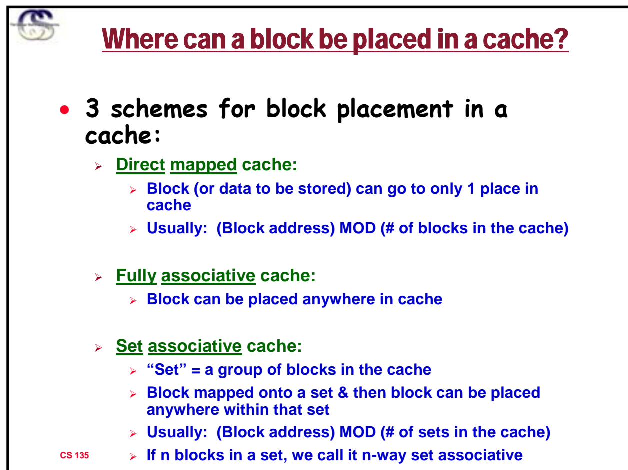
Cache Basics

- Cache consists of block-sized lines
  - Line size typically power of two
  - Typically 16 to 128 bytes in size
- Example
  - Suppose block size is 128 bytes
    - Lowest seven bits determine offset within block
  - Read data at address A=0x7fffa3f4
  - Address begins to block with base address 0x7fffa380


Memory Hierarchy
Placing the fastest memory near the CPU can result in increases in performance
Consider the number of cycles the CPU is stalled waiting for a memory access – memory stall cycles
CPU execution time =
(CPU clk cycles + Memory stall cycles) * clk cycle time.
Memory stall cycles =
number of misses * miss penalty =
IC*(memory accesses/instruction)*miss rate* miss penalty
CS 135


Unified or Separate I-Cache and D-Cache
Two types of accesses:
Instruction fetch
Data fetch (load/store instructions)
Unified Cache
One large cache for both instructions and date
Pros: simpler to manage, less hardware complexity
Cons: how to divide cache between data and instructions? Confuses the standard harvard architecture model; optimizations difficult
Separate Instruction and Data cache
Instruction fetch goes to I-Cache
Data access from Load/Stores goes to D-cache
Pros: easier to optimize each
Cons: more expensive; how to decide on sizes of each
CS 135


Cache Design--Questions
Q1: Where can a block be placed in the upper level?
block placement
Q2: How is a block found if it is in the upper level?
block identification
Q3: Which block should be replaced on a miss?
block replacement
Q4: What happens on a write?
Write strategy
CS 135


Where can a block be placed in a cache?
3 schemes for block placement in a cache:
Direct mapped cache:
Block (or data to be stored) can go to only 1 place in cache
Usually: (Block address) MOD (# of blocks in the cache)
Fully associative cache:
Block can be placed anywhere in cache
Set associative cache:
"Set" = a group of blocks in the cache
Block mapped onto a set & then block can be placed anywhere within that set
Usually: (Block address) MOD (# of sets in the cache)
If n blocks in a set, we call it n-way set associative
CS 135

Where can a block be placed in a cache?

	Fully Associative	Direct Mapped	Set Associative
Cache:	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8 (Set 0, Set 1, Set 2, Set 3)
	Block 12 can go anywhere	Block 12 can go only into Block 4 (12 mod 8)	Block 12 can go anywhere in set 0 (12 mod 4)

Memory: 1 2 3 4 5 6 7 8 9….. 12

Associativity

- If you have associativity > 1 you have to have a replacement policy
  - FIFO
  - LRU
  - Random
- "Full" or "Full-map" associativity means you check every tag in parallel and a memory block can go into any cache block
  - Virtual memory is effectively fully associative
  - (But don't worry about virtual memory yet)

Cache Organizations

- Direct Mapped vs Fully Associate
  - Direct mapped is not flexible enough; if X(mod K)=Y(mod K) then X and Y cannot both be located in cache
  - Fully associative allows any mapping, implies all locations must be searched to find the right one –expensive hardware
- Set Associative
  - Compromise between direct mapped and fully associative
  - Allow many-to-few mappings
  - On lookup, subset of address bits used to generate an index
  - BUT index now corresponds to a set of entries which can be searched in parallel – more efficient hardware implementation than fully associative, but due to flexible mapping behaves more like fully associative

Associativity

- If total cache size is kept same, increasing the associativity increases number of blocks per set
  - Number of simultaneous compares needed to perform the search in parallel = number of blocks per set
  - Increase by factor of 2 in associativity doubles number of blocks per set and halves number of sets

Large Blocks and Subblocking

- Large cache blocks can take a long time to refill
  - refill cache line *critical word first*
  - restart cache access before complete refill
- Large cache blocks can waste bus bandwidth if block size is larger than spatial locality
  - divide a block into subblocks
  - associate separate valid bits for each subblock.

v	subblock	v	subblock	• • • •	v	subblock	tag

Block Identification: How is a block found in the cache

- Since we have many-to-one mappings, need tag
- Caches have an address tag on each block that gives the block address.
  - Eg: if slot zero in cache contains tag K, the value in slot zero corresponds to block zero from area of memory that has tag K
  - Address consists of <tag t,block b,offset o>
    - Examine tag in slot b of cache:
      - if matches t then extract value from slot b in cache
      - Else use memory address to fetch block from memory, place copy in slot b of cache, replace tag with t, use o to select appropriate byte

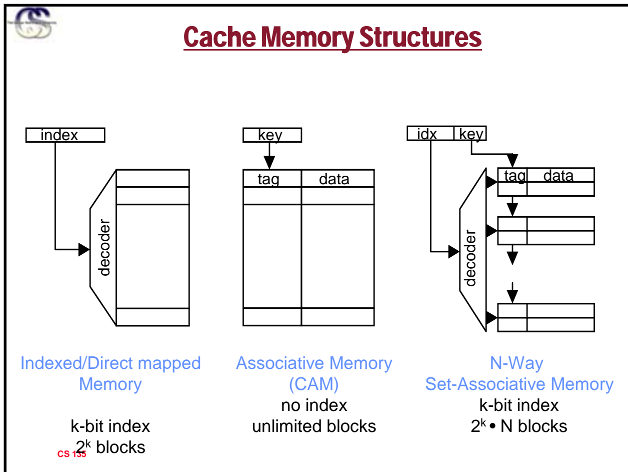
How is a block found in the cache?

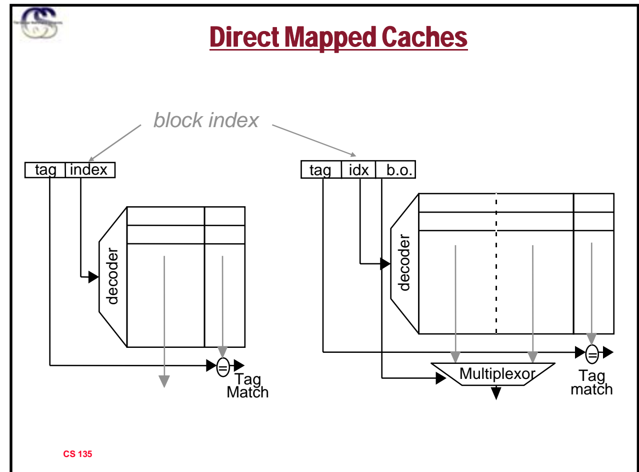
- Cache's have address tag on each block frame that provides block address
  - Tag of every cache block that might have entry is examined against CPU address (in parallel! – why?)
- Each entry usually has a valid bit
  - Tells us if cache data is useful/not garbage
  - If bit is not set, there can't be a match…
- How does address provided to CPU relate to entry in cache?
  - Entry divided between block address & block offset…
  - …and further divided between tag field & index field

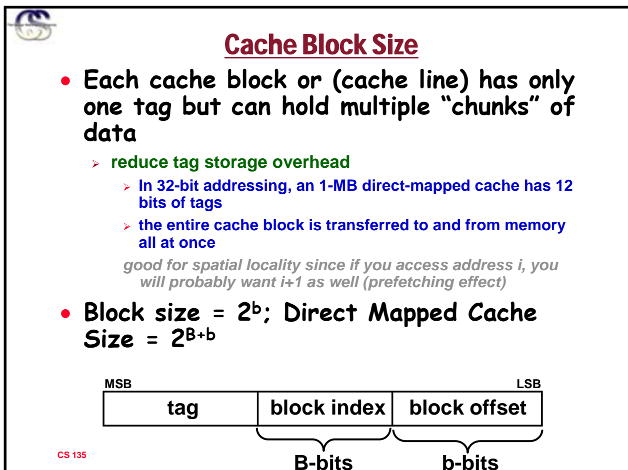
How is a block found in the cache?

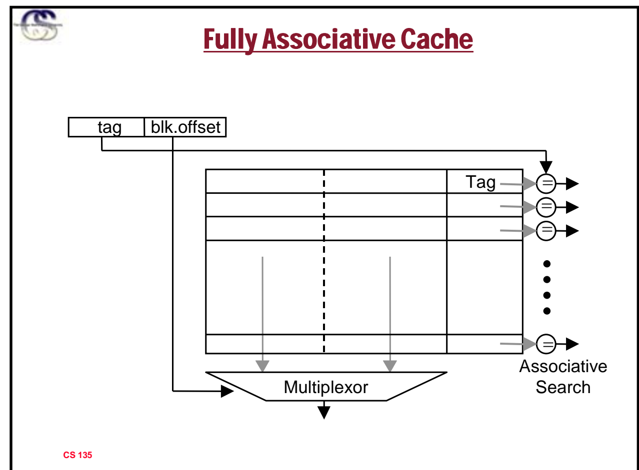
Block Address		Block Offset
Tag	Index	

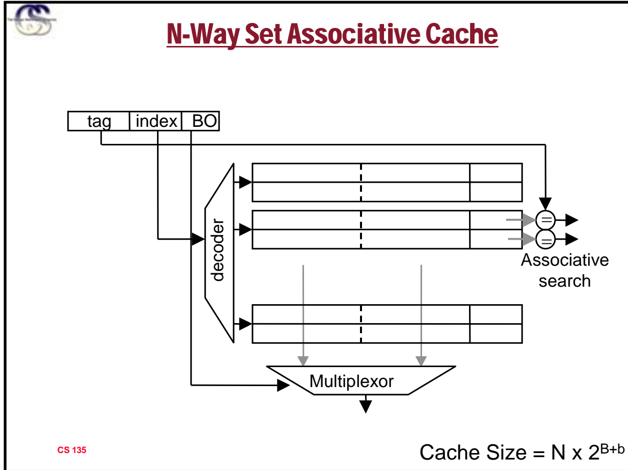
- Block offset field selects data from block
  - (i.e. address of desired data within block)
- Index field selects a specific set
  - Fully associative caches have no index field
- Tag field is compared against it for a hit
- Could we compare on more of address than the tag?
  - Not necessary; checking index is redundant
    - Used to select set to be checked
    - Ex.: Address stored in set 0 must have 0 in index field
  - Offset not necessary in comparison –entire block is present or not and all block offsets must match

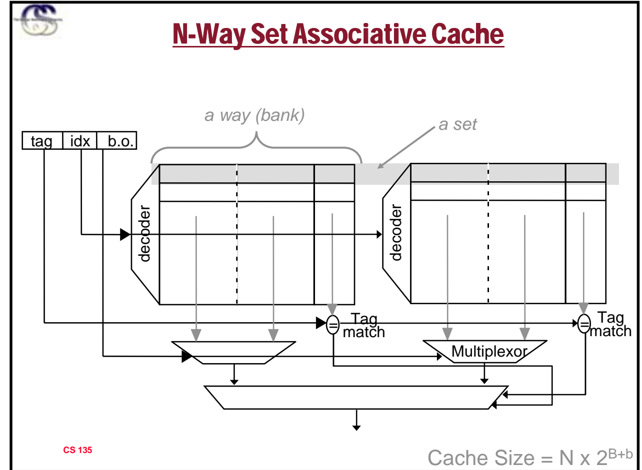

Cache Memory Structures
index
decoder
key
tag
data
idx
key
tag
data
decoder
Indexed/Direct mapped Memory
k-bit index
2^k blocks
Associative Memory (CAM)
no index
unlimited blocks
N-Way Set-Associative Memory
k-bit index
2^k • N blocks
CS 135


Direct Mapped Caches
block index
tag
index
decoder
Tag Match
tag
idx
b.o.
decoder
Multiplexor
Tag match
CS 135


Cache Block Size
Each cache block or (cache line) has only one tag but can hold multiple "chunks" of data
reduce tag storage overhead
In 32-bit addressing, an 1-MB direct-mapped cache has 12 bits of tags
the entire cache block is transferred to and from memory all at once
good for spatial locality since if you access address i, you will probably want i+1 as well (prefetching effect)
Block size = 2^b; Direct Mapped Cache Size = 2^(B+b)
MSB
LSB
tag
block index
block offset
B-bits
b-bits
CS 135

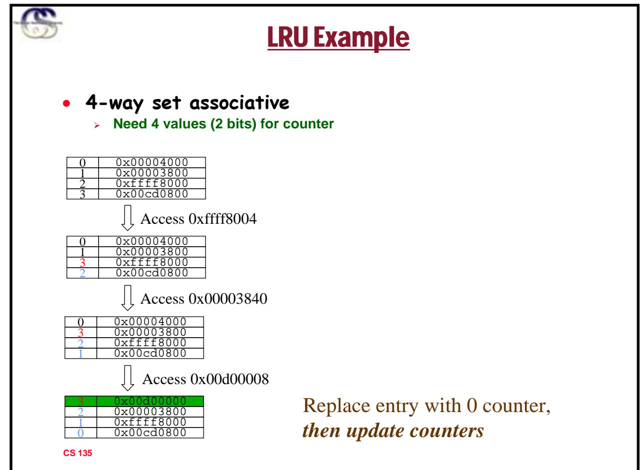

Fully Associative Cache
tag
blk.offset
Tag
Multiplexor
Associative Search
CS 135

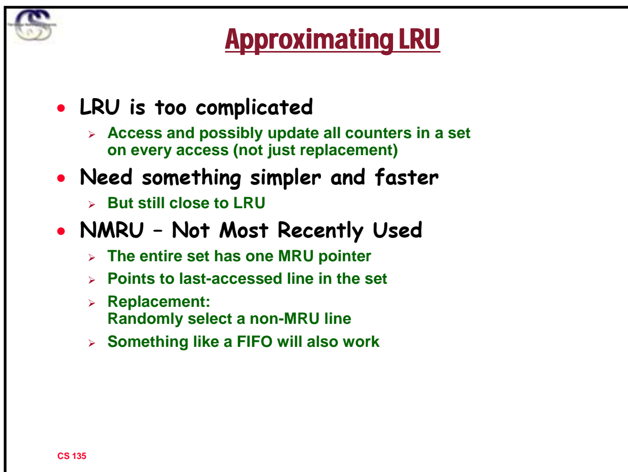


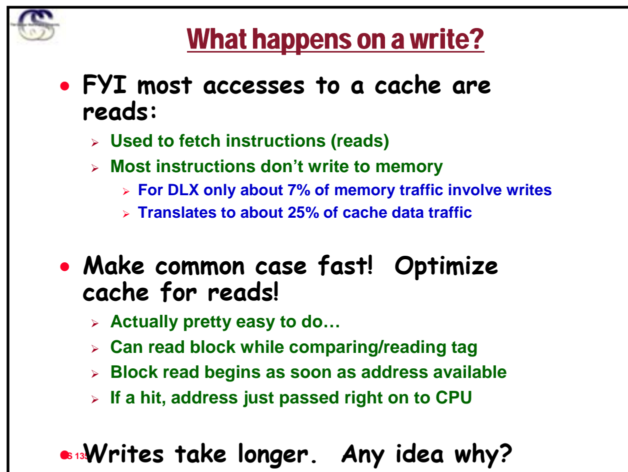


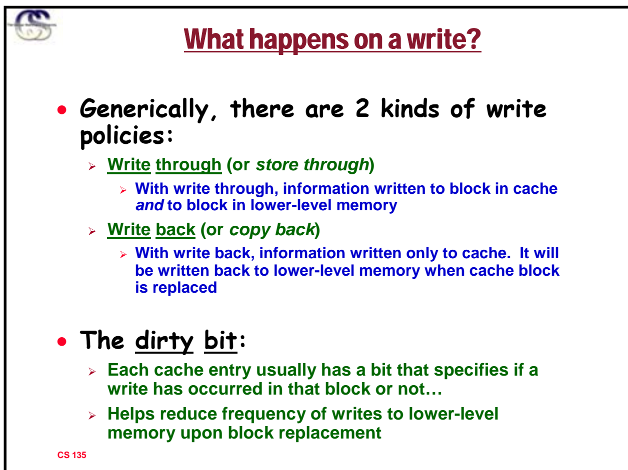
Which block should be replaced on a cache miss?

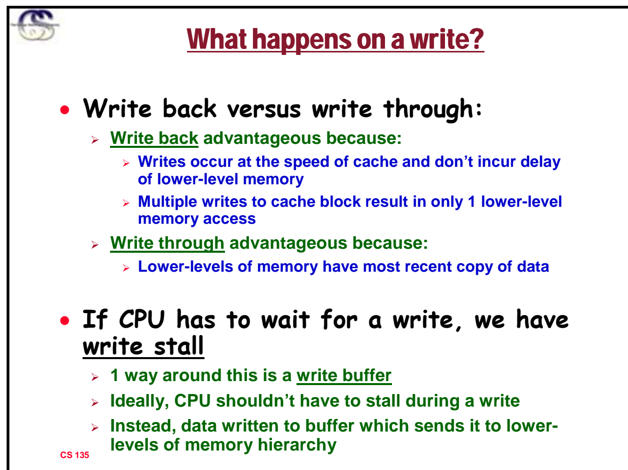
- If we look something up in cache and entry not there, generally want to get data from memory and put it in cache
  - B/c principle of locality says we'll probably use it again
- Direct mapped caches have 1 choice of what block to replace
- Fully associative or set associative offer more choices
- Usually 2 strategies:
  - Random – pick any possible block and replace it
  - LRU – stands for "Least Recently Used"
    - Why not throw out the block not used for the longest time
    - Usually approximated, not much better than random – i.e. 5.18% vs. 5.69% for 16KB 2-way set associative

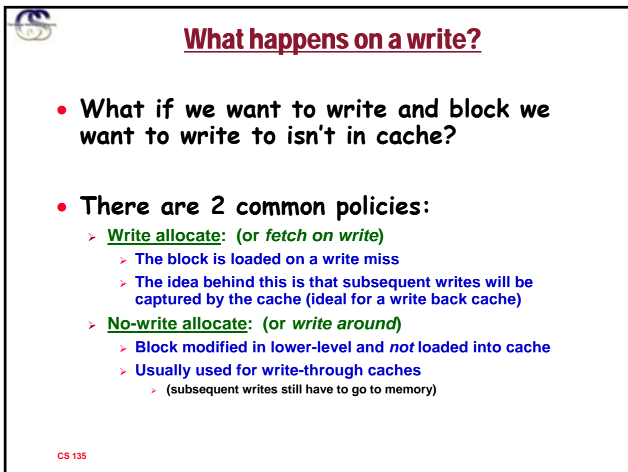


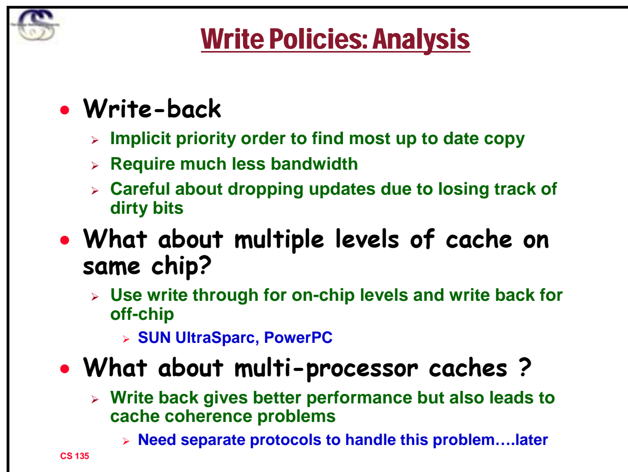

Approximating LRU
LRU is too complicated
Access and possibly update all counters in a set on every access (not just replacement)
Need something simpler and faster
But still close to LRU
NMRU – Not Most Recently Used
The entire set has one MRU pointer
Points to last-accessed line in the set
Replacement: Randomly select a non-MRU line
Something like a FIFO will also work
CS 135

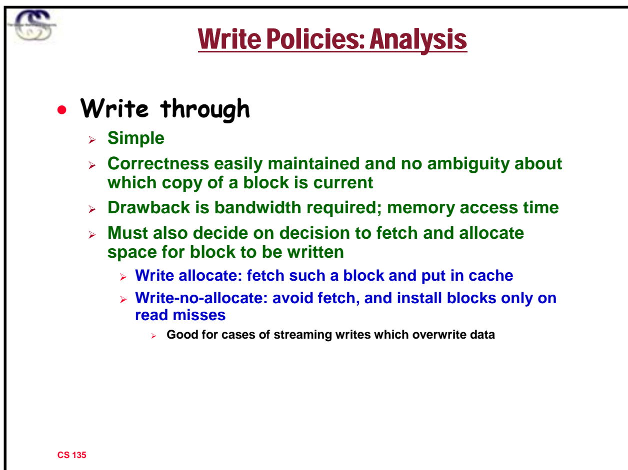

What happens on a write?
FYI most accesses to a cache are reads:
Used to fetch instructions (reads)
Most instructions don't write to memory
For DLX only about 7% of memory traffic involve writes
Translates to about 25% of cache data traffic
Make common case fast! Optimize cache for reads!
Actually pretty easy to do…
Can read block while comparing/reading tag
Block read begins as soon as address available
If a hit, address just passed right on to CPU
Writes take longer. Any idea why?

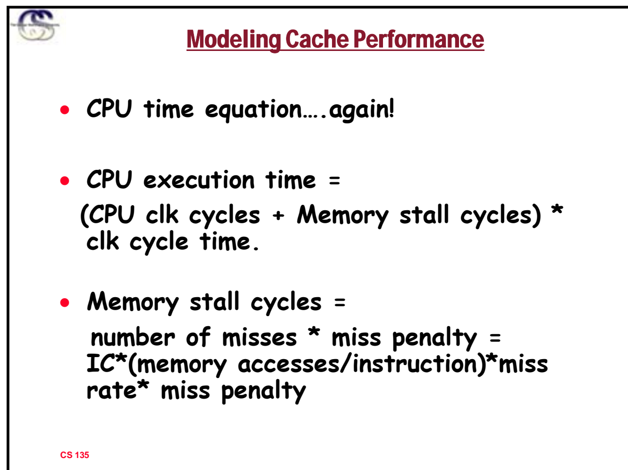

What happens on a write?
Generically, there are 2 kinds of write policies:
Write through (or store through)
With write through, information written to block in cache and to block in lower-level memory
Write back (or copy back)
With write back, information written only to cache. It will be written back to lower-level memory when cache block is replaced
The dirty bit:
Each cache entry usually has a bit that specifies if a write has occurred in that block or not…
Helps reduce frequency of writes to lower-level memory upon block replacement
CS 135

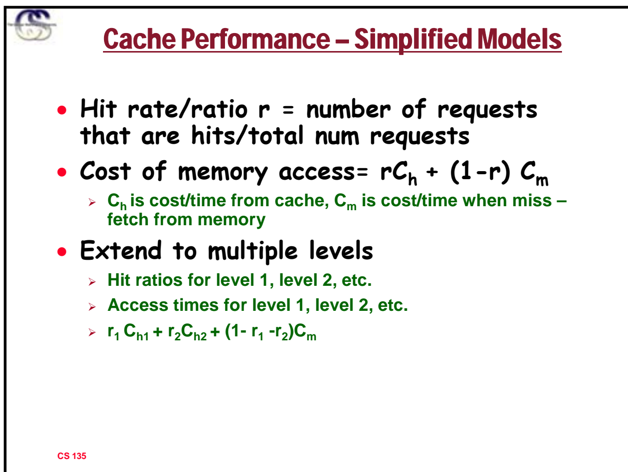

What happens on a write?
Write back versus write through:
Write back advantageous because:
Writes occur at the speed of cache and don't incur delay of lower-level memory
Multiple writes to cache block result in only 1 lower-level memory access
Write through advantageous because:
Lower-levels of memory have most recent copy of data
If CPU has to wait for a write, we have write stall
1 way around this is a write buffer
Ideally, CPU shouldn't have to stall during a write
Instead, data written to buffer which sends it to lower-levels of memory hierarchy
CS 135

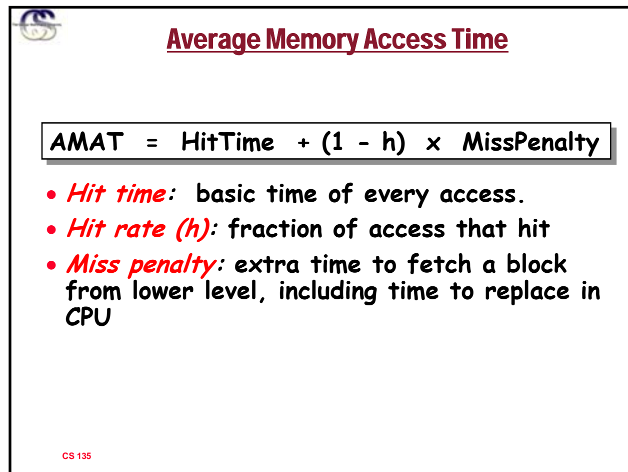

What happens on a write?
What if we want to write and block we want to write to isn't in cache?
There are 2 common policies:
Write allocate: (or fetch on write)
The block is loaded on a write miss
The idea behind this is that subsequent writes will be captured by the cache (ideal for a write back cache)
No-write allocate: (or write around)
Block modified in lower-level and not loaded into cache
Usually used for write-through caches
(subsequent writes still have to go to memory)
CS 135

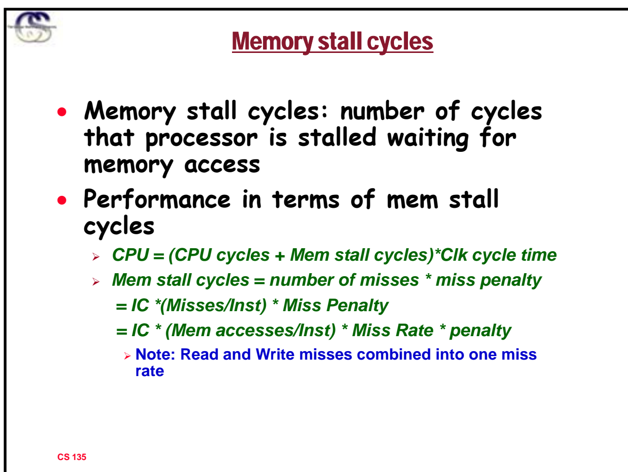

Write Policies: Analysis
Write-back
Implicit priority order to find most up to date copy
Require much less bandwidth
Careful about dropping updates due to losing track of dirty bits
What about multiple levels of cache on same chip?
Use write through for on-chip levels and write back for off-chip
SUN UltraSparc, PowerPC
What about multi-processor caches ?
Write back gives better performance but also leads to cache coherence problems
Need separate protocols to handle this problem….later
CS 135

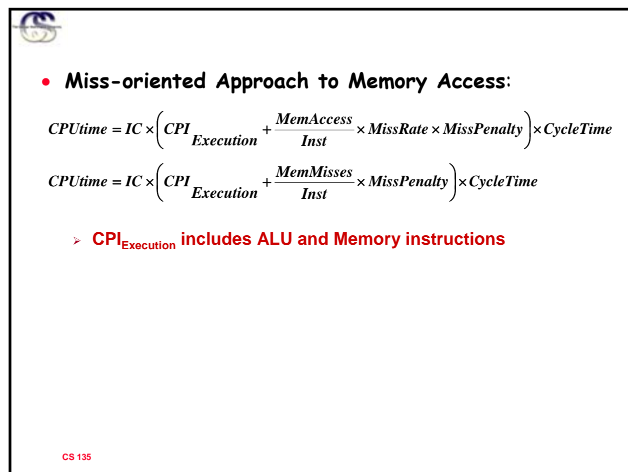

Write Policies: Analysis
Write through
Simple
Correctness easily maintained and no ambiguity about which copy of a block is current
Drawback is bandwidth required; memory access time
Must also decide on decision to fetch and allocate space for block to be written
Write allocate: fetch such a block and put in cache
Write-no-allocate: avoid fetch, and install blocks only on read misses
Good for cases of streaming writes which overwrite data
CS 135

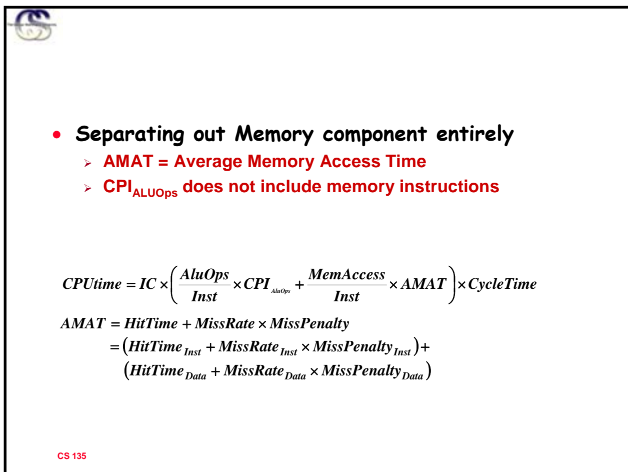

Modeling Cache Performance
CPU time equation….again!
CPU execution time =
(CPU clk cycles + Memory stall cycles) * clk cycle time.
Memory stall cycles =
number of misses * miss penalty = IC*(memory accesses/instruction)*miss rate* miss penalty
CS 135

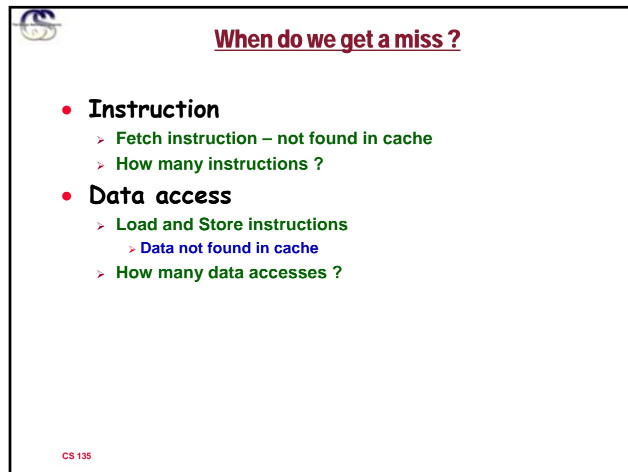

Cache Performance – Simplified Models
Hit rate/ratio r = number of requests that are hits/total num requests
Cost of memory access= rC_h + (1-r) C_m
C_h is cost/time from cache, C_m is cost/time when miss – fetch from memory
Extend to multiple levels
Hit ratios for level 1, level 2, etc.
Access times for level 1, level 2, etc.
$r_1 C_{h1} + r_2 C_{h2} + (1- r_1 - r_2) C_m$
CS 135

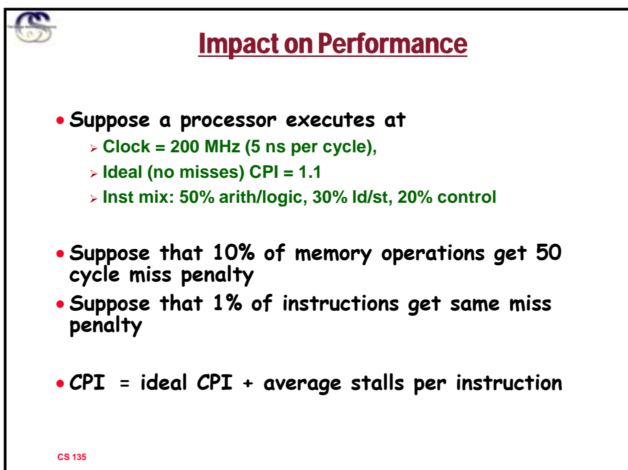

Average Memory Access Time
AMAT = HitTime + (1 - h) x MissPenalty
Hit time: basic time of every access.
Hit rate (h): fraction of access that hit
Miss penalty: extra time to fetch a block from lower level, including time to replace in CPU
CS 135

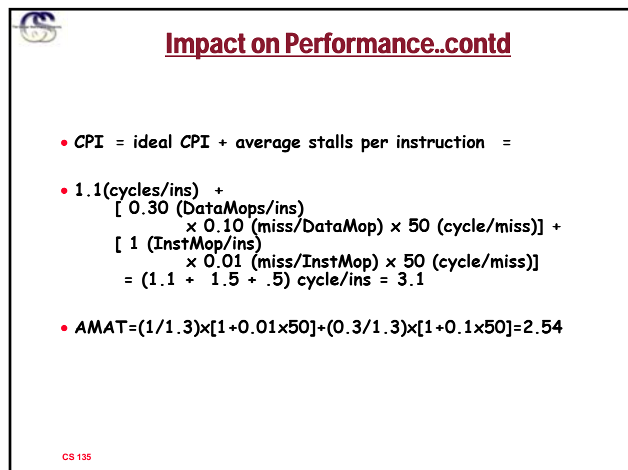

Memory stall cycles
Memory stall cycles: number of cycles that processor is stalled waiting for memory access
Performance in terms of mem stall cycles
CPU = (CPU cycles + Mem stall cycles)*Clk cycle time
Mem stall cycles = number of misses * miss penalty
= IC *(Misses/Inst) * Miss Penalty
= IC * (Mem accesses/Inst) * Miss Rate * penalty
Note: Read and Write misses combined into one miss rate
CS 135

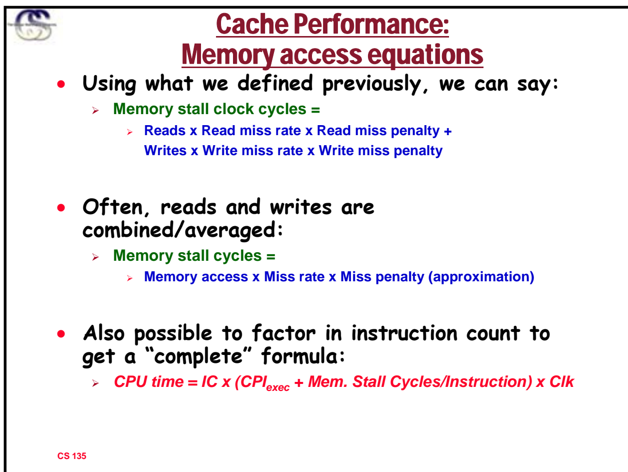

Miss-oriented Approach to Memory Access:
$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty\right) \times CycleTime$
$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalty\right) \times CycleTime$
$CPI_{Execution}$ includes ALU and Memory instructions
CS 135

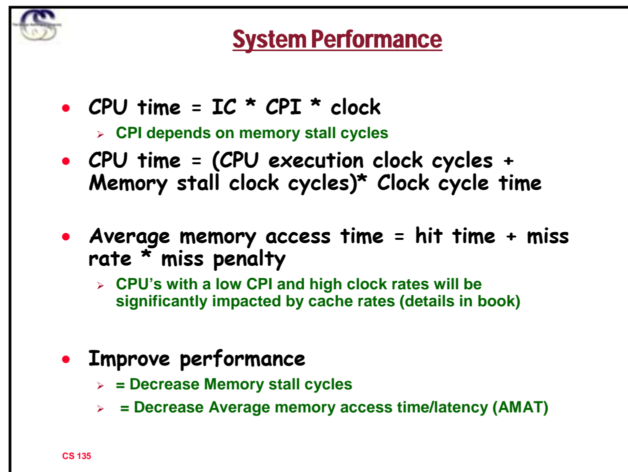

Separating out Memory component entirely
AMAT = Average Memory Access Time
CPI ALUOps does not include memory instructions
$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$
$AMAT = HitTime + MissRate \times MissPenalty$
$= (HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}) +$
$(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data})$
CS 135

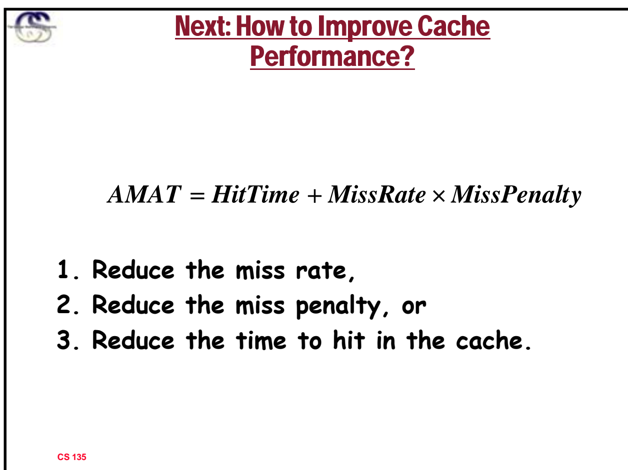

When do we get a miss ?
Instruction
Fetch instruction – not found in cache
How many instructions ?
Data access
Load and Store instructions
Data not found in cache
How many data accesses ?
CS 135

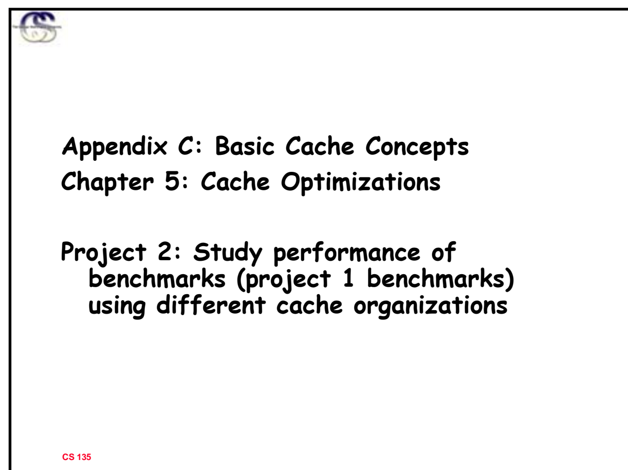

Impact on Performance
Suppose a processor executes at
Clock = 200 MHz (5 ns per cycle),
Ideal (no misses) CPI = 1.1
Inst mix: 50% arith/logic, 30% ld/st, 20% control
Suppose that 10% of memory operations get 50 cycle miss penalty
Suppose that 1% of instructions get same miss penalty
CPI = ideal CPI + average stalls per instruction
CS 135


Impact on Performance..contd
CPI = ideal CPI + average stalls per instruction =
1.1(cycles/ins) +
[0.30 (DataMops/ins)
x 0.10 (miss/DataMop) x 50 (cycle/miss)] +
[1 (InstMop/ins)
x 0.01 (miss/InstMop) x 50 (cycle/miss)]
= (1.1 + 1.5 + .5) cycle/ins = 3.1
AMAT=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54
CS 135


Cache Performance:
Memory access equations
Using what we defined previously, we can say:
Memory stall clock cycles =
Reads x Read miss rate x Read miss penalty +
Writes x Write miss rate x Write miss penalty
Often, reads and writes are combined/averaged:
Memory stall cycles =
Memory access x Miss rate x Miss penalty (approximation)
Also possible to factor in instruction count to get a "complete" formula:
CPU time = IC x (CPIexec + Mem. Stall Cycles/Instruction) x Clk
CS 135


System Performance
CPU time = IC * CPI * clock
CPI depends on memory stall cycles
CPU time = (CPU execution clock cycles + Memory stall clock cycles)* Clock cycle time
Average memory access time = hit time + miss rate * miss penalty
CPU's with a low CPI and high clock rates will be significantly impacted by cache rates (details in book)
Improve performance
= Decrease Memory stall cycles
= Decrease Average memory access time/latency (AMAT)
CS 135


Next: How to Improve Cache Performance?
AMAT = HitTime + MissRate × MissPenalty
1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.
CS 135


Appendix C: Basic Cache Concepts
Chapter 5: Cache Optimizations
Project 2: Study performance of benchmarks (project 1 benchmarks) using different cache organizations
CS 135

Cache Examples

Physical Address (10 bits)

Tag (6 bits) | Index (2 bits) | Offset (2 bits)

	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

A 4-entry direct mapped cache with 4 data words/block

1. Assume we want to read the following data words:

Tag	Index	Offset	Address Holds Data
101010	10	00	35_{10}
101010	10	01	24_{10}
101010	10	10	17_{10}
101010	10	11	25_{10}

All of these physical addresses would have the same tag

All of these physical addresses map to the same cache entry

2. If we read 101010 10 01 we want to bring data word 24_{10} into the cache.

Where would this data go? Well, the index is 10. Therefore, the data word will go somewhere into the 3rd block of the cache. (make sure you understand terminology)

More specifically, the data word would go into the 2nd position within the block – because the offset is '01'

3. The principle of spatial locality says that if we use one data word, we'll probably use some data words that are close to it – that's why our block size is bigger than one data word. So we fill in the data word entries surrounding 101010 10 01 as well.

CS 135

Physical Address (10 bits)

Tag (6 bits) | Index (2 bits) | Offset (2 bits)

	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

A 4-entry direct mapped cache with 4 data words/block

4. Therefore, if we get this pattern of accesses when we start a new program:

1.) 101010 10 00
2.) 101010 10 01
3.) 101010 10 10
4.) 101010 10 11

After we do the read for 101010 10 00 (word #1), we will automatically get the data for words #2, 3 and 4.

What does this mean? Accesses (2), (3), and (4) ARE NOT COMPULSORY MISSES

5. What happens if we get an access to location: 100011 | 10 | 11 (holding data: 12_{10})

Index bits tell us we need to look at cache block 10.

So, we need to compare the tag of this address – 100011 – to the tag that associated with the current entry in the cache block – 101010

These DO NOT match. Therefore, the data associated with address 100011 10 11 IS NOT VALID. What we have here could be:

- A compulsory miss
  - (if this is the 1st time the data was accessed)
- A conflict miss:
  - (if the data for address 100011 10 11 was present, but kicked out by 101010 10 00 – for example)

CS 135

Physical Address (10 bits)

Tag (6 bits) | Index (2 bits) | Offset (2 bits)

	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

This cache can hold 16 data words…

6. What if we change the way our cache is laid out – but so that it still has 16 data words? One way we could do this would be as follows:

	V	D	Tag	000	001	010	011	100	101	110	111
0											
1											

1 cache block entry

All of the following are true:
- This cache still holds 16 words
- Our block size is bigger – therefore this should help with compulsory misses
- Our physical address will now be divided as follows: Tag (6 bits) | Index (1 bit) | Offset (3 bits)
- The number of cache blocks has DECREASED
  - This will INCREASE the # of conflict misses

CS 135

7

What if we get the same pattern of accesses we had before?

	V	D	Tag	000	001	010	011	100	101	110	111
0											
1			101010	35_{10}	24_{10}	17_{10}	25_{10}	A_{10}	B_{10}	C_{10}	D_{10}

Note that there is now more data associated with a given cache block.

Pattern of accesses:
(note different # of bits for offset and index now)

1.) 101010 1 000
2.) 101010 1 001
3.) 101010 1 010
4.) 101010 1 011

However, now we have only 1 bit of index.
Therefore, any address that comes along that has a tag that is different than '101010' and has 1 in the index position is going to result in a conflict miss.

7

But, we could also make our cache look like this…

	V	D	Tag	0	1
000					
001					
010					
011					
100			101010	35_{10}	24_{10}
101			101010	17_{10}	25_{10}
110					
111					

There are now just 2 words associated with each cache block.

Again, let's assume we want to read the following data words:

	Tag	Index	Offset	Address Holds Data
1.)	101010	100	0	35_{10}
2.)	101010	100	1	24_{10}
3.)	101010	101	0	17_{10}
4.)	101010	101	1	25_{10}

Assuming that all of these accesses were occurring for the 1st time (and would occur sequentially), accesses (1) and (3) would result in compulsory misses, and accesses would result in hits because of spatial locality. (The final state of the cache is shown after all 4 memory accesses).

Note that by organizing a cache in this way, conflict misses will be reduced.
There are now more addresses in the cache that the 10-bit physical address can map too.

	V	D	Tag	0	1
000					
001					
010					
011					
100			101010	35_{10}	24_{10}
101			101010	17_{10}	25_{10}
110					
111					

	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

8

All of these caches hold exactly the same amount of data – 16 different word entries

	V	D	Tag	000	001	010	011	100	101	110	111
0											
1			101010	35_{10}	24_{10}	17_{10}	25_{10}	A_{10}	B_{10}	C_{10}	D_{10}

As a general rule of thumb, "long and skinny" caches help to reduce conflict misses, "short and fat" caches help to reduce compulsory misses, but a cross between the two is *probably* what will give you the best (i.e. lowest) overall miss rate.

But what about capacity misses?

8

What's a capacity miss?

- The cache is only so big. We won't be able to store every block accessed in a program – must them swap out!
- Can avoid capacity misses by making cache bigger

	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

	V	D	Tag	00	01	10	11
000							
001							
010			10101	35_{10}	24_{10}	17_{10}	25_{10}
011							
100							
101							
110							
111							

Thus, to avoid capacity misses, we'd need to make our cache physically bigger – i.e. there are now 32 word entries for it instead of 16.

FYI, this will change the way the physical address is divided. Given our original pattern of accesses, we'd have:

Pattern of accesses:

1.) 10101 | 010 | 00 = 35_{10}
2.) 10101 | 010 | 01 = 24_{10}
3.) 10101 | 010 | 10 = 17_{10}
4.) 10101 | 010 | 11 = 25_{10}

(note smaller tag, bigger index)

End of Examples

Next: Cache Optimization

- Techniques for minimizing AMAT
  - Miss rate
  - Miss penalty
  - Hit time
- Role of compiler ?

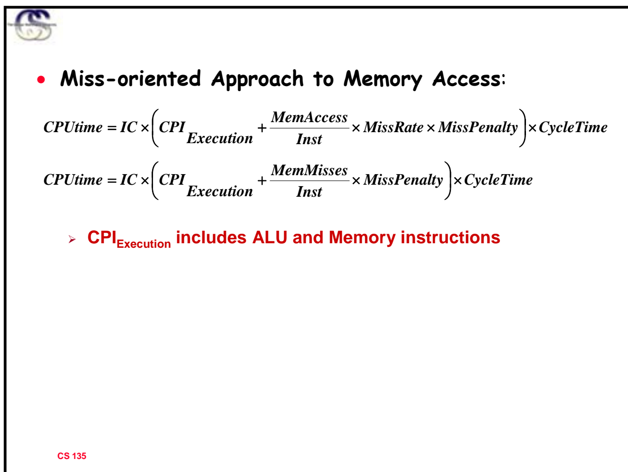
Improving Cache Performance

Memory stall cycles

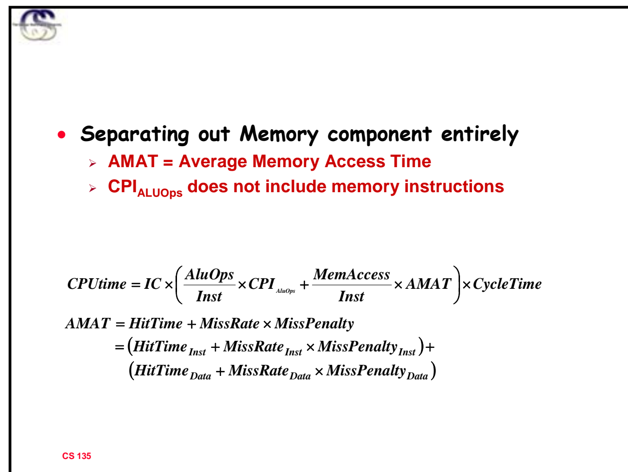
- Memory stall cycles: number of cycles that processor is stalled waiting for memory access
- Performance in terms of mem stall cycles
  - *CPU = (CPU cycles + Mem stall cycles)*Clk cycle time*
  - *Mem stall cycles = number of misses * miss penalty*

    *= IC *(Misses/Inst) * Miss Penalty*

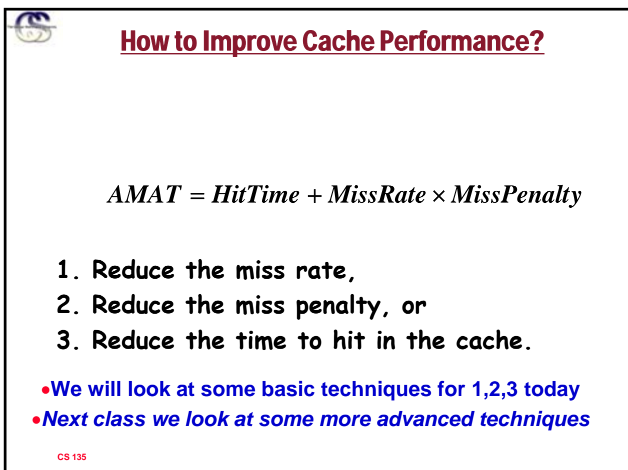
    *= IC * (Mem accesses/Inst) * Miss Rate * penalty*
    - **Note: Read and Write misses combined into one miss rate**



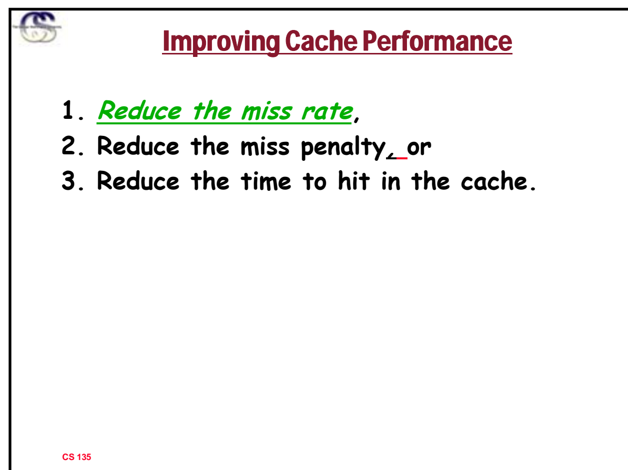
Miss-oriented Approach to Memory Access:
$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty\right) \times CycleTime$$
$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalty\right) \times CycleTime$$
$CPI_{Execution}$ includes ALU and Memory instructions
CS 135



Separating out Memory component entirely
AMAT = Average Memory Access Time
CPI_{ALUOps} does not include memory instructions
$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT\right) \times CycleTime$$
$$AMAT = HitTime + MissRate \times MissPenalty$$
$$= (HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}) + (HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data})$$
CS 135



How to Improve Cache Performance?
$$AMAT = HitTime + MissRate \times MissPenalty$$
1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.
We will look at some basic techniques for 1,2,3 today
Next class we look at some more advanced techniques
CS 135



Improving Cache Performance
1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.
CS 135

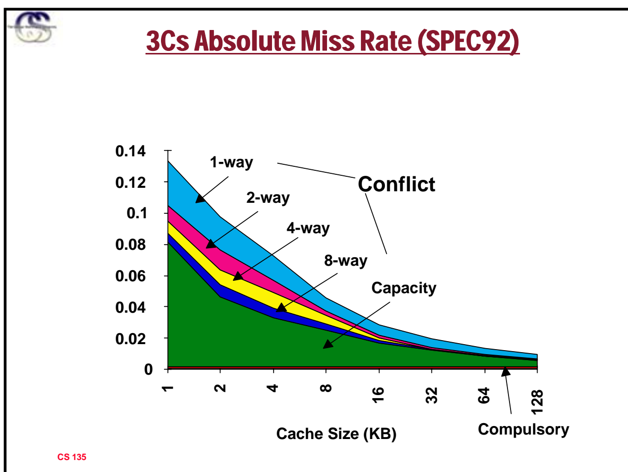
Cache Misses

- Latencies at each level determined by technology
- Miss rates are function of
  - Organization of cache
  - Access patterns of the program/application
- Understanding causes of misses enables designer to realize shortcomings of design and discover creative cost-effective solutions
  - The 3Cs model is a tool for classifying cache misses based on underlying cause

Reducing Miss Rate – 3C's Model

- Compulsory (or cold) misses
  - Due to program's first reference to a block –not in cache so must be brought to cache (cold start misses)
    - These are Fundamental misses – cannot do anything
- Capacity
  - Due to insufficient capacity - if the cache cannot contain all the blocks needed capacity misses will occur because of blocks being discarded and later retrieved)
    - Not fundamental, and by-product of cache organization
    - Increasing capacity can reduce some misses
- Conflict
  - Due to imperfect allocation; if the block placement strategy is set associative or direct mapped, conflict misses will occur because a block may be discarded and later retrieved if too many blocks map to its set. *Interference or collision misses*
    - Not fundamental; by-product of cache organization; fully associative can eliminate conflict misses

3Cs Absolute Miss Rate (SPEC92)



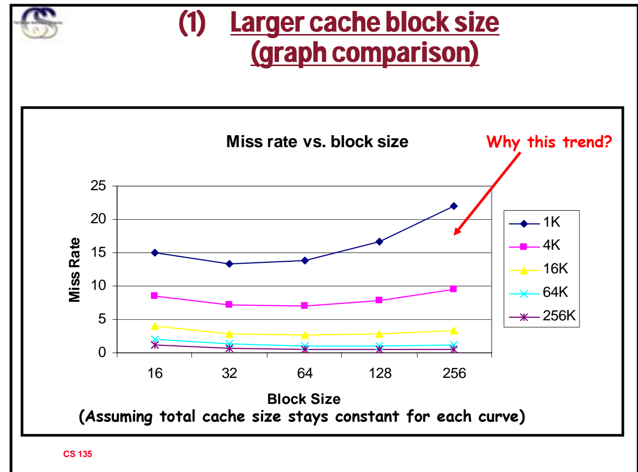
Miss Rate Reduction Strategies

- Increase block size – reduce compulsory misses
- Larger caches
  - Larger size can reduce capacity, conflict misses
  - Larger block size for fixed total size can lead to more capacity misses
  - Can reduce conflict misses
- Higher associativity
  - Can reduce conflict misses
  - No effect on cold miss
- Compiler controlled pre-fetching (faulting/non-faulting)
  - Code reorganization
    - Merging arrays
    - Loop interchange (row column order)
    - Loop fusion (2 loops into 1)
    - Blocking

Larger cache block size

- Easiest way to reduce miss rate is to increase cache block size
  - This will help eliminate what kind of misses?
- Helps improve miss rate b/c of principle of locality:
  - Temporal locality says that if something is accessed once, it'll probably be accessed again soon
  - Spatial locality says that if something is accessed, something nearby it will probably be accessed
    - Larger block sizes help with spatial locality
- Be careful though!
  - Larger block sizes can increase miss penalty!
    - Generally, larger blocks reduce # of total blocks in cache

(1) Larger cache block size (graph comparison)



Larger cache block size (example)

- Assume that to access lower-level of memory hierarchy you:
  - Incur a 40 clock cycle overhead
  - Get 16 bytes of data every 2 clock cycles
- I.e. get 16 bytes in 42 clock cycles, 32 in 44, etc…
- Using data below, which block size has minimum average memory access time?

Cache sizes

Block Size	1K	4K	16K	64K	256K
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

Miss rates

Larger cache block size (ex. continued…)

- Recall that Average memory access time =
  - Hit time + Miss rate X Miss penalty
- Assume a cache hit otherwise takes 1 clock cycle –independent of block size
- So, for a 16-byte block in a 1-KB cache…
  - Average memory access time =
    - 1 + (15.05% X 42) = 7.321 clock cycles
- And for a 256-byte block in a 256-KB cache…
  - Average memory access time =
    - 1 + (0.49% X 72) = 1.353 clock cycles
- Rest of the data is included on next slide…

Larger cache block size (ex. continued)

Cache sizes

Block Size	Miss Penalt	1K	4K	16K	64K	256K
16	42	7.321	4.599	2.655	1.857	1.485
32	44	6.870	4.186	2.263	1.594	1.308
64	48	7.605	4.360	2.267	1.509	1.245
128	56	10.318	5.357	2.551	1.571	1.274
256	72	16.847	7.847	3.369	1.828	1.353

Red entries are lowest average time for a particular configuration

Note: All of these block sizes are common in processor's today
Note: Data for cache sizes in units of "clock cycles"

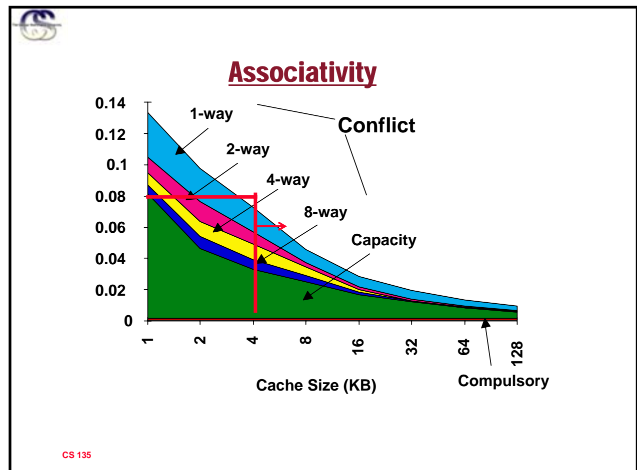
Larger cache block sizes (wrap-up)

- We want to minimize cache miss rate & cache miss penalty at same time!
- Selection of block size depends on latency and bandwidth of lower-level memory:
  - High latency, high bandwidth encourage large block size
    - Cache gets many more bytes per miss for a small increase in miss penalty
  - Low latency, low bandwidth encourage small block size
    - Twice the miss penalty of a small block may be close to the penalty of a block twice the size
    - Larger # of small blocks may reduce conflict misses

Higher associativity

- Higher associativity can improve cache miss rates…
- Note that an 8-way set associative cache is…
  - …essentially a fully-associative cache
- Helps lead to 2:1 cache rule of thumb:
  - It says:
    - A direct mapped cache of size *N* has about the same miss rate as a 2-way set-associative cache of size *N*/2
- But, diminishing returns set in sooner or later…
  - Greater associativity can cause increased hit time

Associativity



Miss Rate Reduction Strategies

- Increase block size – reduce compulsory misses
- Larger caches
  - Larger size can reduce capacity, conflict misses
  - Larger block size for fixed total size can lead to more capacity misses
  - Can reduce conflict misses
- Higher associativity
  - Can reduce conflict misses
  - No effect on cold miss
- Compiler controlled pre-fetching (faulting/non-faulting)
  - Code reorganization
    - Merging arrays
    - Loop interchange (row column order)
    - Loop fusion (2 loops into 1)
    - Blocking

CS 135

Larger Block Size (fixed size&assoc)

25%
20%
15%
10%
5%
0%
Miss Rate
16
32
64
128
256
Block Size (bytes)
1K
4K
16K
64K
256K
Reduced compulsory misses
Increased Conflict Misses

What else drives up block size?

CS 135

Cache Size

0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
1-way
2-way
4-way
8-way
Capacity
1
2
4
8
16
32
64
128
Cache Size (KB)
Compulsory

- Old rule of thumb: 2x size => 25% cut in miss rate
- What does it reduce?

CS 135

How to Improve Cache Performance?

$$AMAT = HitTime + MissRate \times MissPenalty$$

1. Reduce the miss rate,
2. *Reduce the miss penalty*, or
3. Reduce the time to hit in the cache.

CS 135

Improving Performance: Reducing Cache Miss Penalty

- Multilevel caches –
  - second and subsequent level caches can be large enough to capture many accesses that would have gone to main memory, and are faster (therefore less penalty)
- Critical word first and early restart –
  - don't wait for full block of cache to be loaded, send the critical word first, restart the CPU and continue the load
- Priority to read misses over write misses
- Merging write buffer –
  - if the address of a new entry matches that of one in the write buffer, combine it
- Victim Caches –
  - cache discarded blocks elsewhere
    - Remember what was discarded in case it is needed again
  - Insert small fully associative cache between cache and its refill path
    - This "victim cache" contains only blocks that were discarded as a result of a cache miss (replacement policy)
    - Check victim cache in case of miss before going to next lower level of memory

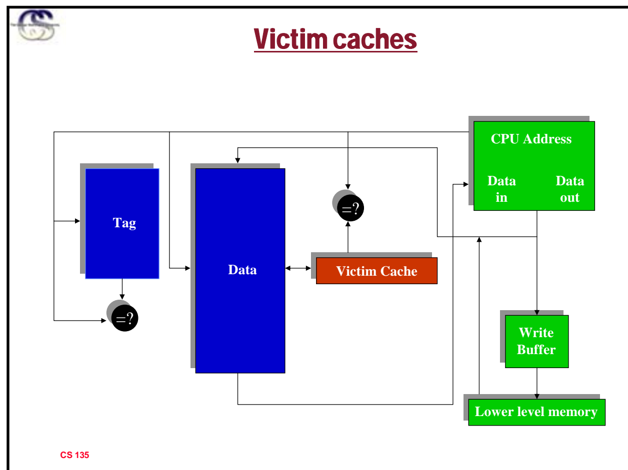
Early restart and critical word 1st

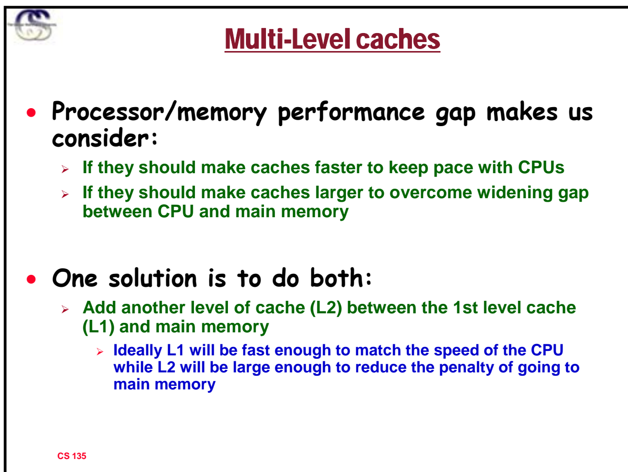
- With this strategy we're going to be impatient
  - As soon as some of the block is loaded, see if the data is there and send it to the CPU
    - (i.e. we don't wait for the whole block to be loaded)
- There are 2 general strategies:
  - Early restart:
    - As soon as the word gets to the cache, send it to the CPU
  - Critical word first:
    - Specifically ask for the needed word 1st, make sure it gets to the CPU, then get the rest of the cache block data

Victim caches

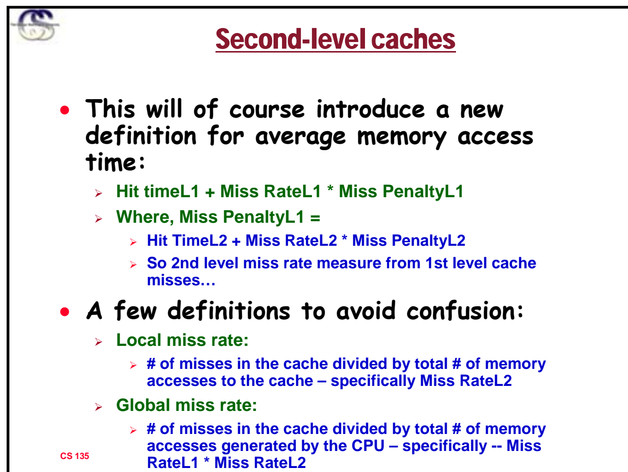
- 1st of all, what is a "victim cache"?
  - A victim cache temporarily stores blocks that have been discarded from the main cache (usually not that big) – due to conflict misses
- 2nd of all, how does it help us?
  - If there's a cache miss, instead of immediately going down to the next level of memory hierarchy we check the victim cache first
  - If the entry is there, we swap the victim cache block with the actual cache block
- Research shows:
  - Victim caches with 1-5 entries help reduce conflict misses
  - For a 4KB direct mapped cache victim caches:
    - Removed 20% - 95% of conflict misses!

Victim caches

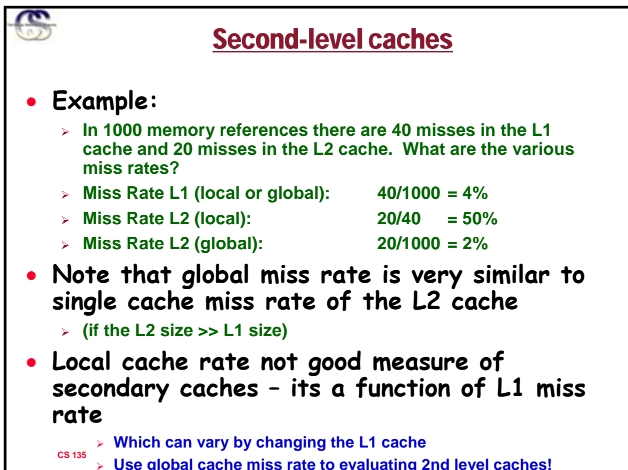




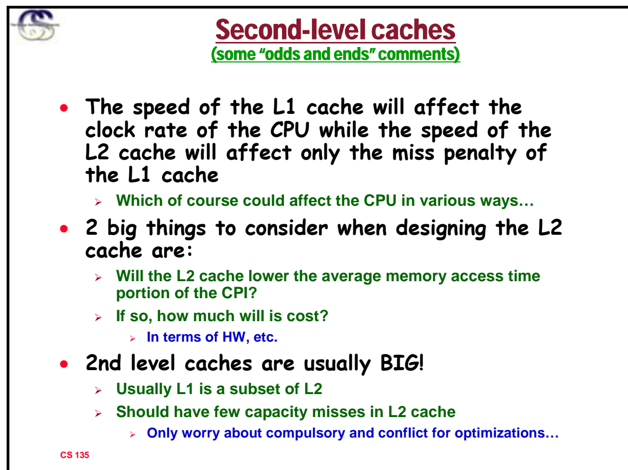
Multi-Level caches
Processor/memory performance gap makes us consider:
If they should make caches faster to keep pace with CPUs
If they should make caches larger to overcome widening gap between CPU and main memory
One solution is to do both:
Add another level of cache (L2) between the 1st level cache (L1) and main memory
Ideally L1 will be fast enough to match the speed of the CPU while L2 will be large enough to reduce the penalty of going to main memory
CS 135



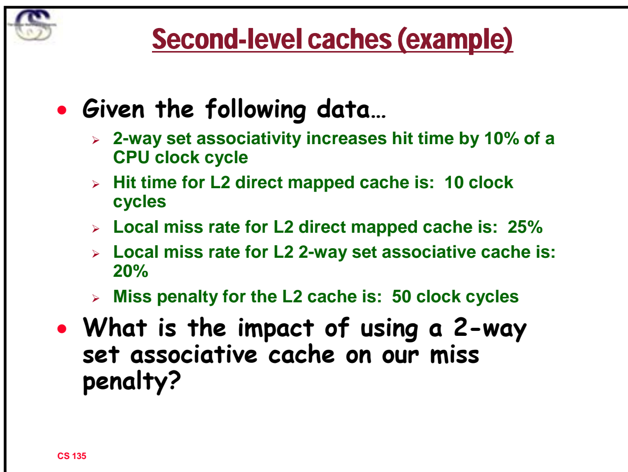
Second-level caches
This will of course introduce a new definition for average memory access time:
Hit timeL1 + Miss RateL1 * Miss PenaltyL1
Where, Miss PenaltyL1 =
Hit TimeL2 + Miss RateL2 * Miss PenaltyL2
So 2nd level miss rate measure from 1st level cache misses…
A few definitions to avoid confusion:
Local miss rate:
# of misses in the cache divided by total # of memory accesses to the cache – specifically Miss RateL2
Global miss rate:
# of misses in the cache divided by total # of memory accesses generated by the CPU – specifically -- Miss RateL1 * Miss RateL2
CS 135

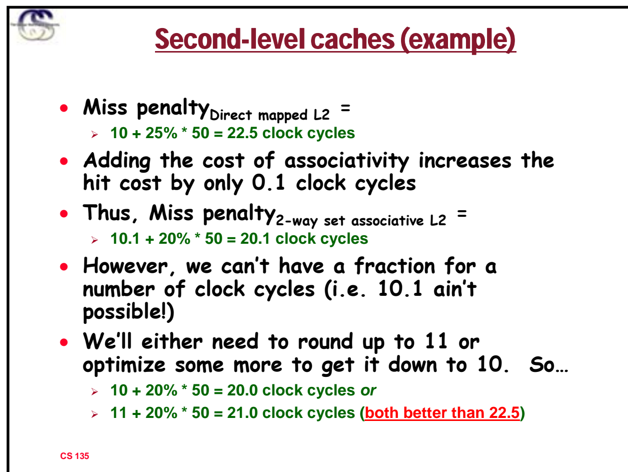


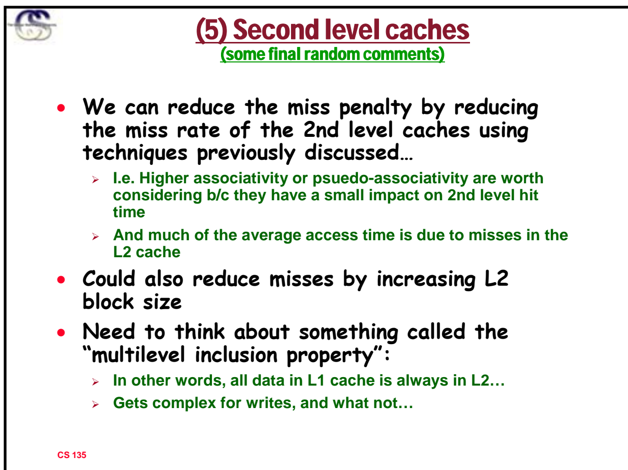
Second-level caches
Example:
In 1000 memory references there are 40 misses in the L1 cache and 20 misses in the L2 cache. What are the various miss rates?
Miss Rate L1 (local or global): 40/1000 = 4%
Miss Rate L2 (local): 20/40 = 50%
Miss Rate L2 (global): 20/1000 = 2%
Note that global miss rate is very similar to single cache miss rate of the L2 cache
(if the L2 size >> L1 size)
Local cache rate not good measure of secondary caches – its a function of L1 miss rate
Which can vary by changing the L1 cache
Use global cache miss rate to evaluating 2nd level caches!
CS 135

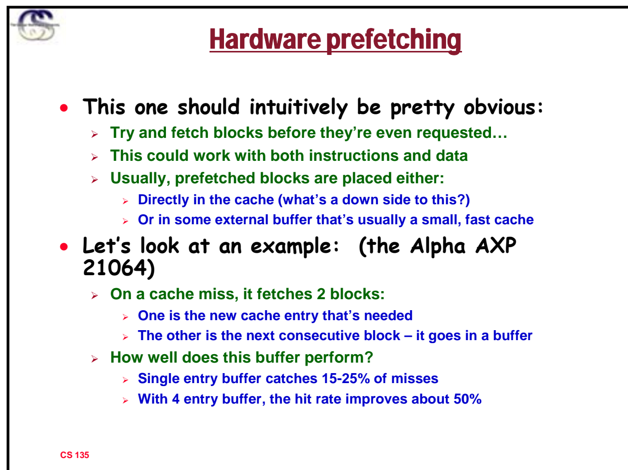


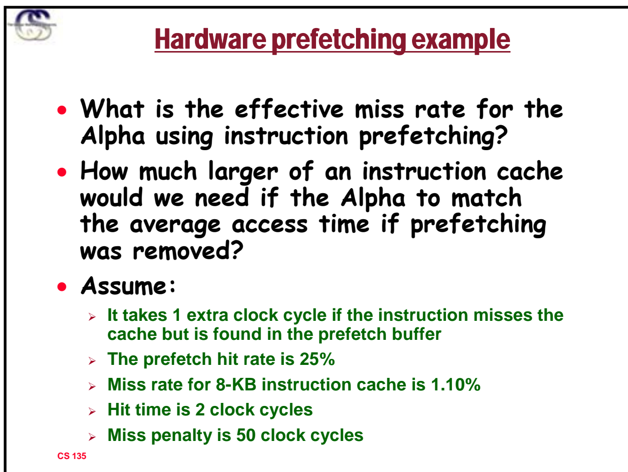
Second-level caches
(some "odds and ends" comments)
The speed of the L1 cache will affect the clock rate of the CPU while the speed of the L2 cache will affect only the miss penalty of the L1 cache
Which of course could affect the CPU in various ways…
2 big things to consider when designing the L2 cache are:
Will the L2 cache lower the average memory access time portion of the CPI?
If so, how much will is cost?
In terms of HW, etc.
2nd level caches are usually BIG!
Usually L1 is a subset of L2
Should have few capacity misses in L2 cache
Only worry about compulsory and conflict for optimizations…
CS 135

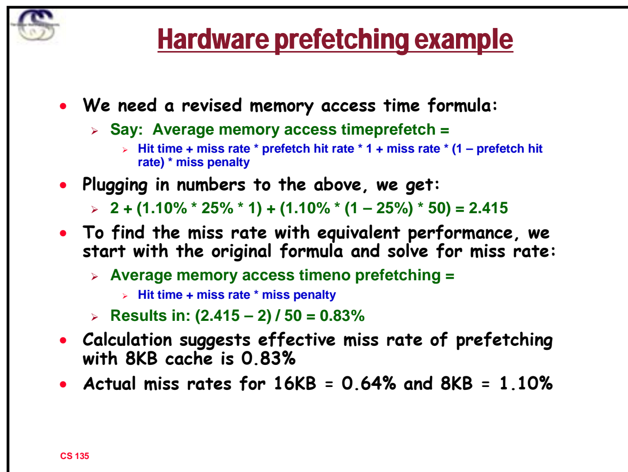

Second-level caches (example)
Given the following data…
2-way set associativity increases hit time by 10% of a CPU clock cycle
Hit time for L2 direct mapped cache is: 10 clock cycles
Local miss rate for L2 direct mapped cache is: 25%
Local miss rate for L2 2-way set associative cache is: 20%
Miss penalty for the L2 cache is: 50 clock cycles
What is the impact of using a 2-way set associative cache on our miss penalty?
CS 135

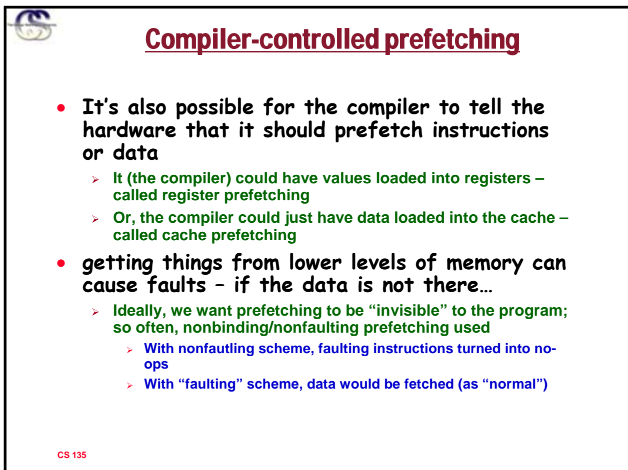

Second-level caches (example)
Miss penalty$_{\text{Direct mapped L2}}$ =
10 + 25% * 50 = 22.5 clock cycles
Adding the cost of associativity increases the hit cost by only 0.1 clock cycles
Thus, Miss penalty$_{\text{2-way set associative L2}}$ =
10.1 + 20% * 50 = 20.1 clock cycles
However, we can't have a fraction for a number of clock cycles (i.e. 10.1 ain't possible!)
We'll either need to round up to 11 or optimize some more to get it down to 10. So…
10 + 20% * 50 = 20.0 clock cycles or
11 + 20% * 50 = 21.0 clock cycles (both better than 22.5)
CS 135

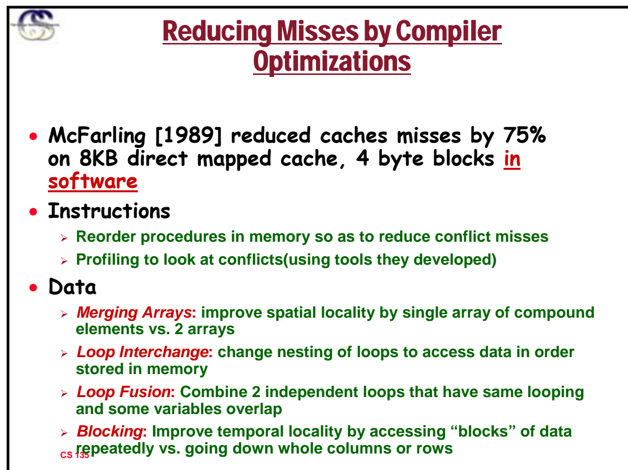

(5) Second level caches
(some final random comments)
We can reduce the miss penalty by reducing the miss rate of the 2nd level caches using techniques previously discussed…
I.e. Higher associativity or psuedo-associativity are worth considering b/c they have a small impact on 2nd level hit time
And much of the average access time is due to misses in the L2 cache
Could also reduce misses by increasing L2 block size
Need to think about something called the "multilevel inclusion property":
In other words, all data in L1 cache is always in L2…
Gets complex for writes, and what not…
CS 135


Hardware prefetching
This one should intuitively be pretty obvious:
Try and fetch blocks before they're even requested…
This could work with both instructions and data
Usually, prefetched blocks are placed either:
Directly in the cache (what's a down side to this?)
Or in some external buffer that's usually a small, fast cache
Let's look at an example: (the Alpha AXP 21064)
On a cache miss, it fetches 2 blocks:
One is the new cache entry that's needed
The other is the next consecutive block – it goes in a buffer
How well does this buffer perform?
Single entry buffer catches 15-25% of misses
With 4 entry buffer, the hit rate improves about 50%
CS 135


Hardware prefetching example
What is the effective miss rate for the Alpha using instruction prefetching?
How much larger of an instruction cache would we need if the Alpha to match the average access time if prefetching was removed?
Assume:
It takes 1 extra clock cycle if the instruction misses the cache but is found in the prefetch buffer
The prefetch hit rate is 25%
Miss rate for 8-KB instruction cache is 1.10%
Hit time is 2 clock cycles
Miss penalty is 50 clock cycles
CS 135


Hardware prefetching example
We need a revised memory access time formula:
Say: Average memory access timeprefetch =
Hit time + miss rate * prefetch hit rate * 1 + miss rate * (1 – prefetch hit rate) * miss penalty
Plugging in numbers to the above, we get:
2 + (1.10% * 25% * 1) + (1.10% * (1 – 25%) * 50) = 2.415
To find the miss rate with equivalent performance, we start with the original formula and solve for miss rate:
Average memory access timeno prefetching =
Hit time + miss rate * miss penalty
Results in: (2.415 – 2) / 50 = 0.83%
Calculation suggests effective miss rate of prefetching with 8KB cache is 0.83%
Actual miss rates for 16KB = 0.64% and 8KB = 1.10%
CS 135


Compiler-controlled prefetching
It's also possible for the compiler to tell the hardware that it should prefetch instructions or data
It (the compiler) could have values loaded into registers – called register prefetching
Or, the compiler could just have data loaded into the cache – called cache prefetching
getting things from lower levels of memory can cause faults – if the data is not there…
Ideally, we want prefetching to be "invisible" to the program; so often, nonbinding/nonfaulting prefetching used
With nonfautling scheme, faulting instructions turned into no-ops
With "faulting" scheme, data would be fetched (as "normal")
CS 135


Reducing Misses by Compiler Optimizations
McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
Instructions
Reorder procedures in memory so as to reduce conflict misses
Profiling to look at conflicts(using tools they developed)
Data
Merging Arrays: improve spatial locality by single array of compound elements vs. 2 arrays
Loop Interchange: change nesting of loops to access data in order stored in memory
Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap
Blocking: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows
CS 135

Compiler optimizations – merging arrays

- This works by improving spatial locality
- For example, some programs may reference multiple arrays of the same size at the same time
  - Could be bad:
    - Accesses may interfere with one another in the cache
- A solution: Generate a single, compound array…

```
/* Before:*/
int tag[SIZE]
int byte1[SIZE]
int byte2[SIZE]
int dirty[size]
```

```
/* After */
struct merge {
        int tag;
        int byte1;
        int byte2;
        int dirty;
}
struct merge cache_block_entry[SIZE]
```

CS 135

Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of stuctures */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key; improve spatial locality

CS 135

Compiler optimizations – loop interchange

- Some programs have nested loops that access memory in non-sequential order
  - Simply changing the order of the loops may make them access the data *in* sequential order…
- What's an example of this?

```
/* Before:*/
for( j = 0; j < 100; j= j + 1) {
        for( k = 0; k < 5000; k = k + 1) {
                x[k][j] = 2 * x[k][j];

/* After:*/
for( k = 0; k < 5000; k= k + 1) {
        for( j = 0; j < 5000; j = j + 1) {
                x[k][j] = 2 * x[k][j];
```

But who really writes loops like this???

CS 135

Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
      for (i = 0; i < 5000; i = i+1)
          x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
      for (j = 0; j < 100; j = j+1)
          x[i][j] = 2 * x[i][j];
```

Sequential accesses instead of striding through memory every 100 words; improved spatial locality

CS 135

Compiler optimizations – loop fusion

- This one's pretty obvious once you hear what it is…
- Seeks to take advantage of:
  - Programs that have separate sections of code that access the same arrays in different loops
    - Especially when the loops use common data
  - The idea is to "fuse" the loops into one common loop
- What's the target of this optimization?
- Example:

```
/* Before:*/
for( j = 0; j < N; j= j + 1) {
        for( k = 0; k < N; k = k + 1) {
                a[j][k] = 1/b[j][k] * c[j][k];
for( j = 0; j < N; j= j + 1) {
        for( k = 0; k < N; k = k + 1) {
                d[j][k] = a[j][k] + c[j][k];
```

```
/* After:*/
for( j = 0; j < N; j= j + 1) {
        for( k = 0; k < N; k = k + 1) {
                a[j][k] = 1/b[j][k] * c[j][k];
                d[j][k] = a[j][k] + c[j][k];
        }
}
```


Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {   a[i][j] = 1/b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];}
```

2 misses per access to a & c vs. one miss per access; improve spatial locality

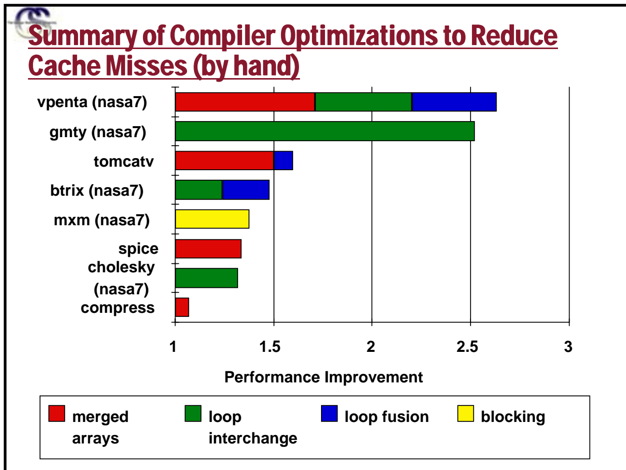
Compiler optimizations – blocking

- This is probably the most "famous" of compiler optimizations to improve cache performance
- Tries to reduce misses by improving temporal locality
- To get a handle on this, you have to work through code on your own
  - Homework!
- this is used mainly with arrays!
- Simplest case??
  - Row-major access

Blocking Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
     {r = 0;
      for (k = 0; k < N; k = k+1){
        r = r + y[i][k]*z[k][j];};
      x[i][j] = r;
     };
```

- Two Inner Loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]
- Capacity Misses a function of N & Cache Size:
  - $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits



Improving Cache Performance

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. *Reduce the time to hit in the cache*.

Reducing the hit time

- Again, recall our average memory access time equation:
  - Hit time + Miss Rate * Miss Penalty
  - We've talked about reducing the Miss Rate and the Miss Penalty – Hit time can also be a big component…
- On many machines cache accesses can affect the clock cycle time – so making this small is a good thing!

Small and simple caches

- Why is this good?
  - Generally, smaller hardware is faster – so a small cache should help the hit time…
  - If an L1 cache is small enough, it should fit on the same chip as the actual processing logic…
    - Processor avoids time going off chip!
    - Some designs compromise and keep tags on a chip and data off chip – allows for fast tag check and >> memory capacity
  - Direct mapping also falls under the category of "simple"
    - Relates to point above as well – you can check tag and read data at the same time!

Avoid address translation during cache indexing

- This problem centers around virtual addresses. Should we send the virtual address to the cache?
  - In other words we have Virtual caches vs. Physical caches
  - Why is this a problem anyhow?
    - Well, recall from OS that a processor usually deals with processes
    - What if process 1 uses a virtual address xyz and process 2 uses the same virtual address?
    - The data in the cache would be totally different! – called aliasing
- Every time a process is switched logically, we'd have to flush the cache or we'd get false hits.
  - Cost = time to flush + compulsory misses from empty cache
- I/O must interact with caches so we need virtual addressess

Separate Instruction and Data Cache

- Multilevel cache is one option for design
- Another view:
  - Separate the instruction and data caches
  - Instead of a Unified Cache, have separate I-cache and D-cache
    - Problem: What size does each have ?

Separate I-cache and D-cache example:

- We want to compare the following:
  - A 16-KB data cache & a 16-KB instruction cache versus a 32-KB unified cache

Size	Inst. Cache	Data Cache	Unified Cache
16 KB	0.64%	6.47%	2.87%
32 KB	0.15%	4.82%	1.99%

Miss Rates

- Assume a hit takes 1 clock cycle to process
- Miss penalty = 50 clock cycles
- In unified cache, load or store hit takes 1 extra clock cycle b/c having only 1 cache port = a structural hazard
- 75% of accesses are instruction references
- What's avg. memory access time in each case?

example continued…

- 1st, need to determine overall miss rate for split caches:
  - (75% x 0.64%) + (25% x 6.47%) = 2.10%
  - This compares to the unified cache miss rate of 1.99%
- We'll use average memory access time formula from a few slides ago but break it up into instruction & data references
- Average memory access time – split cache =
  - 75% x (1 + 0.64% x 50) + 25% x (1 + 6.47% x 50)
  - (75% x 1.32) + (25% x 4.235) = 2.05 cycles
- Average memory access time – unified cache =
  - 75% x (1 + 1.99% x 50) + 25% x (1 + 1 + 1.99% x 50)
  - (75% x 1.995) + (25% x 2.995) = 2.24 cycles
- Despite higher miss rate, access time faster for split cache!

Reducing Time to Hit in Cache: Trace Cache

- **Trace caches**
  - ILP technique
  - Trace cache finds dynamic sequence of instructions including taken branches to load into cache block
    - Branch prediction is folded into the cache

The Trace Cache Proposal

Cache Summary

- **Cache performance crucial to overall performance**
- **Optimize performance**
  - Miss rates
  - Miss penalty
  - Hit time
- **Software optimizations can lead to improved performance**
- **Next . . . Code Optimization in Compilers**