

# Advanced Computer Architecture

## Chapter 4: Caches and Memory Systems Part 3: Miss penalty reduction

November 2025

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup>, 5<sup>th</sup> and 6<sup>th</sup> eds), and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

# Average memory access time:

$$\text{AMAT} = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}$$

## There are three ways to improve AMAT:

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

We now look at each of these in turn...

# Write policy:

## Write-through vs write-back

- **Write-through: all writes update cache and underlying memory/cache**

- Can always discard cached data - most up-to-date data is in memory
- Cache control bit: only a *valid* bit

- **Write-back: all writes simply update cache**

- Can't just discard cached data - may have to write it back to memory
- Cache control bits: both *valid* and *dirty* bits

- **Other Advantages:**

- **Write-through:**

- memory (or other processors – or just the next level of the cache) always has latest data
- Simpler management of cache

- **Write-back:**

- much lower bandwidth, since data often overwritten multiple times
- Better tolerance to long-latency memory?

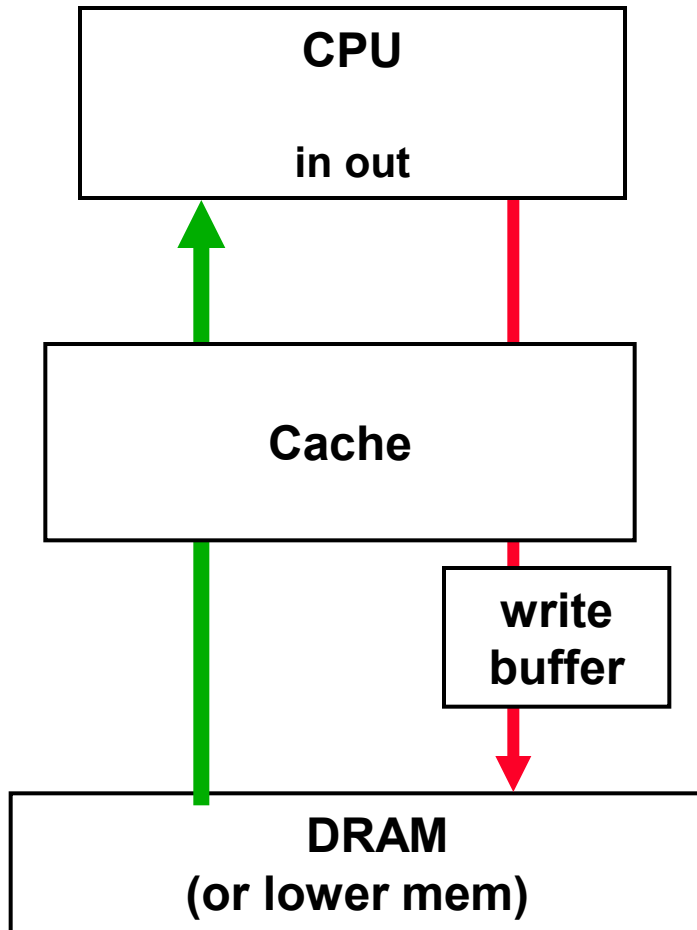
# Write policy 2:

## Write allocate vs non-allocate

### (What happens on write-miss?)

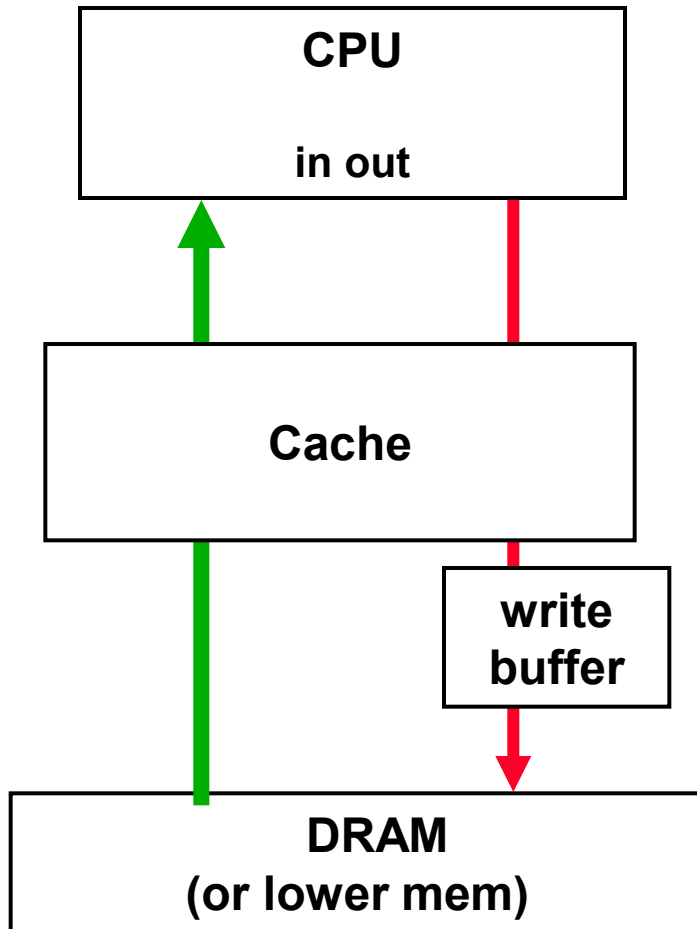
- **Write allocate: allocate new cache line in cache**
  - Usually means that you have to do a “read miss” to fill in rest of the cache-line!
  - Alternative: per/word valid bits
- **Write non-allocate (or “write-around”):**
  - Simply send write data through to underlying memory/cache - don't allocate new cache line!
- Which is right? It depends... maybe get programmer to use a “non-temporal store” instruction

# Reducing Miss Penalty: Read Priority over Write on Miss



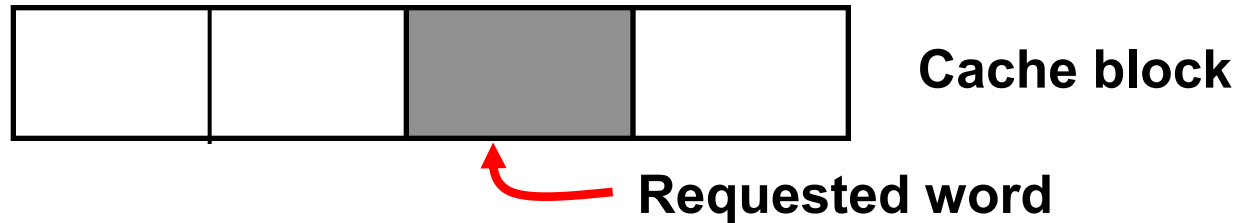
- Consider write-through with write buffers
  - RAW conflicts with main memory reads on cache misses
    - Could simply wait for write buffer to empty, before allowing read
    - Risks serious increase in read miss penalty (old MIPS 1000 by 50% )
    - Solution:
      - Check write buffer contents before read; if no conflicts, let the memory access continue
- If you use write-back, you also need a write buffer to hold *displaced* blocks
  - Read miss replacing dirty block
  - Normal: Write dirty block to memory, and then do the read
  - Instead copy the dirty block to a write buffer, then do the read, and then do the write
  - CPU stall less since restarts as soon as do read

# Write buffer issues



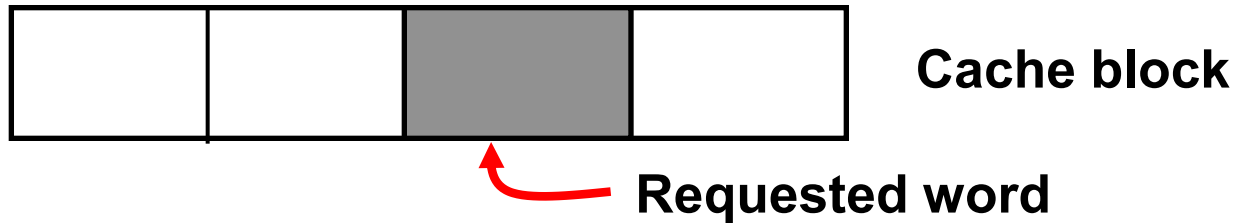
- Size: 2-8 entries are typically sufficient for caches
  - But an entry may store a whole cache line
  - Make sure the write buffer can handle the typical store bursts...
  - Analyze your common programs, consider bandwidth to lower level
- Coalescing write buffers
  - Merge adjacent writes into single entry
  - Especially useful for write-through caches
- Dependency checks
  - Comparators that check load address against pending stores
  - If match there is a dependency so load must stall
- Optimization: load forwarding
  - If match and store has its data, forward data to load...
- Integrate with victim cache?

# Reduce miss penalty: early restart and critical word first

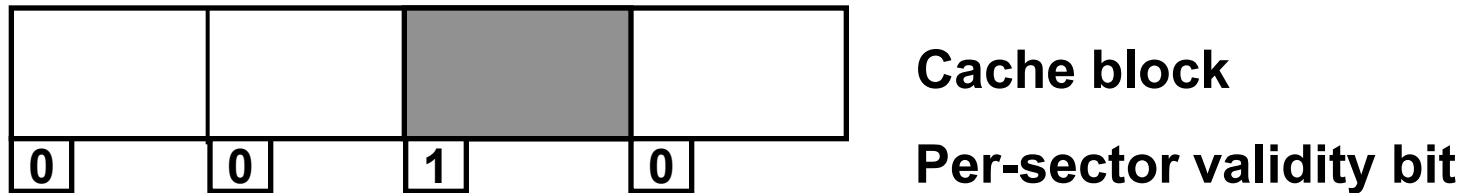


- The processor can continue as soon as the requested word arrives
- Don't wait for full block to be loaded before restarting CPU
  - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First—Request the missed word *first* from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block.
- Generally useful only in large blocks,
  - (Access to contiguous sequential words is very common – perhaps a simple scheme will work pretty well most of the time?)

# Early restart and critical word first and sectored cache lines



- Some care is needed: what if processor issues another load to another word in the cache line, before it arrives?



- Divide cache line into “sectors” – each with its own validity bit (maybe “dirty” bits too)
- We allocate in units of cache lines, but we deliver data in units of sectors
- We can fetch the sectors in any order, perhaps even leaving them invalid until requested
- Eg IBM Power9: 128B lines, 32B sectors (<https://en.wikichip.org/wiki/ibm/microarchitectures/power9>)



# Reduce miss penalty: non-blocking caches to reduce stalls on misses

- Non-blocking cache or lockup-free cache allows data cache to continue to supply cache hits during a miss
  - requires full/empty bits on registers or out-of-order execution
  - requires multi-bank memories
- “hit under miss” reduces the effective miss penalty by working during miss instead of ignoring CPU requests
- “hit under multiple miss” or “miss under miss” may further lower the effective miss penalty by overlapping multiple misses
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  - Requires multiple memory banks (otherwise cannot support)
  - Eg IBM Power5 allows 8 outstanding cache line misses

## Compare:

prefetching: overlap memory access with pre-miss instructions,

Non-blocking cache: overlap memory access with post-miss instructions

# What happens on a Cache miss?

- For in-order pipeline, two options:

- Freeze pipeline in Mem stage (popular early on: Sparc, R4000)

IF	ID	EX	Mem	stall	stall	stall	...	stall	Mem	Wr
	IF	ID	EX	stall	stall	stall	...	stall	stall	Ex Wr

- Use Full/Empty bits in registers + MSHR queue

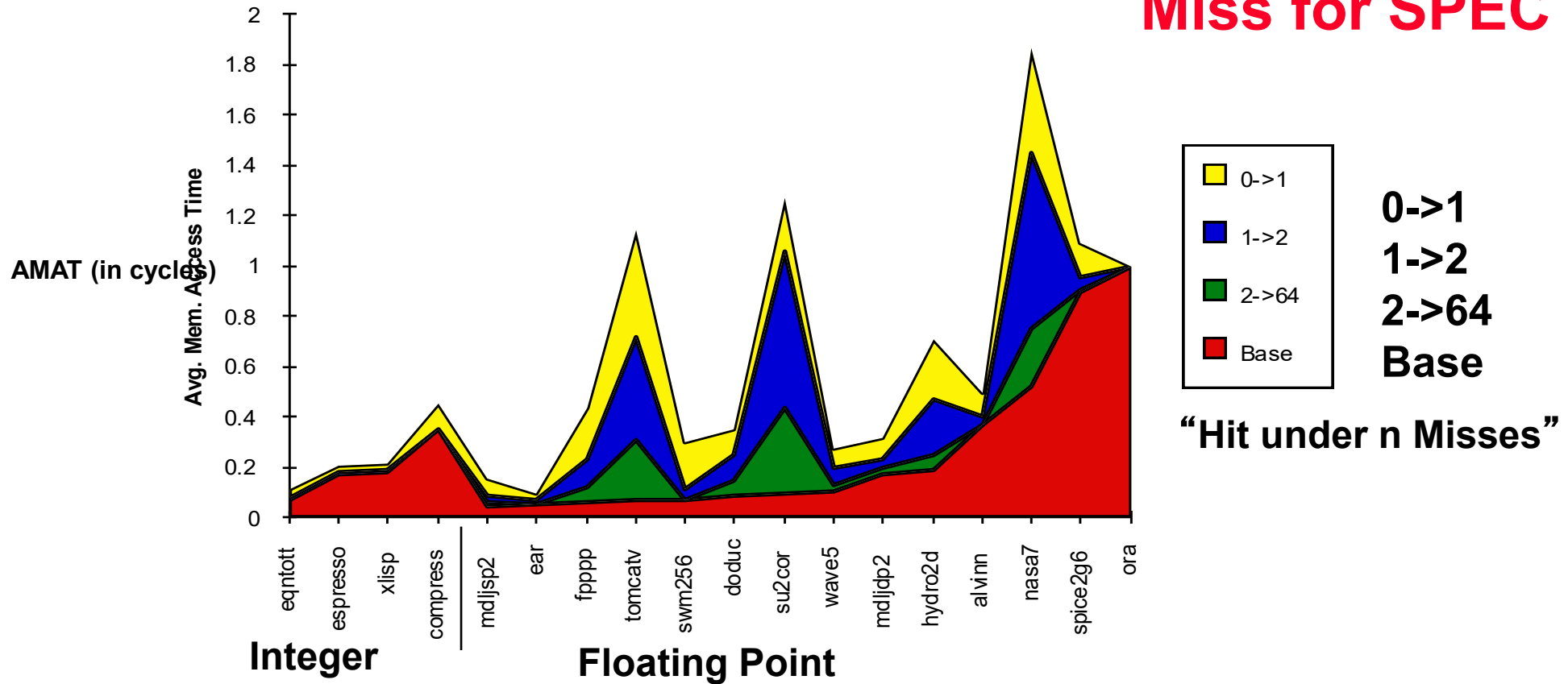
- MSHR = “Miss Status/Handler Registers” (Kroft\*)

Each entry in this queue keeps track of status of outstanding memory requests to one complete memory line.

- Per cache-line: keep info about memory address.
- For each word: register (if any) that is waiting for result.
- Used to “merge” multiple requests to one memory line
- New load creates MSHR entry and sets destination register to “Empty”. Load is “released” from pipeline.
- Attempt to use register before result returns causes instruction to block in decode stage.
- Limited “out-of-order” execution with respect to loads.

Popular with in-order superscalar architectures.

- Out-of-order pipelines already have this functionality built in... (load queues, etc). Cf also Power6 “load lookahead mode”



- FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26

- Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19

- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss

- Hit-under-miss implies loads may be serviced out-of-order...**

- Need a memory “fence” or “barrier” (<http://www.linuxjournal.com/article/8212>)

- PowerPC eieio (Enforce In-order Execution of Input/Output) Instruction**

# Add a second-level cache

## ● L2 Equations

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

$$\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$$

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})$$

## ● Definitions:

- **Local miss rate**—misses in this cache divided by the total number of memory accesses **to this cache** ( $\text{Miss rate}_{L2}$ )
- **Global miss rate**—misses in this cache divided by the total number of memory accesses **generated by the CPU** ( $\text{Miss Rate}_{L1} \times \text{Miss Rate}_{L2}$ )
- Global Miss Rate is what matters

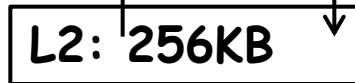
# Multiple levels of cache - example



**L1: 32KB, 8-way associative I and D**

**L1D: writeback, two 256-bit loads and a 256-bit store every cycle**

64B/cycle 32B/cycle



**L2: 256KB, 8-way writeback with ECC. Can provide a full 64B line to the data or instruction cache every cycle, 11 cycle minimum latency and 16 outstanding misses.**

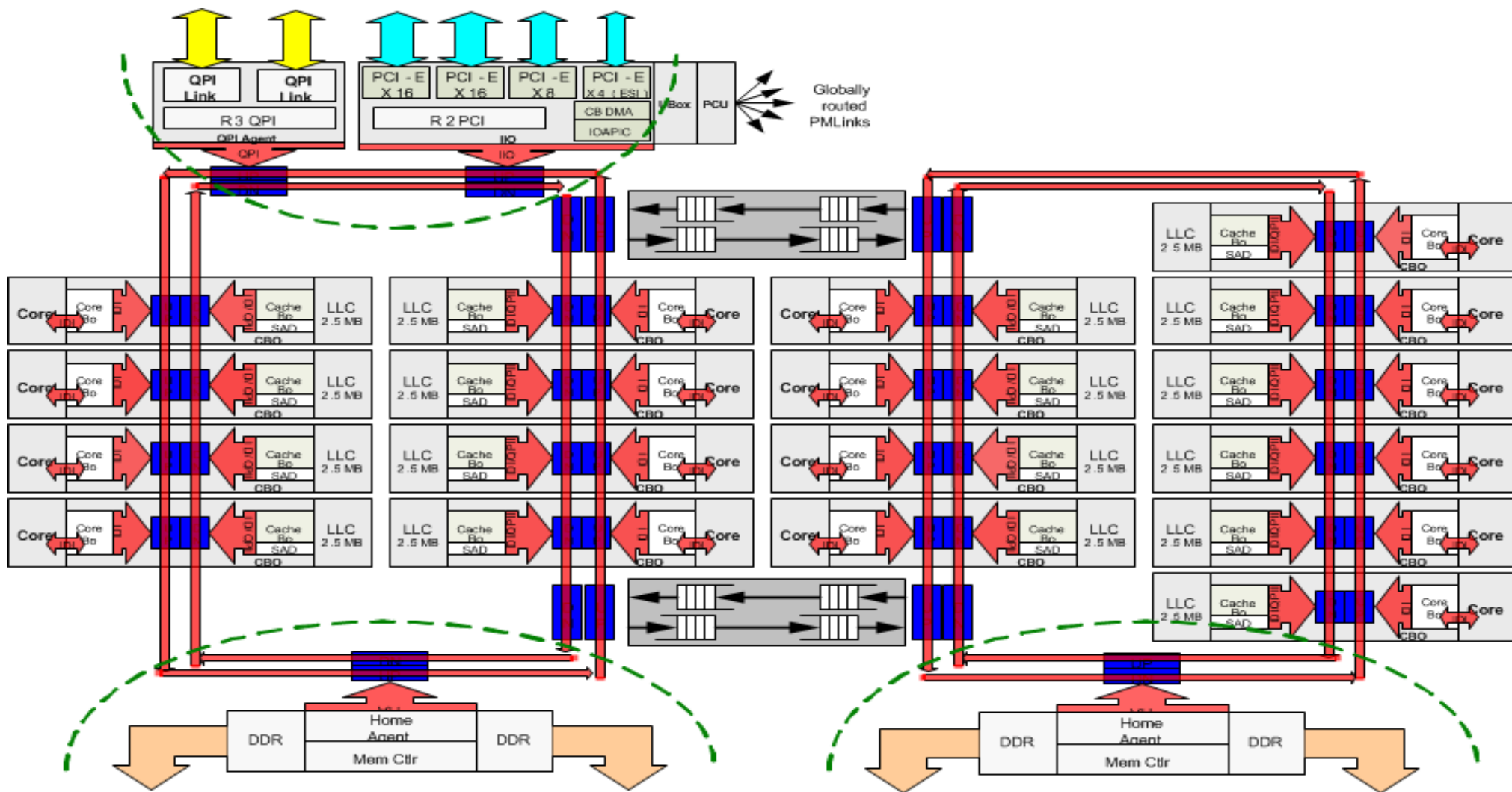
32B/cycle



**L3: Size varies from device to device. Shared by all cores on chip. Connected by ring interconnect (actually two connected rings)**



- Example: Intel Haswell e5 2600 v3
- 18 cores, 145W TDP, 5.56B transistors



- Example: Intel Haswell e5 2600 v3
- Q: do all LLC hits have same latency?
- Q: do all LLC misses have same latency?

## ● Multi-level inclusion

- L2 cache contains everything in L1
- $L_{n+1}$  cache contains everything in  $L_n$

## Multilevel inclusion

- *We might allocate into L1 but not into L2*
- *We might allocate into L2 but not into L1*
- *We might allocate into L1 and L2 but not LLC*

- L3 (Last-level cache) is sometimes managed as a victim cache – data is allocated into LLC when displaced from L2 (eg AMD Barcelona, Apple A9)
- Example: Intel's Crystalwell processor has a 128MB DRAM L4 cache on a separate chip in the same package as the CPU, managed as a victim cache

## ● Issues:

- replacement of dirty lines?
- Cache coherency - invalidation
  - With MLI, if line is not in L2, we don't need to invalidate it in L1

# Summary

We can reduce the miss penalty.....

- By choosing write back instead of write-through
  - (because reducing traffic to the next level of the memory system may mean you don't stall later)
- Using a write buffer
  - On a load, check in the write buffer in parallel with cache access
- By choosing between write-allocate and write-no-allocate wisely
- Early restart and critical-word first
- Avoid stalling on misses: non-blocking cache, hit-under-miss
- Add a second cache
- Add a third, fourth cache
  - Multi-level inclusion? Why does it matter?
- Look in your neighbour's cache