ECE 5730

Memory Systems

Spring 2009

Non-Blocking Caches
Cache Content Management

Cornell University

Announcements

- Quiz on Tuesday

- Quiz 1
  - Average = 8.6
  - Scores are in the Gradebook on Blackboard
  - Remind me to hand them back at the end of class!

see lecture 3

Non-Blocking Caches

- A *blocking cache* stalls the pipeline on a cache miss

- A *non-blocking cache* permits additional cache accesses on a miss
  - Proposed by [Kroft81]; refined by [Farkas94], [Farkas97]
  - *Hit-under-miss*: the next miss causes a stall
  - *Miss-under-miss*: multiple misses may be present up to a limit (and then a stall)

- *Memory-level parallelism*: A miss-under-miss cache coupled with a parallel lower-level memory system see lecture 3

Miss-Under-Miss Cache Design

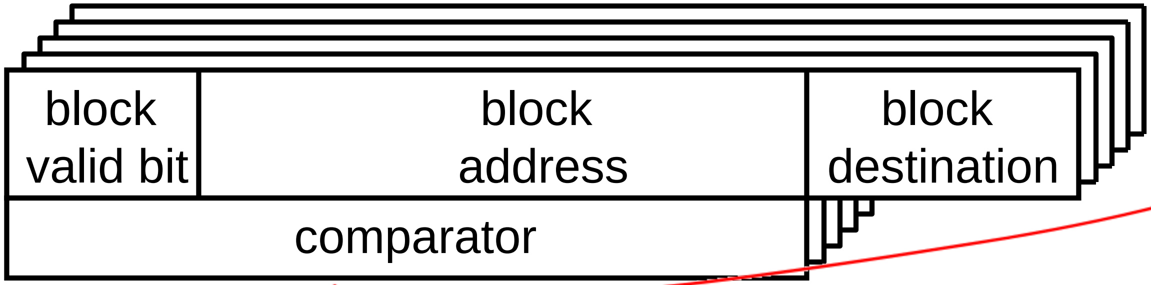
- A *primary miss* occurs when there are no outstanding misses to the same cache line
- Otherwise, a *secondary miss* occurs and is merged with the primary miss (not issued to the next level of the memory hierarchy)
- Hardware required for a miss-under-miss design with an out-of-order processor
  - Miss Status Holding Registers (MSHRs)
  - Address Stack

keeps track of parallel fetching

MSHRs

- When a miss occurs, the MSHRs are looked up to determine if the cache block is already being fetched

- Each MSHR holds information for a primary miss
  - **Block (line) address**
  - **Destination: whether block corresponds to L1 Icache, L1 Dcache, or neither (uncached)**



- If an MSHR hit, then a secondary miss has occurred (primary miss to the same line is outstanding)

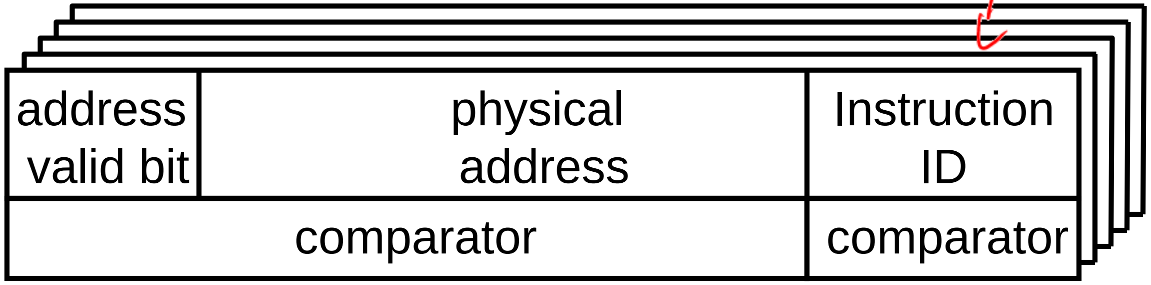
multiple in-flight fetches, don't know order of return of data

{ you need to know what address come back from lower mem-hierarchy levels to make sure you associate the data w/ the right entry allows us to replay instruction selectively (mispredicted branch, exception)

Address Stack

- Resolves primary and secondary data cache misses when a cache block is returned
- Each address stack entry holds information for a primary or secondary miss
  - Physical address
  - Instruction ID (e.g., the PC)

points to the instruction to be replayed



address valid bit

physical address

Instruction ID

comparator

comparator

Address Stack

- If the valid ~~address~~ bit is set and the physical address matches that of the returned block, the instruction is replayed in the pipeline

say we miss on a load. we get the data back, then we replay the load in the pipeline

- If an instruction in the address stack follows a mispredicted branch or exception, that entry's valid bit is reset
  - Instruction will not be found and thus not replayed

load ← primary miss, allocate MSHR

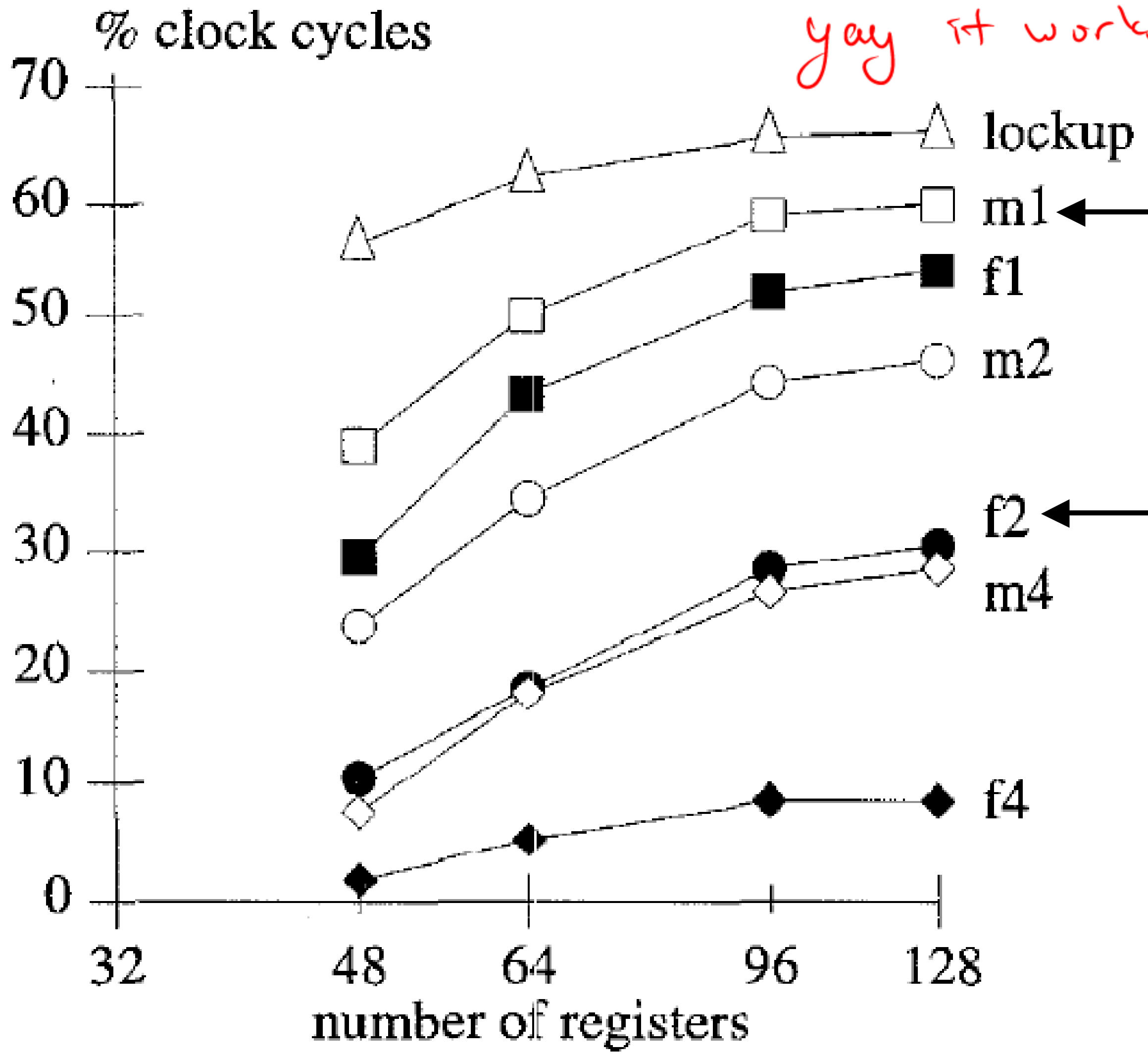
branch ← mispredicted NOT taken, actually taken

load ← secondary miss, should NOT be replayed

on this mispredicted branch, we reset the valid bits on all instruction ID's after the branch, so as to not replay the second load.

Miss-Under-Miss Effectiveness

yay it works!



at most 1 miss allowed

at most 2 *primary* misses allowed; unlimited *secondary* misses

mx: x miss only (primary or secondary)

fx: x primary misses unlimited secondary misses

x MSHR's

∞ address stack entries

[Farkas97] (a) Percentage of (simulated) clock cycles in which a load-miss induced stall was in effect

(this stuff is not strictly pigeonholed, categorization is fuzzy)

Cache Content Management

- Partitioning heuristics
  - ~~Which items get placed store where (cache lev~~el, ~~cache sets/ways, buffers~~)

poorly worded, disregard

what items do I put where?

L1 vs L2, buffers, which set, etc

- Fetching heuristics
  - When to bring an item into the cache
- Locality optimizations
  - Change layout or ordering to improve reuse

Decisions can be made at runtime (*on-line heuristics*),
at design/compile time (*off-line heuristics*),
or a combination of the two (*combined approaches*)

On-line Partitioning Heuristics

- **Replacement policies** LRU, FIFO, etc

  MRU

- **Write policies** Write back vs Write through

  simplistic, there are more complicated schemes

- **Cache/not-cache strategies**

  → may want to make more intelligent, more complex decisions about whether or not to store an item.

Replacement Policy

- Determines which item is replaced in the cache (e.g., for associative caches)

(aka MRU)

- Static policies not most recently used (don't kill the most recently used)

  less overhead than LRU, similar idea

  least recently used

  - LRU, NMRU, random, etc
  - Replacement policy is fixed at design time (prob in hardware)

- Dynamic policies
  - Policy changes with the workload characteristics
  - Recently proposed for microprocessor caches
  - Example: Set Dueling

lawl

Set Dueling

- Proposed for L2 caches by [Qureshi07]

- Replacement policy is LRU
  - The block marked as LRU is replaced

- But the LRU *position* in which a new block is inserted is *dynamic*

  by definition, the block is the most recently used

  choose

  - *LRU policy*: incoming block is placed into the MRU position (traditional LRU approach)
  - *Bimodal Insertion Policy (BIP)*: most blocks are placed into the *LRU* position → set the new block as the least recently used
    - These are replaced unless referenced again right away
  - Either LRU or BIP is chosen based on which one is doing better

it's not so much the LRU/MRU "position" as the state of the block. you set the appropriate bits to set LRU vs MRU

(why BIP?)

Why Placement into the LRU Position?

- Traditional LRU algorithm works well for programs with good temporal locality
- For program with a working set larger than the cache, LRU causes thrashing
  - Blocks continually replace each other

oh noes!

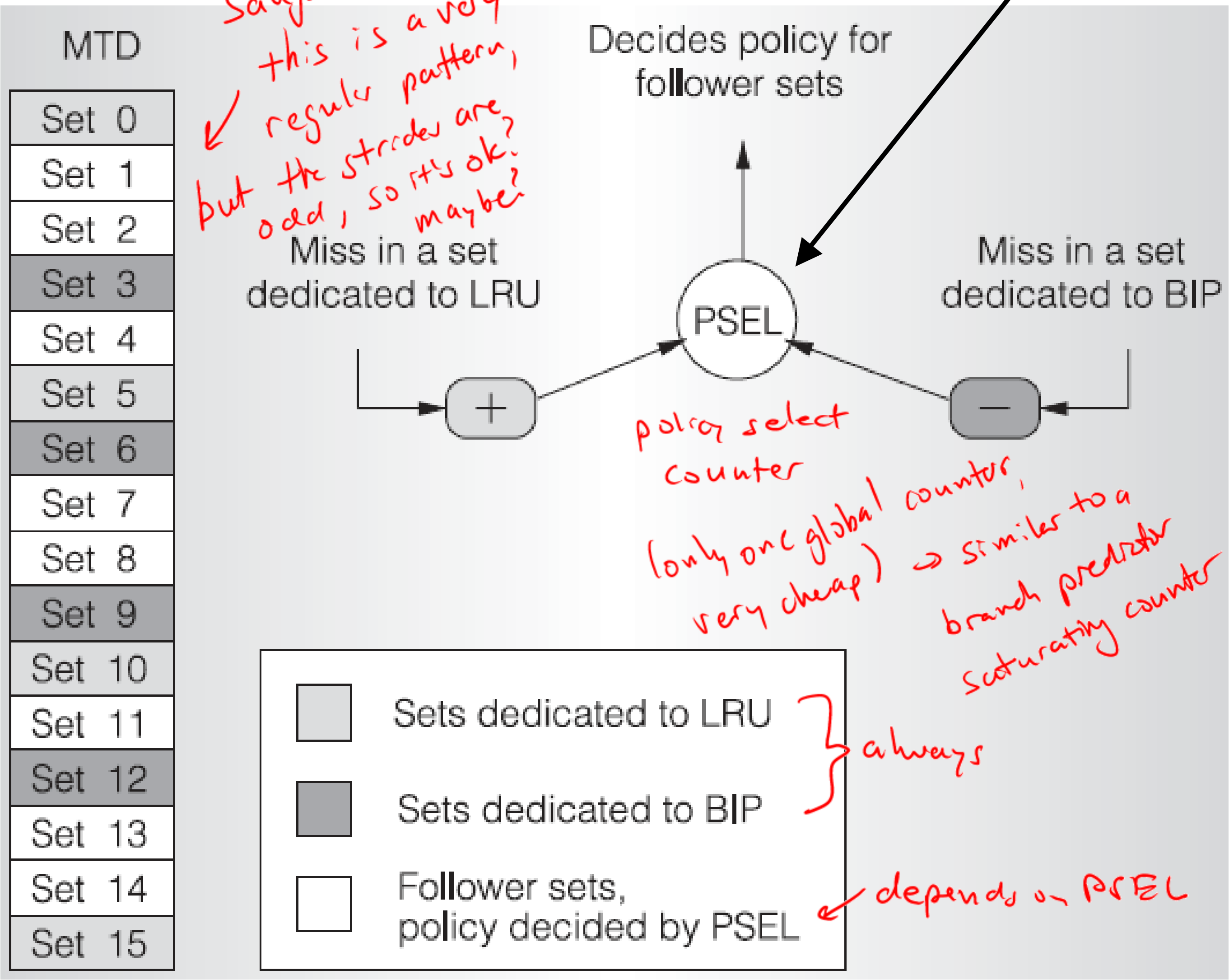
- Can retain some portion of the working set by inserting most blocks into the LRU position
  - These will be quickly replaced
  - Other blocks in the same set will be retained

can get up to 20% improvement with some workloads.

Set Dueling

- **Some sets use LRU, others BIP**
- ***Follower sets* follow the policy that does best**

10 bit saturating counter



Saugata: this is a very regular pattern, but the strides are odd, so it's ok? maybe?

policy select counter

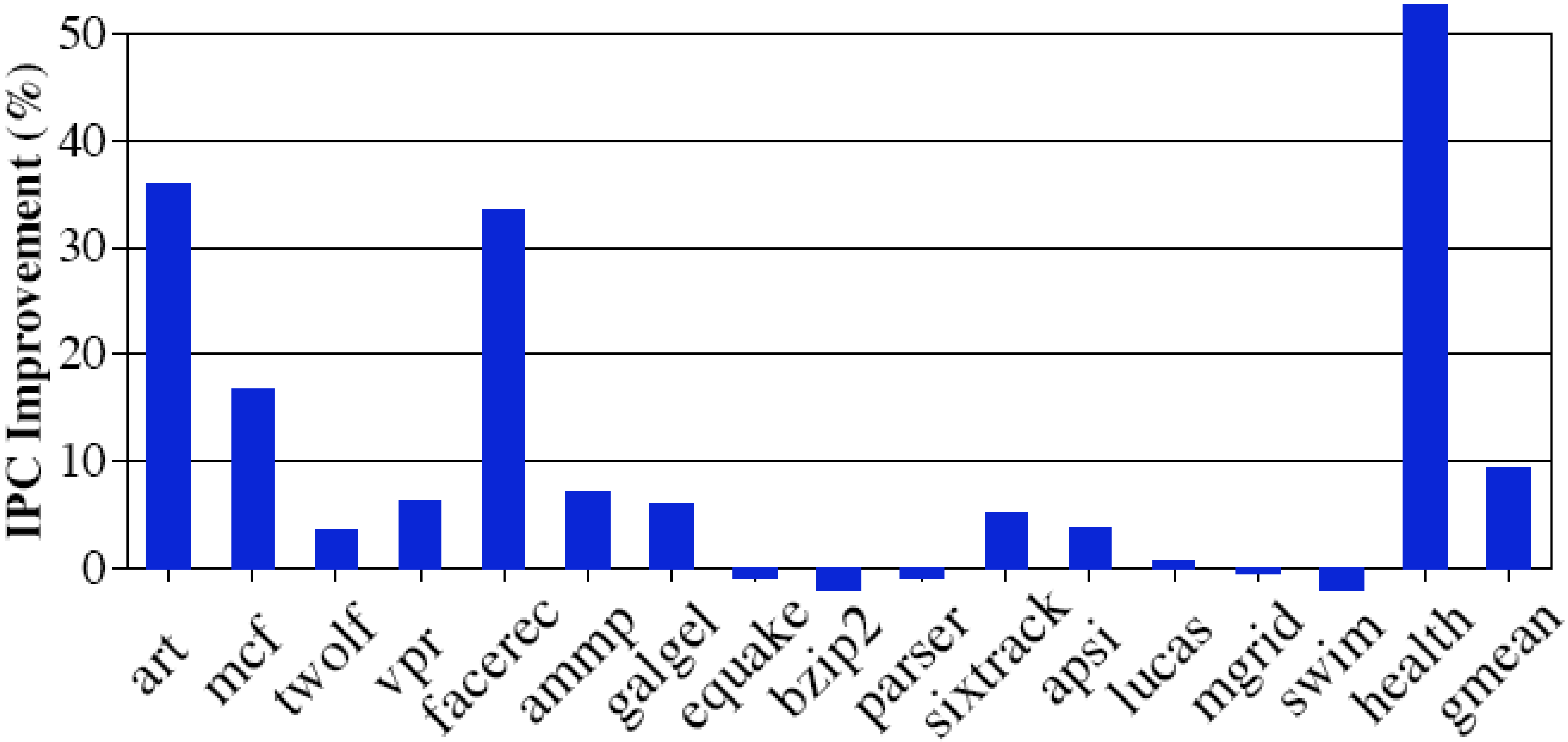
(only one global counter, very cheap) → similar to a branch predictor saturating counter

always

depends on PSEL

[Qureshi08]

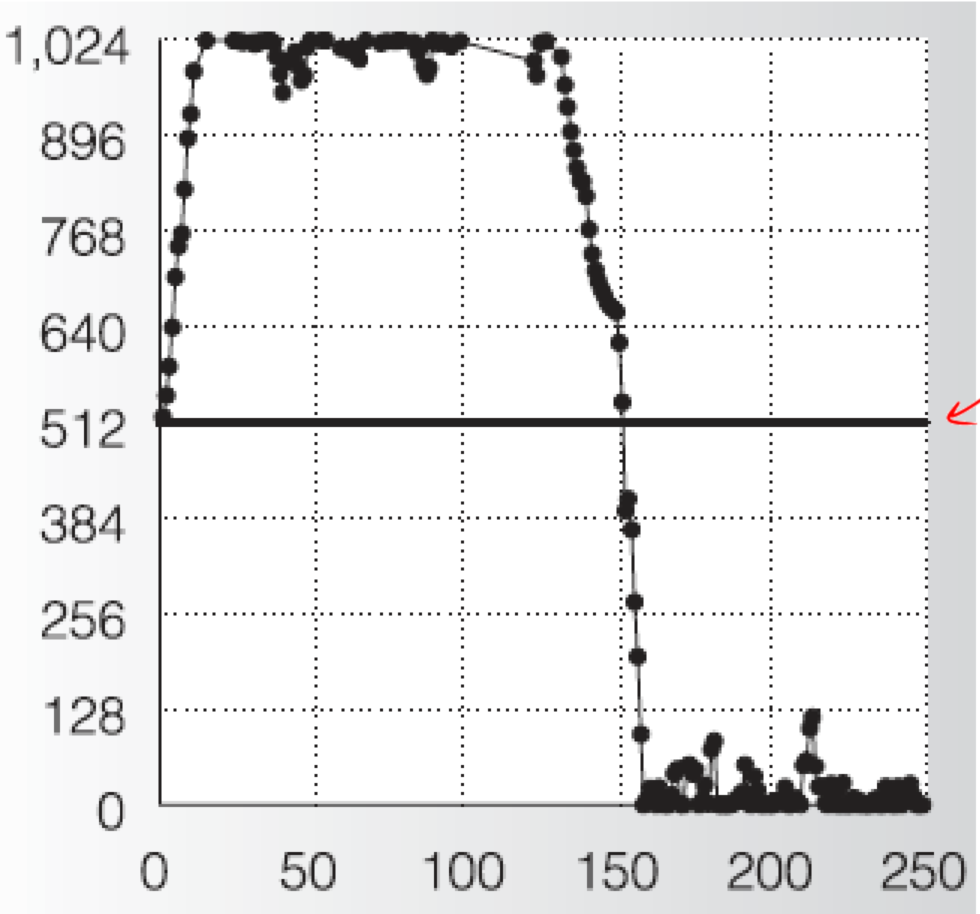
Performance Improvement



[Qureshi07]

Dynamic Behavior of *ammp*

PSEL
Value



BIP

policy switch threshold

LRU

Millions of Instructions

[Qureshi08]

unlike reads, writes are inherently asynchronous and non-blocking

→ just need to put the written data somewhere safe till we can write it back to DRAM

Cache Write Policies

- Cache management decisions involving writes must balance several tradeoffs
  - – The time delay to store the data on a hit or miss
  - – The amount of traffic generated to the next level
    - The future usage of the rest of the line containing the write address and the line that is replaced in the cache

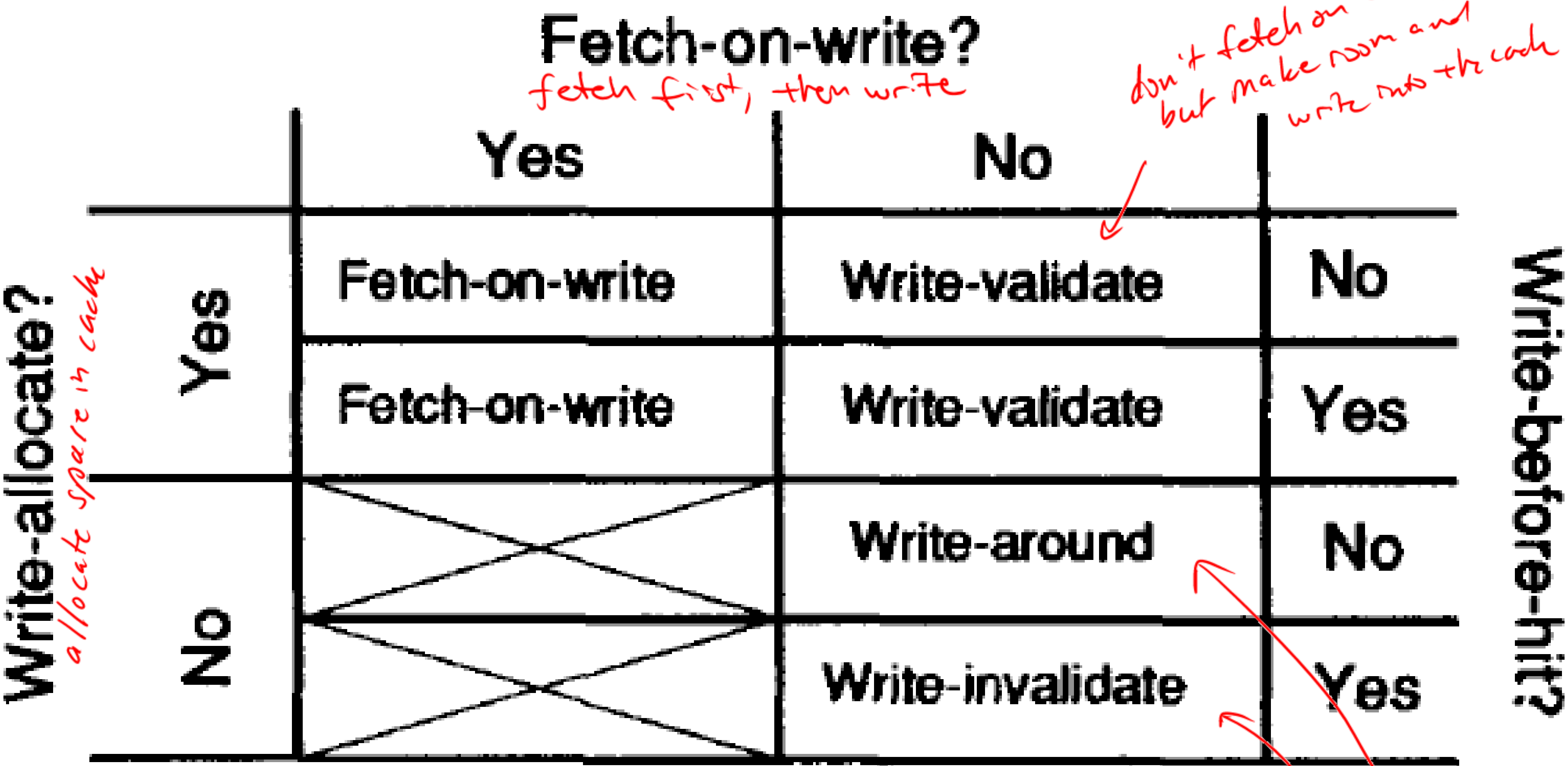
- For write hits, need to decide whether or not to write to the next level of the hierarchy
  - – Writethrough versus writeback

- For write misses, there are a number of options…

Cache Write Miss Options

- **Fetch-on-write?** and put it into the cache, then write?
  - **Do we fetch the block that is being written?**
- **Write-allocate?** → do we write to the cach at all?
  - **Do we allocate space in the cache for the write?**
- **Write-before-hit?** → can we write in parallel with the hit check to save time? assume hit
  - **Do we write the block before checking for a cache hit?**

Write Miss Alternatives



Write-allocate?		Fetch-on-write? Yes	Fetch-on-write? No	Write-before-hit?
Yes		Fetch-on-write	Write-validate	No
		Fetch-on-write	Write-validate	Yes
No		✗	Write-around	No
		✗	Write-invalidate	Yes

fetch first, then write

don't fetch on write, but make room and write into the cache

allocate spare in cache

write to the next lower mem level

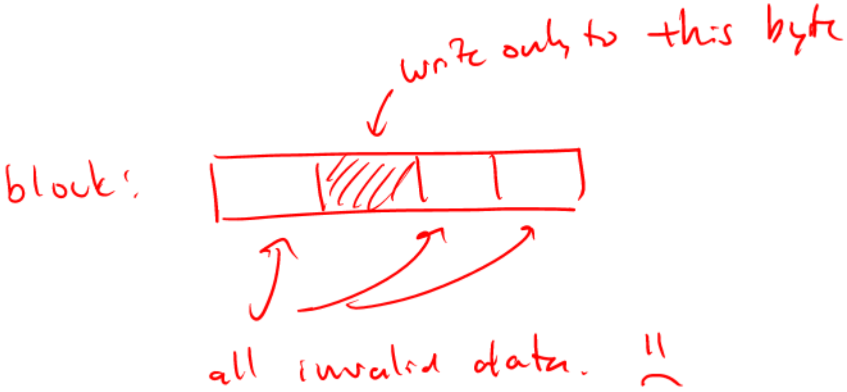
[Jouppi93] as of 2009, he's @ HP, apparently he's super awesome

write anyway, before I know if it's a hit or miss fix it later

Write Miss Alternatives

- Fetch-on-write
  - Line is fetched and data is written into the cache

- Write-validate
  - Line is not fetched and data is written into the cache
  - Valid bits for all but the written data are turned off
  - If used with write-before-hit, entire line is invalidated on a miss



Write Miss Alternatives

- Write-around
  - On a miss, write data bypasses the cache and is written to the next level of the memory hierarchy

- Write-invalidate
  - Data is written into the cache before hit detection
  - If a miss occurred, line is invalidated

additional "miss" due to fetch

Which Approach is Best?

- Fetch-on-write is simple and useful if other words in the fetched block are often accessed

→ spatial locality

- Write-invalidate avoids misses when
  - neither the line containing the data being written,
  - or the old contents of the cache line
  - are read before being replaced

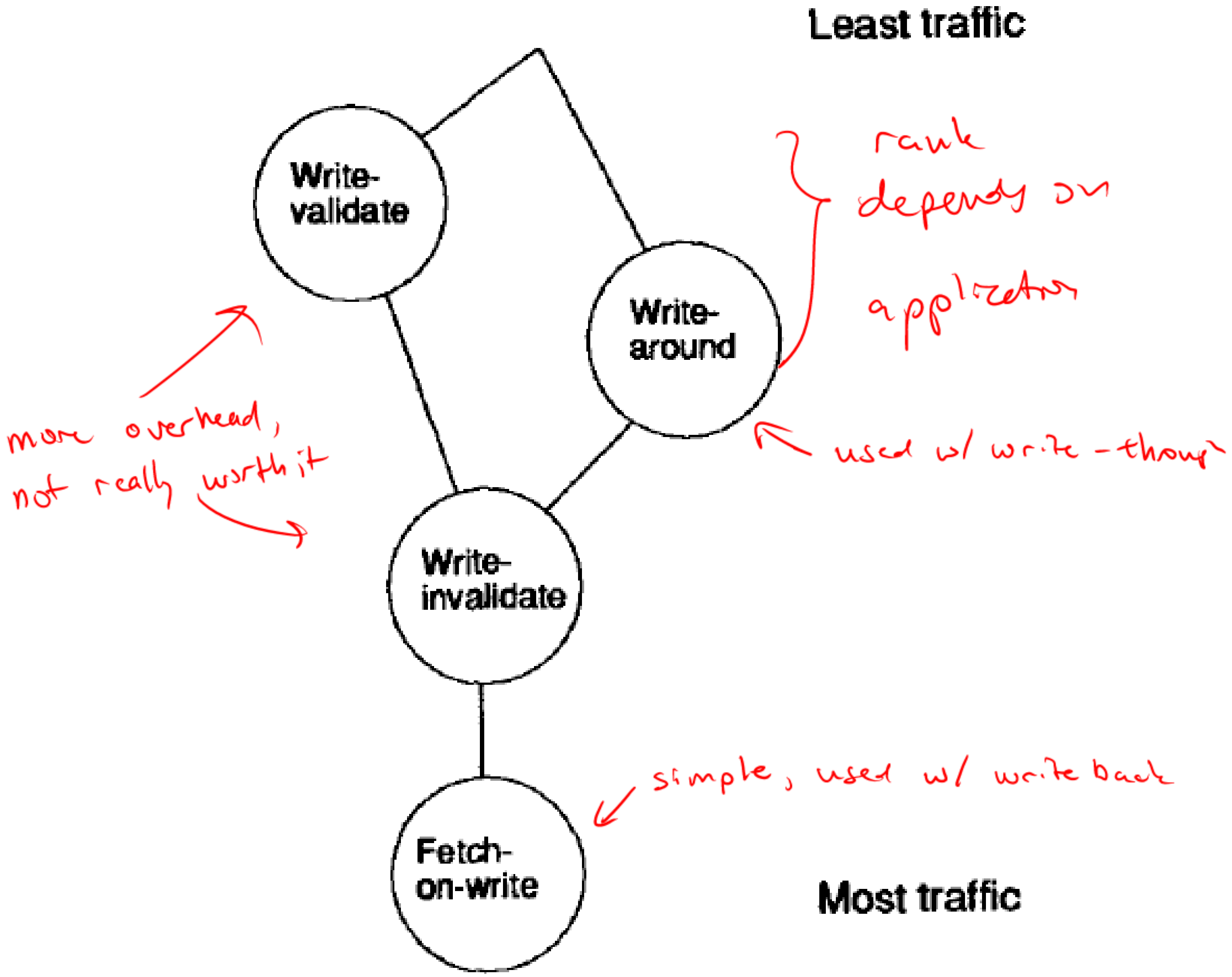
we don't fetch if we're just writing and not reading the block

- Write-around also avoids misses when the old contents of the cache line are read next

- Write-validate also avoids misses when the data that was just written is accessed next

→ writes over what we just wrote

Relative Fetch Traffic



Least traffic

Write-validate

Write-around

Write-invalidate

Fetch-on-write

Most traffic

more overhead, not really worth it

rank depends on application

used w/ write-through

simple, used w/ write back

[Jouppi93]

Writethrough Versus Writeback

- **Advantages of writethrough**
  - **Simpler to maintain L1-L2 coherence** ↙ bigger problem when L2 was off-die
  - **Parity instead of ECC for error tolerance (bigger issue)**
    - **Can generate byte parity on a byte write** ← overhead not terrible
    - **If an error occurs, can force a miss and read from L2**
    - **For writeback, need ECC, requiring a read-modify-write operation on a byte write**, only copy I have could be corrupted

      high overhead (up to 50%)

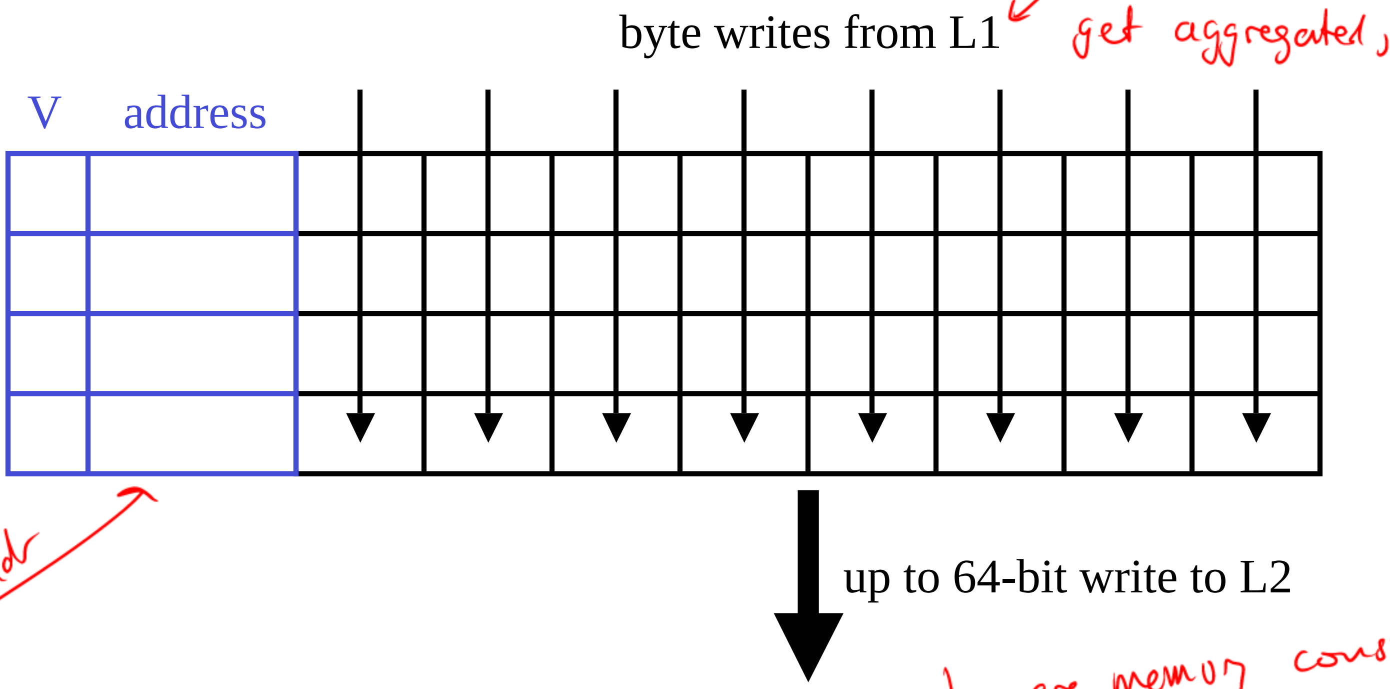
caches have soft errors due to bit flips, fix w/ parity

- **Disadvantage of writethrough** → L1 + L2, 2 copies! ☺
  - **Higher write traffic to L2**

"Merging" Write Buffer

Coalescing Write Buffer

- Hardware buffer between L1 and L2 that can accumulate write bursts to L2 L1 ⟷ CWB ⟷ L2
- Narrow writes are merged into a wider entry
  - Reduces the number of L2 writes



lots of byte writes get aggregated, then written

not super helpful, because all the byte writes have to be from the same cache line

addr

dunno what that is?

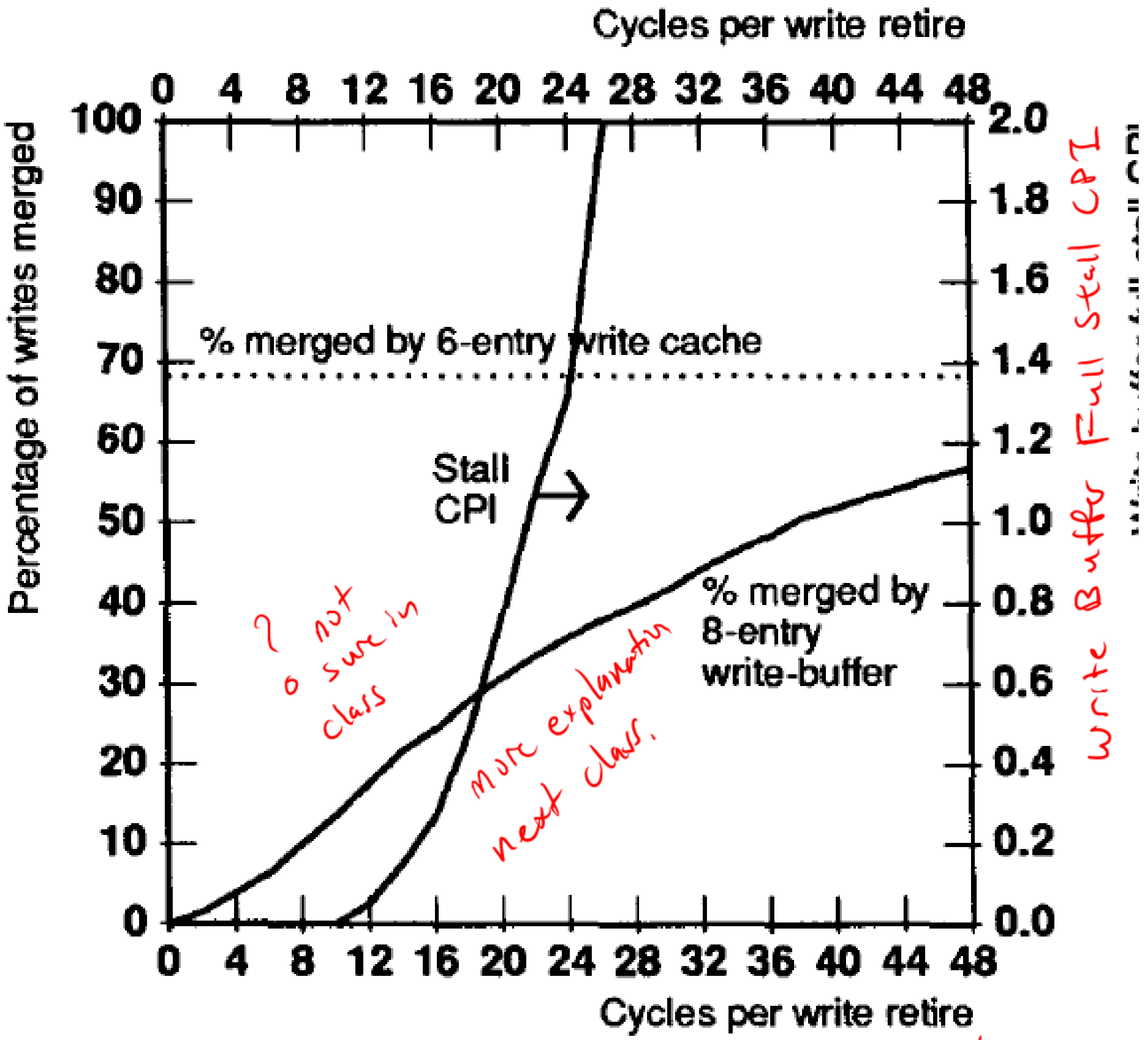
there are memory consistency problems, so there might be a fence operation

When to Empty the Write Buffer?

- Want to retain data to increase the merging of data into a wider L2 write

conflicting constraints

- Want to empty the buffer to prevent it from getting full and stalling the pipeline



? not sure in class

more explanation next class.

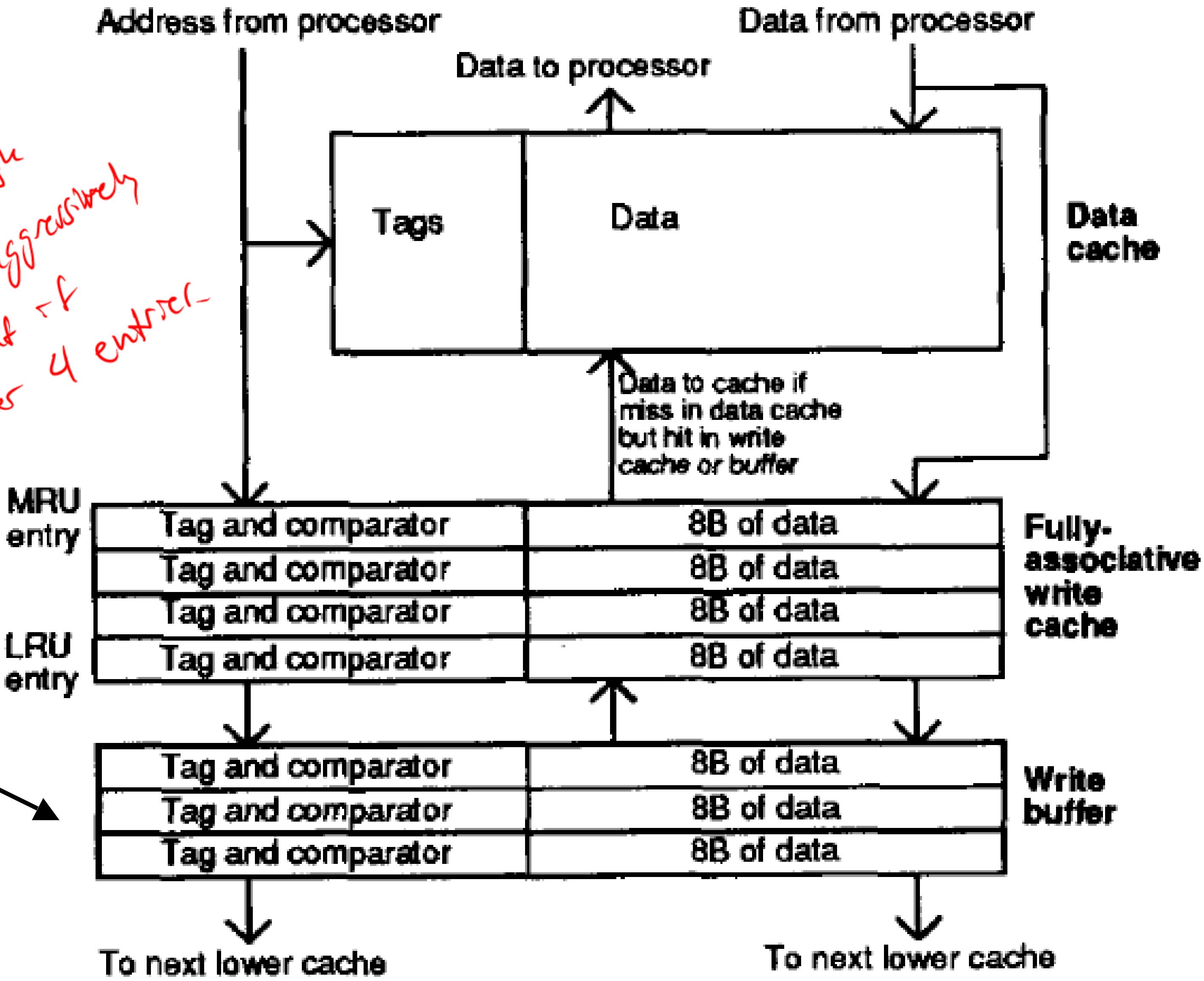
Write Buffer Full Stall CPI

how long it takes to retire the write

[Jouppi93]

Write Cache

basically a single buffer that aggressively empties itself if we get over 4 entries



empty only if need room

empty aggressively

[Jouppi93]

Next Time

Cache Content Management