# Causes of misses

- 3C's model
- Classifying misses
- (1) Compulsory
  - First access to a block not in cache called cold-start misses or first reference miss
  - Misses even in an infinite cache
- (2) Capacity
  - Cache cannot contain all needed blocks, Block discarded and retrieved later
  - Misses in a fully associative cache
- (3) Conflict
  - Placement in set-associative or direct map
  - More than 1 block maps to same set/location
  - Collision misses or interference misses
  - Misses in N-way assoc size cache
- (4) Coherence
  - More recent 4th C
  - Due to cache flishes to keep multiple cache coherent in multi-processors

# Effect of 3C's

- Reduce the compulsory misses with large block size
  - Over a period of time less effective
- Reduce capacity misses
  - By large cache size
- Reduce conflict misses
  - By high associativity
- Change in cache size
  - Changes capacity as well as conflict misses
  - As references spread out
  - Thus miss can move from capacity -> to -> conflict
- Techniques that reduce miss rate increase the hit time or miss penalty
- Seek a balance in all options to increase performance

# Larger block size to reduce miss rate

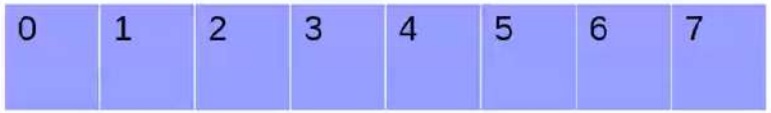
- Large block size also reduces compulsory misses as more elements brought together (spatial locality)
  - Miss penalty increases as we need to bring a large block
  - Increases conflict miss as large block, so less num of blocks in cache if small cache
  - No benefit in this option if it increases AMAT (as miss penalty may go up)
  - Solution of blk size depends on latency+bandwidth of lower-level memory
- High latency + high b/w -> encourages larger blocks, as cache gets more bytes per miss for a small increase in miss penalty
- Low latency + low b/w -> smaller blks, since there is little time saved from a larger block
- e.g. twice miss penalty of small block maybe approx-equal-to penalty of block twice the size

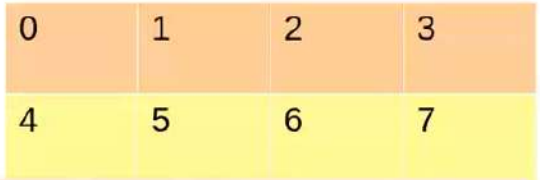
# Larger cache to reduce miss rate

- Obvious way to reduce miss rate
- Drawback: increases hit-time and higher cost + power
- This technique is used for off-chip caches

# Higher assoc to lower miss rate

- Two rules of thumb
  - (1) 8-way set-associative cache is as effective as fully associative in reducing miss rate
  - (2) 2:1 cache rule of thumb: Direct map cache of size N has about the same miss rate as a 2-way set assoc cache of size N/2




1-way : Direct
Look at addresses mapped in each cell

2-way with set-0 and set-1 having 2 blocks each
... 1:2 i.e. N vs N/2

# Higher assoc to lower miss rate

- Two rules of thumb
  - (1) 8-way set-associative cache is as effective as fully associative in reducing miss rate
  - (2) 2:1 cache rule of thumb: Direct map cache of size N has about the same miss rate as a 2-way set assoc cache of size N/2
- Block size increase -> miss rate decrease -> miss penalty increase
- Assoc increase -> miss rate decrease -> hit time increase
- Fast processor clock cycle -> simple cache design
- But to reduce miss penalty assoc can help

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|

Set-0

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 4 | 5 | 6 | 7 |

1-way : Direct
Look at addresses mapped in each cell

2-way with set-0 and set-1 having 2 blocks each
... 1:2 i.e. N vs N/2

# Multi-level cache to reduce miss rate

- Technology making processor faster than DRAM can catch up
- Miss penalty increases over time
- Solution
  - (1) fast cache to match processor speed
  - (2) large cache to avoid widening gap between processor and main memory
- Do-BOTH: how? Add 2nd level caches
  - First level = small + fast
  - Second level = large to capture accesses going to main memory thus reducing miss penalty

AMAT = Hit time_L1 + Miss rate_L1 x Miss Penalty_L1
Miss Penalty_L1 = Hit time_L2 + Miss rate_L2 x Miss Penalty_L2

# Mutli level ...

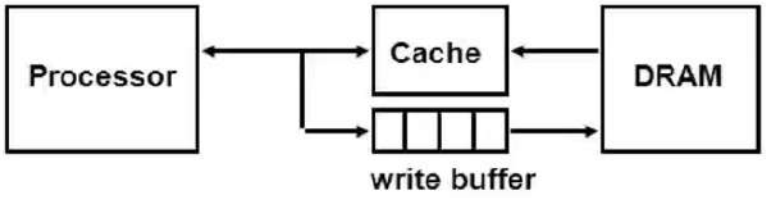
- Miss rate of L2 is measures from left overs of L1 cache
- Local miss rate = num of misses / tot access to this cache
  - Local L1 = miss rate L1
  - Local L2 = miss rate L2
  - Large for L2, as L1 cache skims the cream of mem-accesses
- Global miss rate = num of misses / tot mem accesses generated by processor
  - Global L1 = miss rate L1
  - Global L2 = miss rate L1 x miss rate L2
  - Useful measure as it indicates what fraction of mem-references go all the way to main memory
- Global miss rate is same as single level cache miss rate if L2 cache much larger than L1
- Local miss rate of L2 is a function of L1 cache and will vary as L1 varies
  - Therefore use global miss-rate to evaluate L2-cache

# Priority to reads

- Give priority to read misses over write misses to reduce miss penalty
- Serve reads before writes have completed
- Use write buffer
  - But this buffer complicates matters as it may hold latest data values
- In wr-thru, no-wr allocate
  - Stall read miss until wr-buffer is empty, or
  - Check contents of wr-buffer first, if match OK else go to memory
- For wr-back
  - For read miss if block to remove is dirty then put it in write buffer and proceed with read. Later write the block from buffer to mem
  - New read miss: check write-buffer or stall read until wr-buffer empty

# Write buffers

- Write buffer: a small buffer
  - Stores put address/value to write buffer, keep going
  - Write buffer writes stores of D$ in the background
  - Loads must search write buffer (in addition to D$)
  - +ve Eliminates stalls on write misses (mostly found in) Write Back Buffers
  - –ve Creates some problems (later)

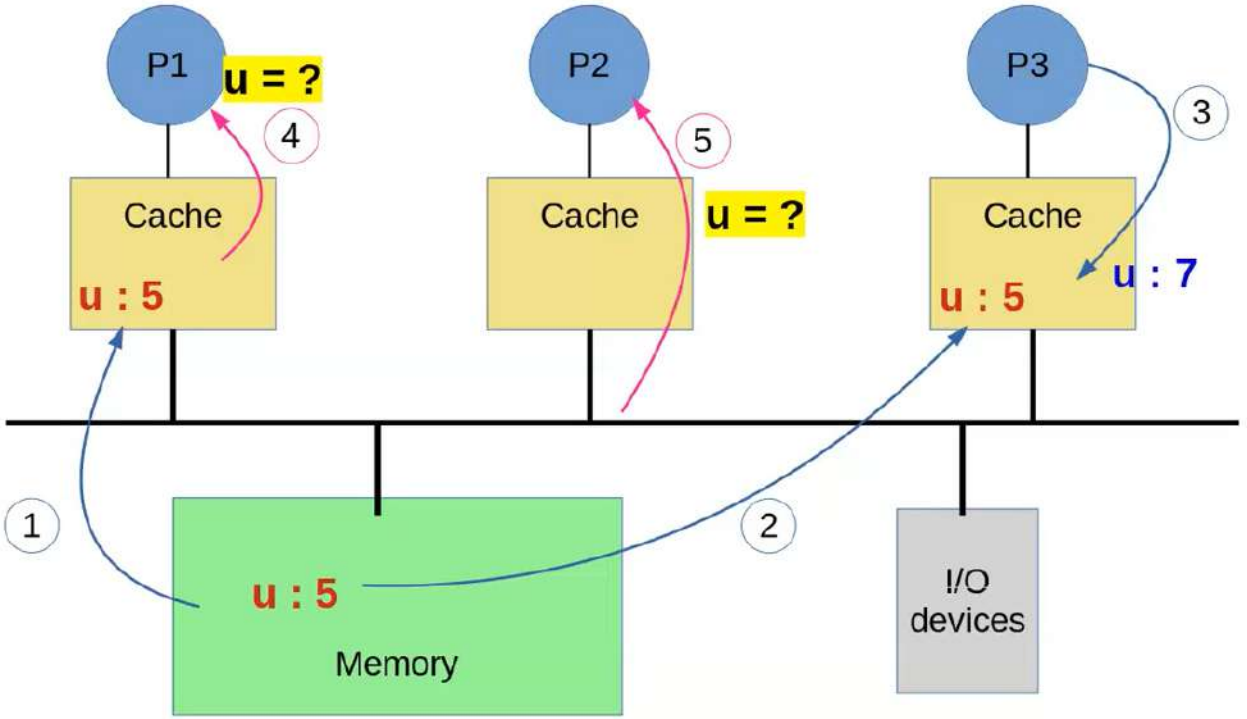


# Cache Coherence Problem

# Cache Coherence

- Intuitive model of what a memory should do?
  - It should provide a set of locations that holds values
  - When a location is read it should return the latest value written to that location
  - Our shared address space based inter process communication also depends on this
- Private cache with processor creates problem
  - Copies of variable may be present in multiple caches
  - A write by one processor, may not become visible to others, in that others keep accessing stale data
- This is the cache coherence problem
- What to do about it?
  - Organise memory hierarchy to make it go away -or-
  - Detect and take actions to eliminate the problem

# Ex: cache coherence problem



# Cache coherence

- Processors see different values of 'u' after event-3
- If write-through cache then
  - <P3, Mem> updated ;
  - <P1> old value ; <P2> may get new value
- If write-back cache then
  - memory also has old value
  - <P1, P2, Mem> = old value
- In case write back of value happens after P1, P2 read value from memory
  - They get the old value
- In case two processor perform write back, then the final value that will reach the memory will depend on the order in which the processors replace the block containing u
- Unacceptable for programming, but this is frequent !

# Cache coherence

- How to solve the coherence problem?
  - We don't want to disallow caching
  - We don't want to invoke OS at each reference to shared data
- Cache coherence has to be addressed as a basic hardware design issue
  - Can be done by eliminating other copies of the data when one copy is getting modified
  - Or by updating the other copies with new value
- Each read should return the latest value. How do we define latest value in a parallel system?

# How to define latest value?

- Two processors might write to same location at same instant
- Or one processor might read so soon after the write by another processor that there is no time to propagate invalidation or update message
- Even in sequential program, “last” is not chronological or physical, it refers to last in the program order
- In parallel case we need to make sense of the collection of program orders

# Concepts of program order and serialisation in sequential and parallel case

- We will discuss the concepts in
  - A uniprocessor - sequential
  - In multiprocessor setup - parallel

# Prog-order + serial...uniprocessor

- Memory operations = read/write within an instruction are assumed to execute atomically with respect to each other in a specified order
- Memory operation issues when it leaves the processor
  - But goes through the cache, write-buffer, memory-modules
  - i.e. it takes long time to complete
- **Only way processor observes state of memory is by issuing memory operations, e.g. Reads**
- Ex: Processor knows that write operation is performed if subsequent read returns the written value or value of a later write
- Ex: Processor completes a read operation means that subsequent writes to that location cannot affect the read value
- "**Subsequent**" is well defined in sequential process => i.e. they are in program order

# Prog-order + serial...parallel case

- Memory operations are same as sequential
- "subsequent" means something more here
- As we do not have one program order
- Rather several program orders interacting with the memory system
- To sharpen our idea of coherence
  - ASSUME: multi-cores, single memory, no caches
- All reads/writes will be directly performed by memory for all processors
- The memory imposes a serial order on accesses
- Also, the read/write acesses of individual process should be in program-order within this overall serial order

# Parallel case ...

- Thus memory system is a point in which hardware determines the order of access
- There is not fixed sequence of access among the processes
- There is arbitrary interleaving
- Fairness -> as each process will eventually access memory
- Our understanding of "last" or "subsequent" access will be defined in this hypothetical serial order
- As the serial order must be consistent, the processes see the writes to a location in the same order