CMPEN 431 Computer Architecture Fall 2018

Caching: Exploiting the Memory Hierarchy

Jack Sampson(www.cse.psu.edu/~sampson)

Topics

- Improving cache performance
- Virtual memory hardware support (TLBs)

The Memory Hierarchy

- What properties would an ideal memory system have?
  - Uniform random access to every named element
  - Very large (as close to infinite as you can afford)
  - Very fast (as close to processing speed as you can get)
  - Very cheap
- Dynamic tensions
  - Physics: Small is fast, large is slow (speed of light)
  - Technology: Cheap, dense memory (DRAM) is also slow memory
    Fast memory (SRAM) is neither cheap nor dense
  - Anthropology: Actual accesses are neither uniform, nor random
- Insight: make the common case (memory access) fast
  - Spatial locality (non-uniformity of access in the address space)
  - Temporal locality (non-uniformity of access in temporal sequence)

The Memory Hierarchy, cont.

- Use caching to **emulate** an ideal memory
  - Provide fast access to high-locality, common case data accesses
  - Back each level with larger, cheaper technologies
  - Spend efforts proportional to “capturable” locality exploitation

- Will caching always work?
  - Short answer – no.
  - Longer answer – it will tend to work on the sorts of structures (both code and data) that people tend to use for many types of computations, and many other types of computations can be **restructured** to be more cache friendly
  - Caching works on enough parts of enough applications that we tend to regard pathologically non-cache-friendly applications as special cases
  - How ***well*** caching works, on the other hand…

Measuring Cache Performance

- Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{CC}$$

$$= \text{IC} \times \underbrace{(\text{CPI}_{\text{ideal}} + \text{Memory-stall cycles})}_{\text{CPI}_{\text{with-stalls}}} \times \text{CC}$$

- Memory-stall cycles come from cache misses (a sum of read-stalls and write-stalls)

$$\text{Read-stall cycles} = \text{reads/program} \times \text{read miss rate} \times \text{read miss penalty}$$

$$\text{Write-stall cycles} = (\text{writes/program} \times \text{write miss rate} \times \text{write miss penalty}) + \text{write buffer stalls}$$

- For write-through caches, we can simplify this to

$$\text{Memory-stall cycles} = \text{accesses/program} \times \text{miss rate} \times \text{miss penalty}$$

Impacts of Cache Performance

- Relative cache penalty increases as processor performance improves (faster clock rate and/or lower CPI)
  - The memory speed is unlikely to improve as fast as processor cycle time. When calculating CPI_{stall}, the cache miss penalty is measured in *processor* clock cycles needed to handle a miss
  - The lower the CPI_{ideal}, the more pronounced the impact of stalls

- A processor with a CPI_{ideal} of 2, a 100 cycle miss penalty, 36% load/store instr's, and 2% I$ and 4% D$ miss rates

Memory-stall cycles =
(100% (I-fetch/Inst) * 2% miss/I-fetch × 100 cycles/miss) +
(36% (Ld-St/Inst) × 4% (miss/Ld-St) × 100 cycles/miss) = 3.44 Cycles/Inst

So $\text{CPI}_{with\text{-}stalls}$ = 2 + 3.44 = **5.44 Cycles/Inst**

more than twice the CPI_{ideal} !

Impacts of Cache Performance, Con't

- Relative cache penalty increases as processor performance improves (lower CPI)
- What if the CPI_{ideal} is reduced to 1? 0.5? 0.25?

CPI_{stall} = 4.44 (up from 3.44/5.44 = 63% to 3.44/4.44 = 77%)

- What if the D$ miss rate went up 1%? 2%?

CPI_{stall} = 2 + (2% x 100 + 36% x 5% x 100) = 5.80

- What if the processor clock rate is doubled (doubling the miss penalty)?

CPI_{stall} = 2 + (2% x 200 + 36% x 4% x 200) = 8.88 !!

(still faster than the 1x clock @ 5.44 CPI, but a lot less than 2x faster…)

Average Memory Access Time (AMAT)

- A larger cache will have a longer access time. An increase in hit time will likely add another stage to the pipeline. At some point the increase in hit time for a larger cache will overcome the improvement in hit rate leading to a decrease in performance.
- Average Memory Access Time (AMAT) is the average to access memory considering both hits and misses

AMAT = Time for a hit + Miss rate x Miss penalty

- What is the AMAT for a core with a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per access and a cache access time of 1 clock cycle?

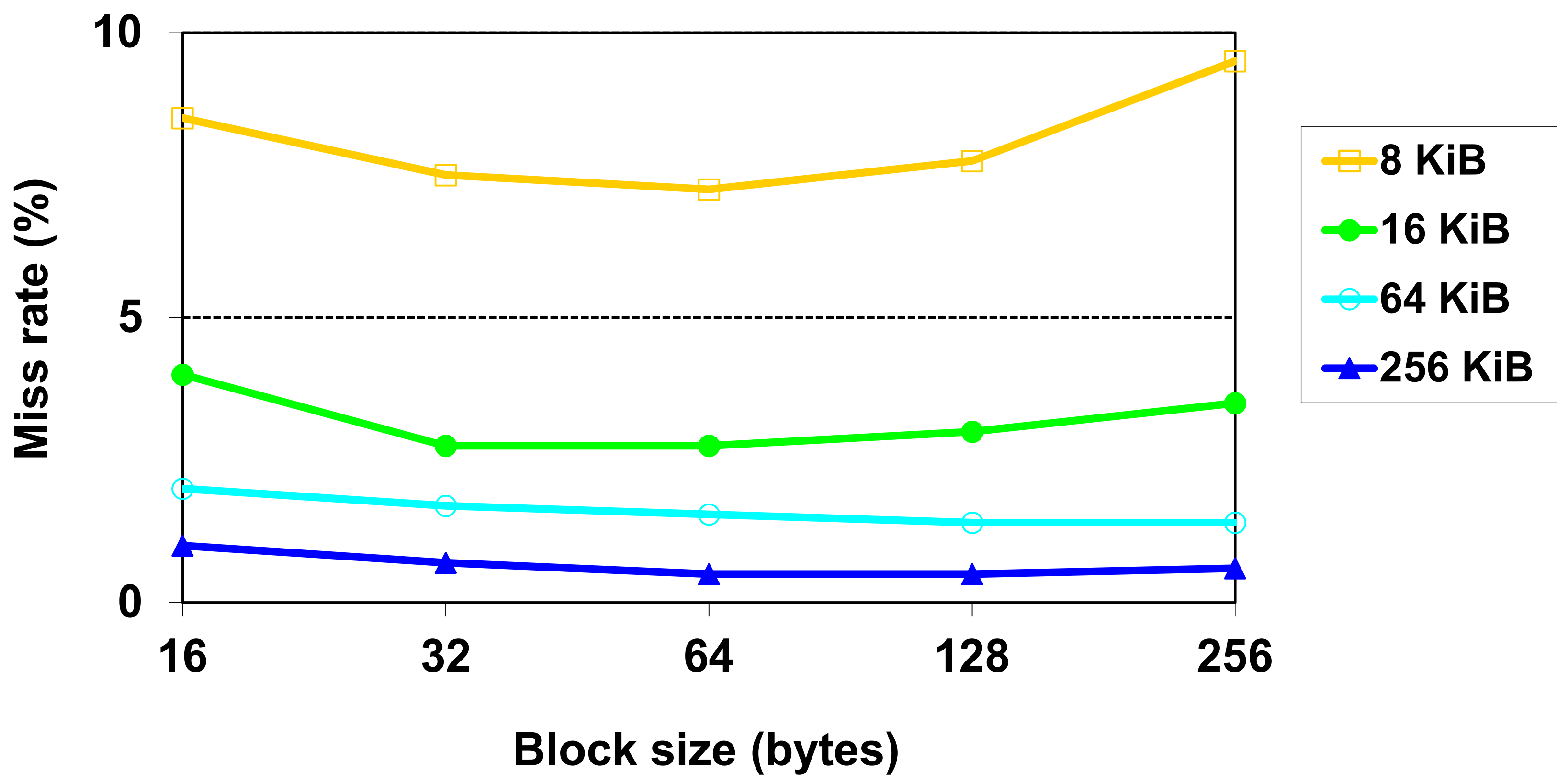
AMAT =

1 cycle/access + 0.02 misses/access x 50 cycles/miss = 2 cycles/access

Recall: The 3 Sources of (uniproc.) Cache Misses

- Compulsory (cold start or process migration, first reference):
  - First access to a block, “cold” fact of life, not a whole lot you can do about it. If you are going to run “millions” of instruction, compulsory misses are insignificant
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (stay tuned) (may increase access time)

Miss Rate vs Block Size vs Cache Size



- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

Reducing Cache Miss Rates #1

1. Allow more flexible block placement

- In a direct mapped cache a memory block maps to exactly one cache block
- At the other extreme, could allow a memory block to be mapped to *any* cache block – fully associative cache
- A compromise is to divide the cache into sets each of which consists of n “ways” (n-way set associative). A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)

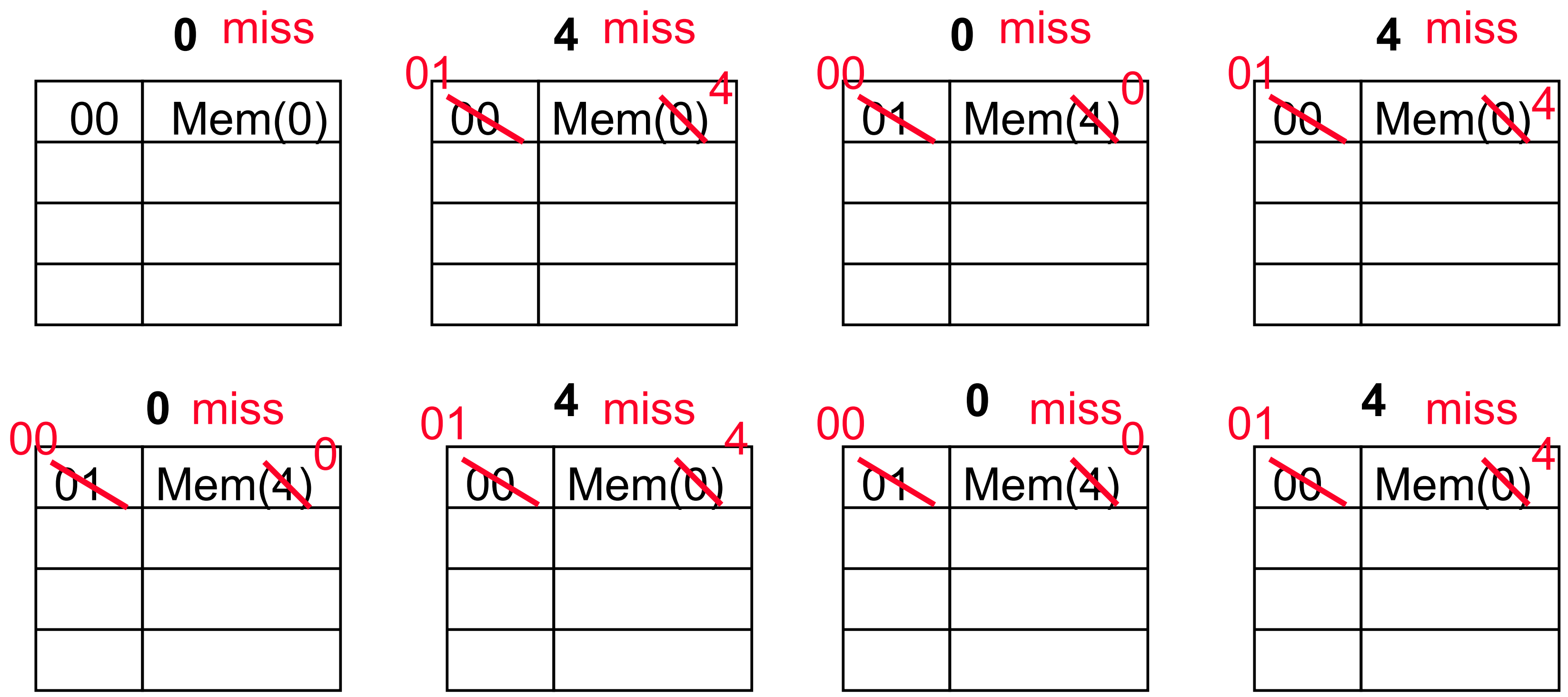
(block address) modulo (# sets in the cache)

Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

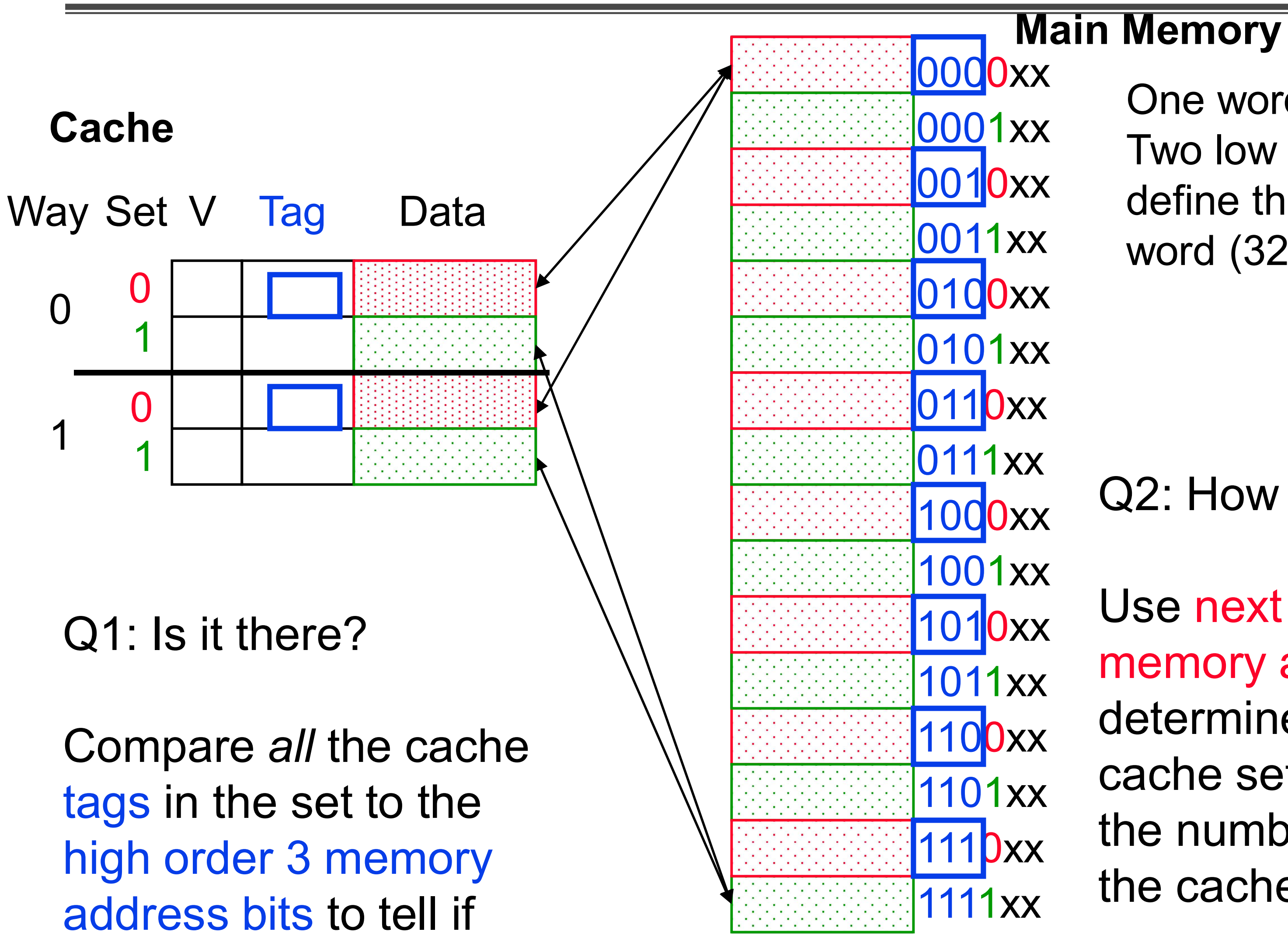
0 4 0 4 0 4 0 4



- 8 requests, 8 misses

- Ping pong effect due to conflict misses - two memory locations that map into the same cache block

Set Associative Cache Example



One word blocks
Two low order bits define the byte in the word (32b words)

Q2: How do we find it?

Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache)

Q1: Is it there?

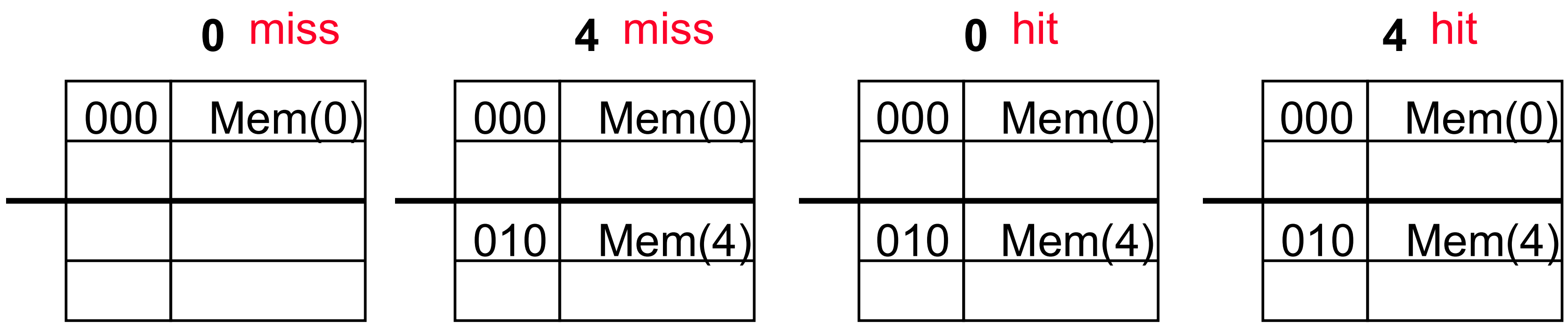
Compare *all* the cache tags in the set to the high order 3 memory address bits to tell if the memory block is in the cache

Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4

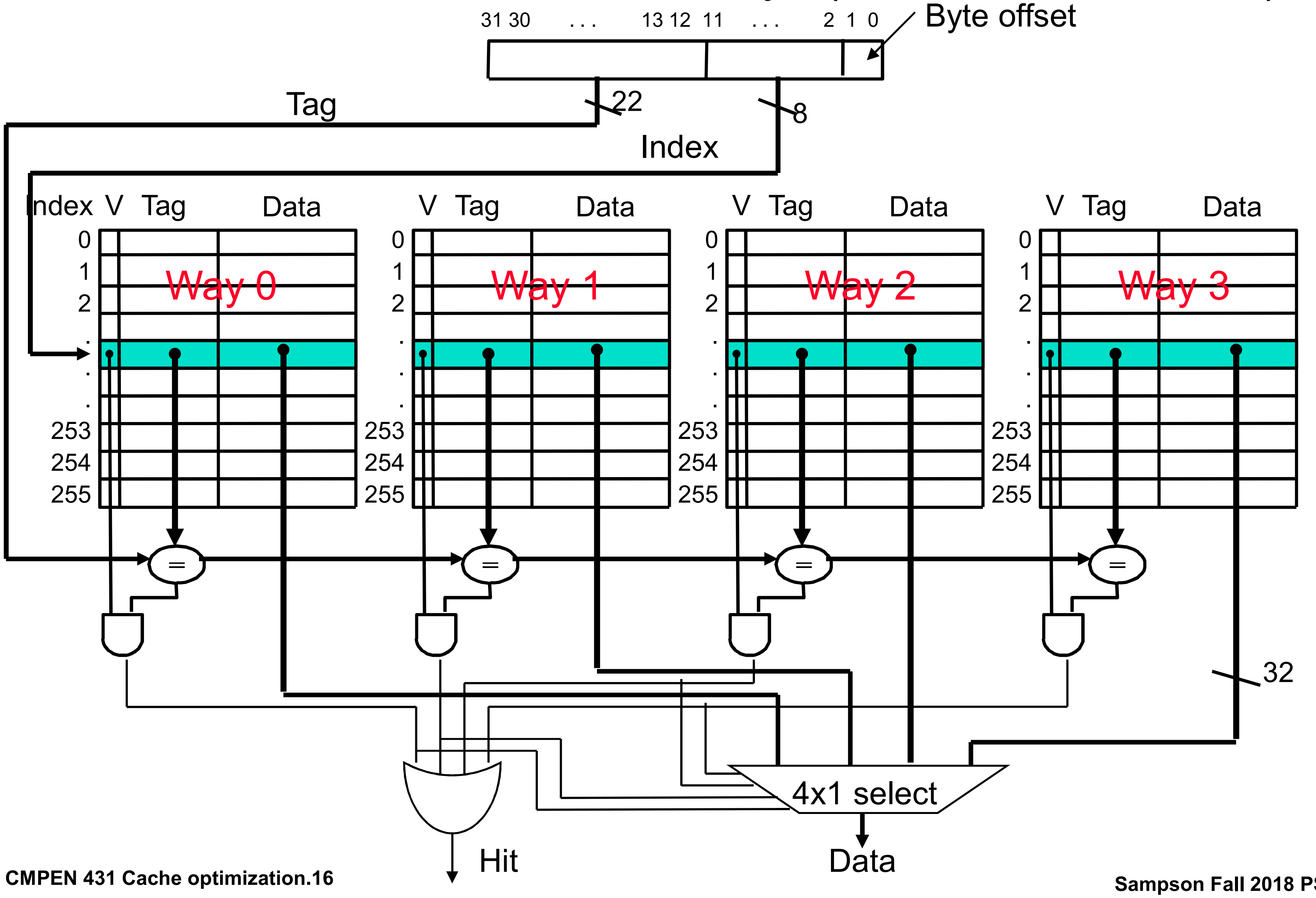


- 8 requests, 2 misses

- Solves the ping pong effect in a direct mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!

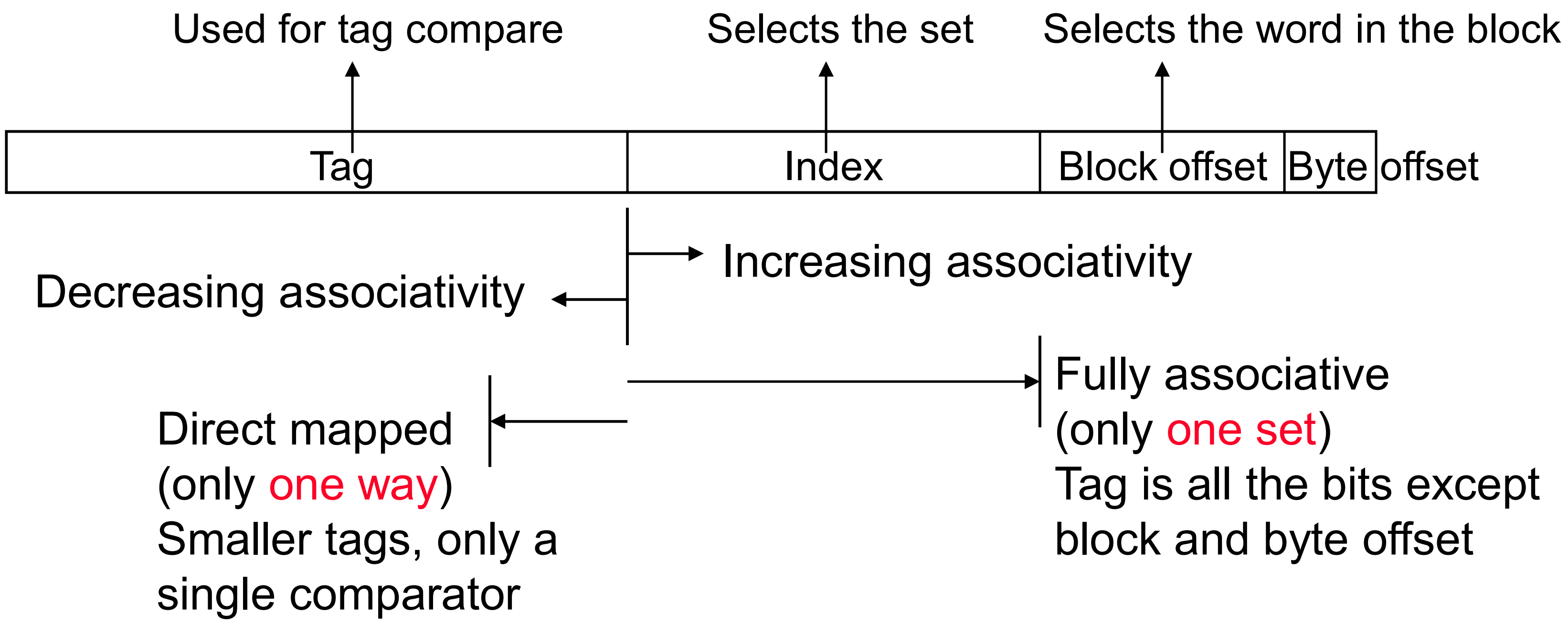
Four-Way Set Associative 4KB Cache

- 2^8 = 256 sets in each of four ways (each with one block)



Range of Set Associative Caches

- For a **fixed size** cache, each increase by a factor of two in associativity doubles the number of ways (i.e., doubles the number of blocks per set) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

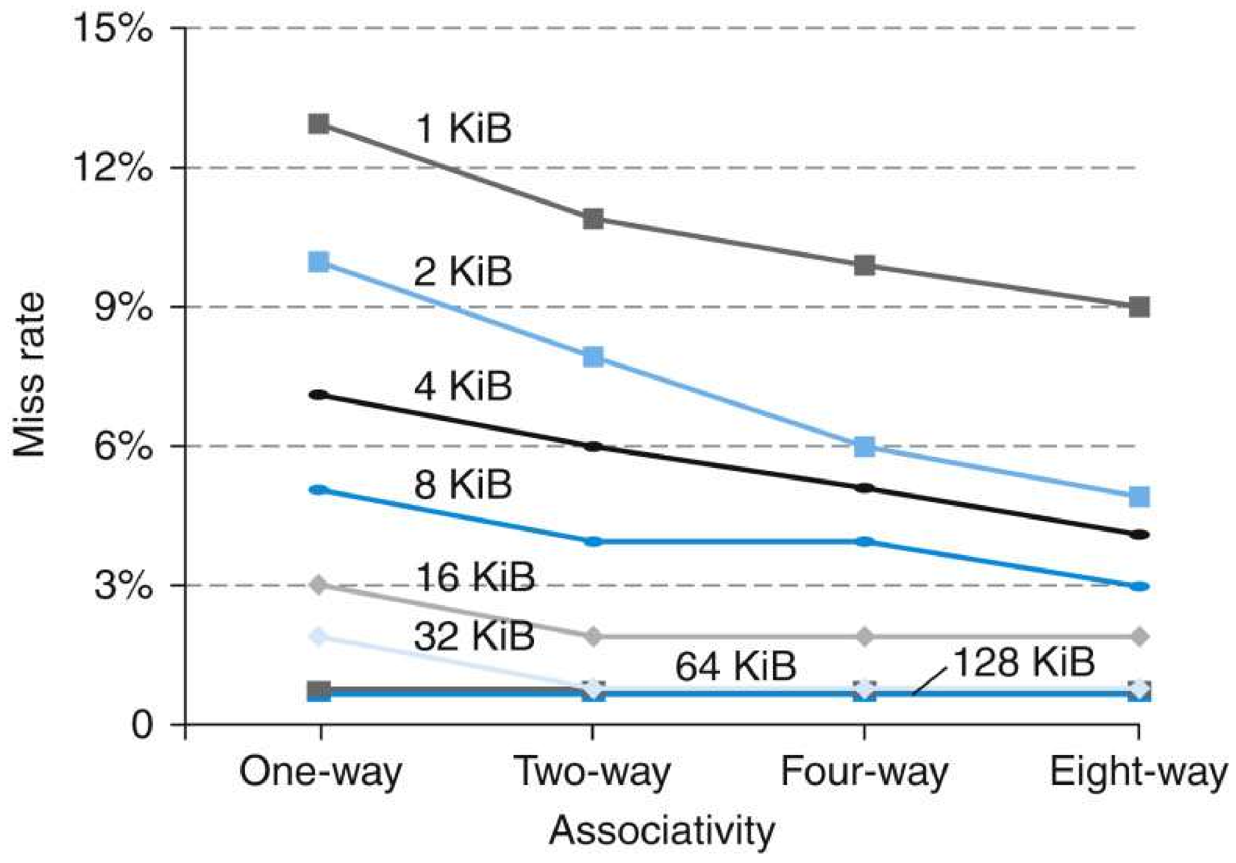


Costs of Set Associative Caches

- When a miss occurs, which way's block do we pick for replacement?
  - Least Recently Used (LRU): the block replaced is the one that has been unused for the longest time
    - Must have hardware to keep track of when each way's block was used relative to the other blocks in the set
    - For 2-way set associative, takes one bit per way → set the bit when a block is referenced (and reset the other way's bit)
- Additional costs of N-way set associative caches
  - N comparators (delay and area) – one per way
  - MUX delay (way selection) before data is available
  - Data available after way selection (and Hit/Miss decision). In a direct mapped cache, the cache block is available before the Hit/Miss decision
    - So its not possible to just assume a hit and continue and recover later if it was a miss

Benefits of Set Associative Caches

- The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation



- Largest gains are in going from direct mapped (1-way) to 2-way (20%+ reduction in miss rate)

Reducing Cache Miss Rates #2

2. Use multiple levels of caches

- With advancing technology have more than enough room on the die for bigger L1 caches *and* for a second level of caches – normally a unified L2 cache (i.e., it holds both instructions and data) and even a unified L3 cache
- For our example, CPI_{ideal} of 2, 100 cycle miss penalty (to main memory) and a 25 cycle miss penalty (to UL2\$), 36% load/stores, a 2% (4%) L1I\$ (L1D\$) miss rate, add a 0.5% UL2\$ miss rate

$$CPI_{stalls\ (L2)} = 2 + 1 * .02*(\underbrace{25 + .005*(100)}) + .36*.04(\underbrace{25 + .005*100}) = 2.8772$$

$$CPI_{stalls\ (NO\text{-}L2)} = 2 + 1 * .02*(100) + .36*.04(100) = 5.44$$

Multilevel Cache Design Considerations

- Design considerations for L1 and L2 caches are very different
  - Primary cache should focus on minimizing hit time in support of a shorter clock cycle
    - Smaller with smaller block sizes
  - Secondary cache(s) should focus on reducing miss rate to reduce the penalty of long main memory access times
    - Larger with larger block sizes
    - Higher levels of associativity
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate
- For the L2 cache, hit time is less important than miss rate
  - The L2$ hit time determines L1$'s miss penalty
  - L2$ local miss rate >> than the global miss rate

Caches & AMAT

- Three key things to target: **Access time**, **miss rate**, **miss penalty**
- A larger cache will (almost always) have a longer access time, a smaller cache will (usually) have a higher miss rate. Local miss rate tends to increase over hierarchy.
- A hierarchy of (balanced) caches will usually manifest as a lower (effective) miss penalty. Consider:

AMAT = 1 cycle/acc$_{L1}$ + 0.02 misses/acc$_{L1}$ x 50 cycles/miss$_{L1}$ =
2 cycles/acc

versus

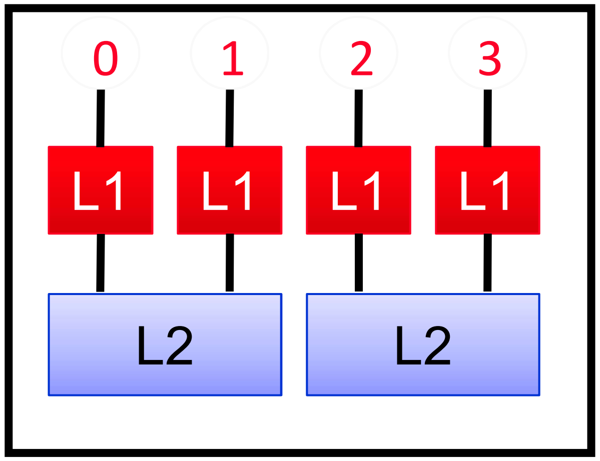
AMAT = 1 cycle/acc$_{L1}$ + 0.02 misses/acc$_{L1}$ x
(10 cycles$_{L2}$ + 0.10 misses/acc$_{L2}$ x (50 cycles/miss$_{L2}$) =
1 cycle/acc$_{L1}$ + 0.02 misses/acc$_{L1}$ x *15 cycles/miss$_{L1\text{-effective}}$* =
1.3 cycles/acc

Split vs Unified Caches

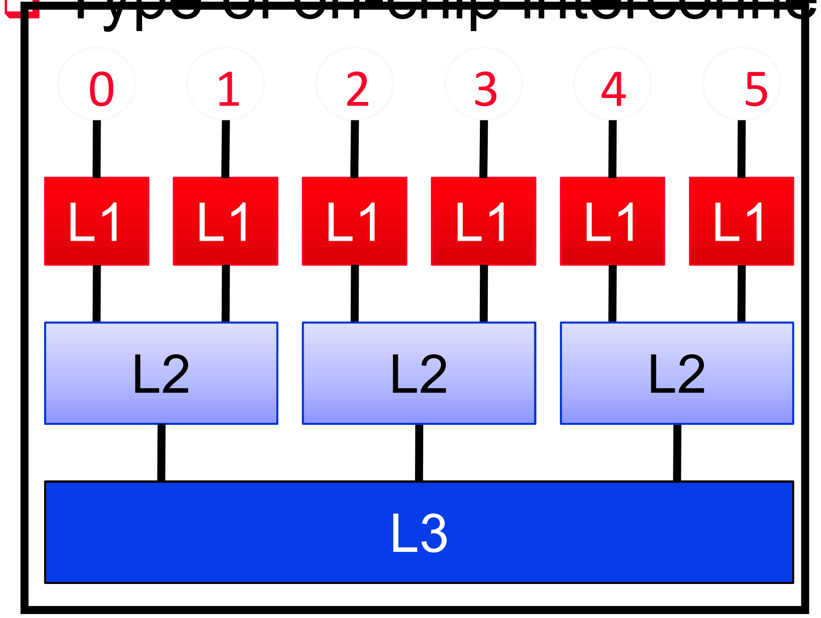
- Split L1I$ and L1D$: instr's & data in different caches at L1
  - To minimize structural hazards and t_{hit}
    - So low capacity/associativity (to reduce t_{hit})
    - So small to medium block size (to reduce conflict misses)
  - To optimize L1I$ for wide output (superscalar) and no writes
  - In multicores, private L1I$ and L1D$ per core
- Unified L2, L3, …: instr's and data together in one cache
  - To minimize $\%_{miss}$ (t_{hit} is less important due to (hopefully) infrequent accesses)
    - So high capacity/associativity/block size (to reduce $\%_{miss}$)
  - Fewer capacity misses: unused instr capacity can be used for data
  - More conflict misses: instr data conflicts (smaller in large caches)
  - Instr/data structural hazards are rare (need a simultaneous L1I$ and L1D$ miss)
  - Various L2 "sharing" options in multicores

Multicore = Platform Variety

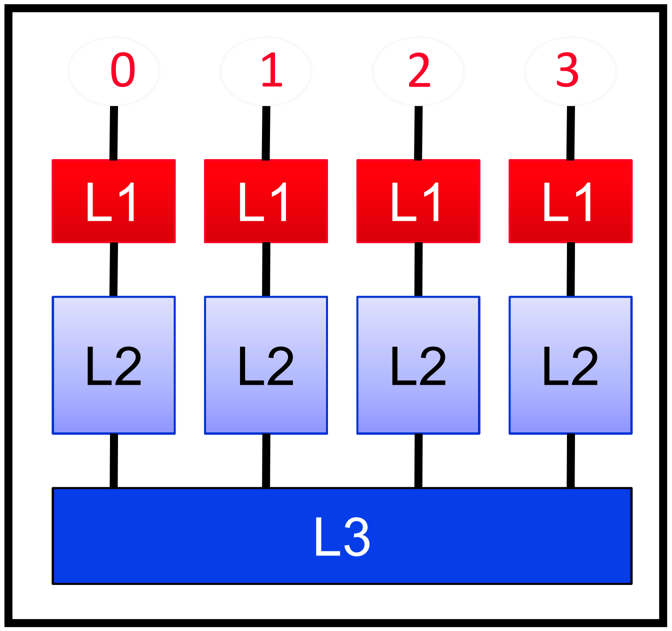
- Number/type of cores
- Levels and structure of the cache hierarchy
- Number of memory controllers (MCs) on-chip
- Type of on-chip interconnect



Harpertown

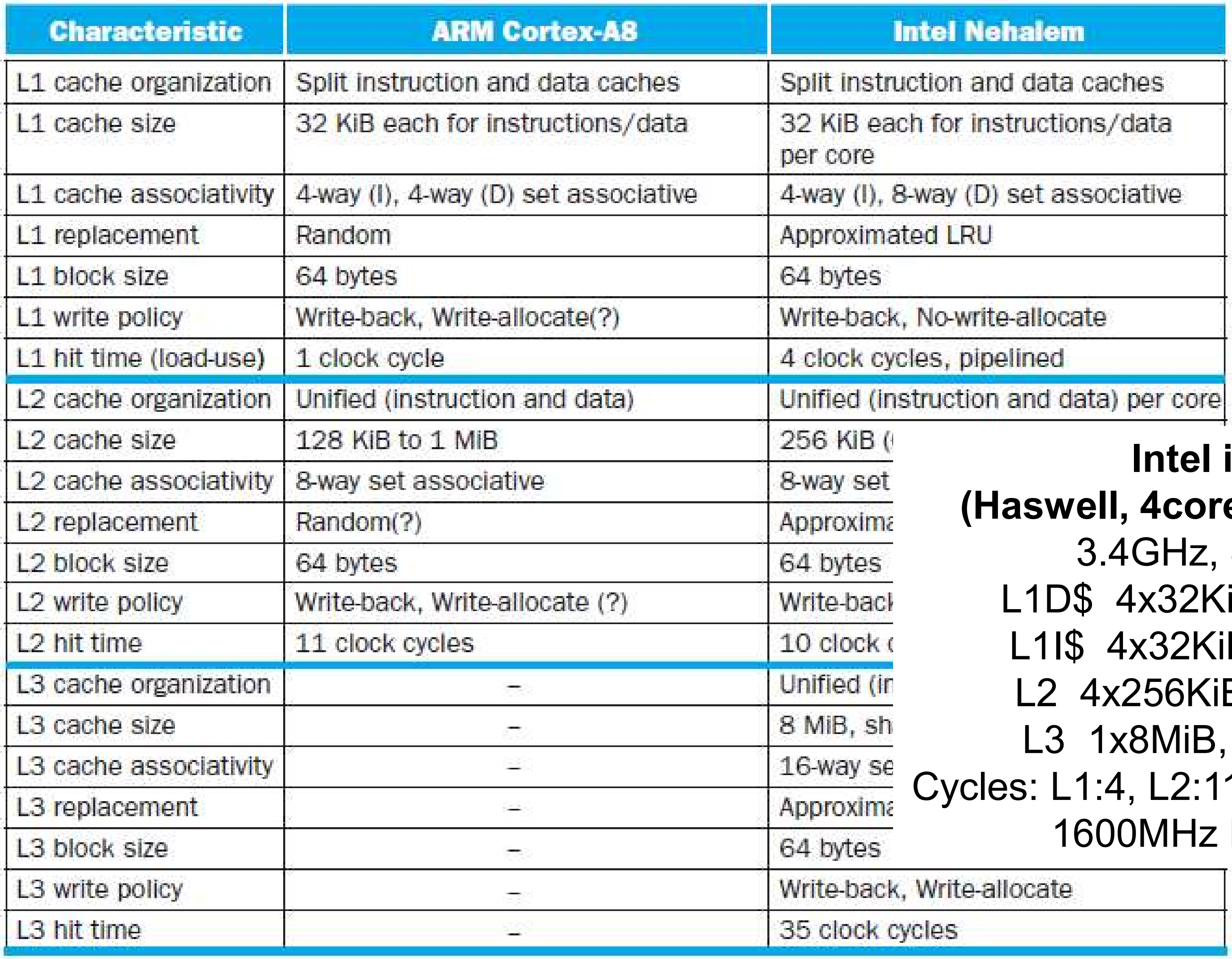


Dunnington



Nehalem

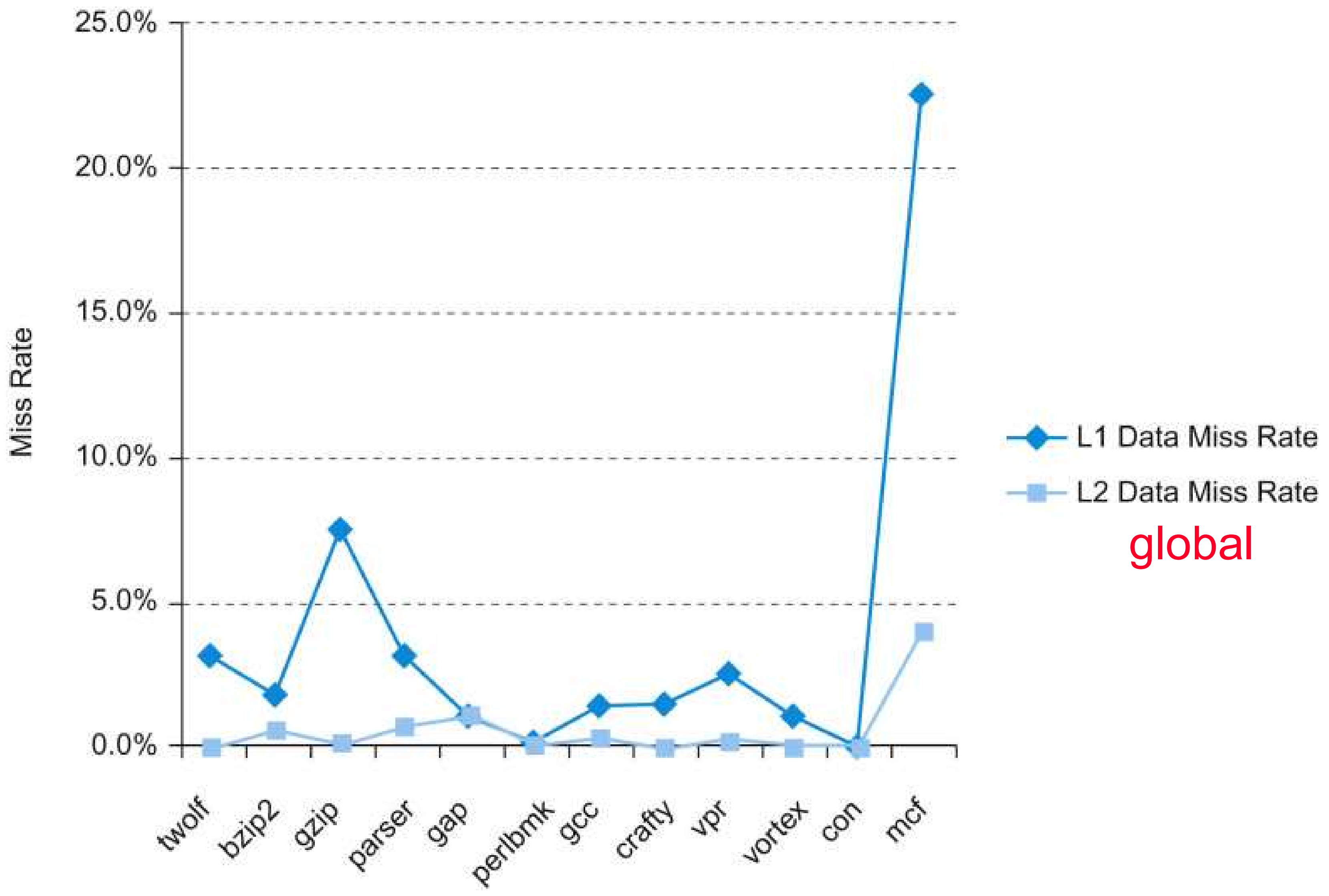
Comparing Cache Memory Architectures



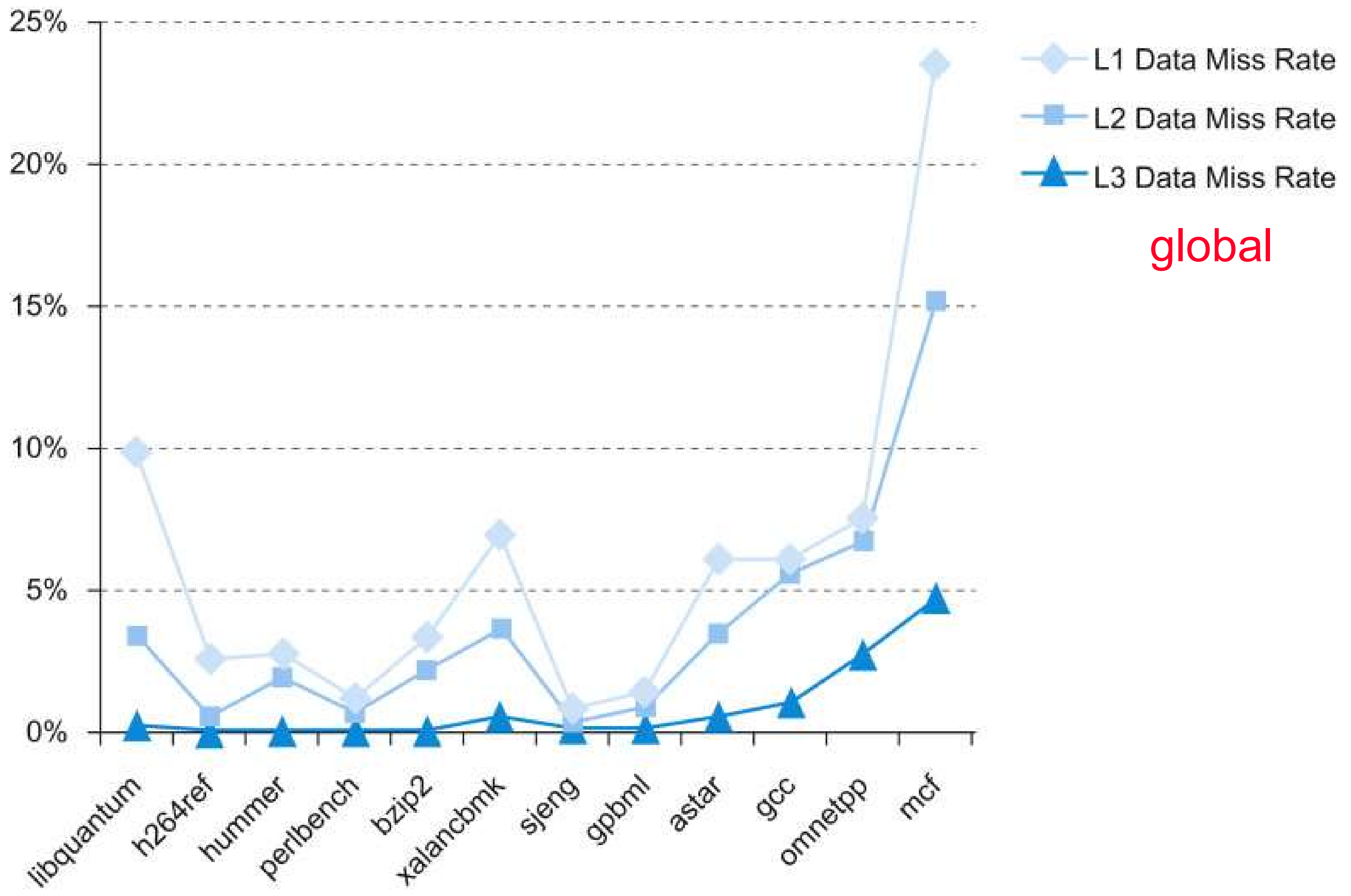
Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (
L2 cache associativity	8-way set associative	8-way set
L2 replacement	Random(?)	Approxima
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-bacl
L2 hit time	11 clock cycles	10 clock (
L3 cache organization	–	Unified (ir
L3 cache size	–	8 MiB, sh
L3 cache associativity	–	16-way se
L3 replacement	–	Approxima
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

Intel i7
(Haswell, 4core, 2wayHT)
3.4GHz, 84W
L1D$ 4x32KiB, 8-way
L1I$ 4x32KiB, 8-way
L2 4x256KiB, 8-way
L3 1x8MiB, 16-way
Cycles: L1:4, L2:11-16, L3:30-55
1600MHz DDR3

ARM Cortex-A8 Data Cache Miss Rates



Intel i7 Data Cache Miss Rates



Reducing Cache Miss Rates #3

- Hardware prefetching
  - Fetch blocks into the cache proactively (speculatively)
  - Key is to anticipate the upcoming miss addresses accurately
  - Relies on having unused memory bandwidth available
- A simple case is to use next block prefetching
  - Miss on address X → anticipate next reference miss on X + block-size
  - Works well for instr's (sequential execution) and for arrays of data
- Need to initiate prefetches sufficiently in advance
- If prefetch instr/data that is not going to be used then have polluted the cache with unnecessary data (possibly evicting useful data)

Reducing Cache Miss Rates #4: Code Transformations

- Code transformations change data access patterns, thus influencing cache hits and misses
- A large body of compiler transformations designed to maximize (data) cache performance
  - **Data Reuse => Data Locality => Cache Performance**
- Examples
  - Loop fusion
  - Loop interchange
  - Iteration space tiling (aka code blocking) – P&H 5.4
  - Statement scheduling

Loop Fusion

- Merges two adjacent countable loops into a single loop
- Reduces the cost of test and branch code
- Fusing loops that refer to the same data enhances temporal locality
- One potential drawback is that larger loop body may reduce instruction locality when the instruction cache is very small

```
for i = 1, N
   A(i) = B(i) +C(i)
for i = 1, N
   D(i) = E(i) + B(i)
```

$\Rightarrow$

```
for i = 1, N
   A(i) = B(i) +C(i)
   D(i) = E(i) + B(i)
```

Loop Interchange

- Changes the direction of array traversing by swapping two loops.
- The goal is to align data access direction with the memory layout order.

```
for i = 1, N
   for j = 1, N
        … A( j, i) …
```

$\Rightarrow$

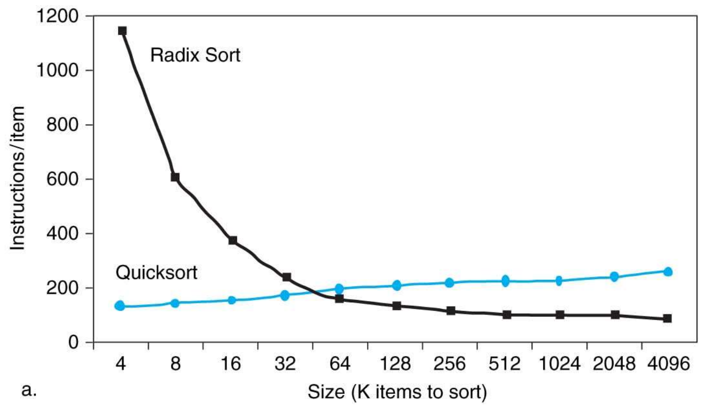
```
for j = 1, N
   for i = 1, N
        … A( j, i) …
```

assuming ROW-MAJOR memory layout

- What type of locality do we exploit?

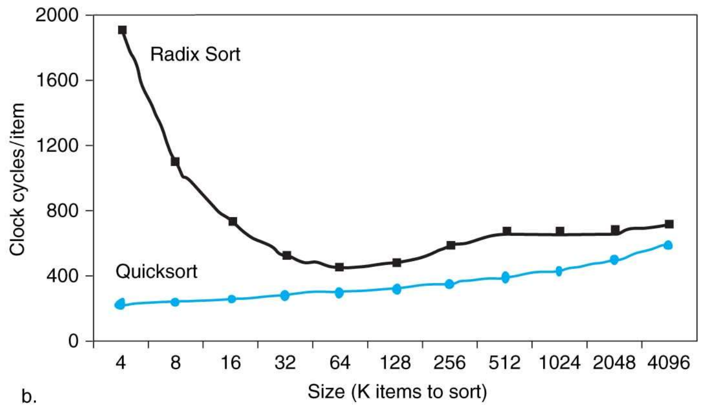
Theory vs. Practice: Using the Memory Hierarchy Well

- Comparing Quicksort and Radix Sort (from LaMarca and Ladner, 1996)



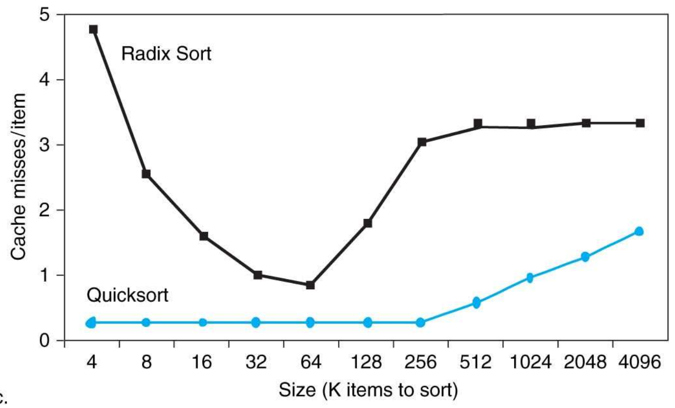
a.

- Instr executed per item sorted



b.

- Time per item sorted (why?)



c.

- Cache misses per item sorted

- Moral: **important to make sure that your algorithm/code uses the cache** well

Cache Coherence Issues – More details later

- Need a cache controller to also ensure cache coherence the most popular of which is snooping
    - The cache controller monitors (snoops) on the broadcast medium (e.g., bus) with duplicate address tag hardware (so it doesn't interfere with core's access to the cache) to determine if its cache has a copy of a block that is requested
- Write invalidate protocol – writes require exclusive access and invalidate *all* other copies
    - Exclusive access ensure that no other readable or writable copies of an item exists
- If two cores attempt to write the same data at the same time, one of them wins the race causing the other core's copy to be invalidated. For the other core to complete, it must obtain a new copy of the data which must now contain the updated value – thus enforcing write serialization

Summary: Improving Cache Performance

0. Reduce the time to hit in the cache
   - smaller cache
   - direct mapped cache
   - smaller blocks
   - for writes
     - no write allocate – no “hit” on cache, just write to write buffer
     - write allocate – to avoid two cycles (first check for hit, then write) pipeline writes via a delayed write buffer to cache

1. Reduce the miss rate
   - bigger cache
   - more flexible placement (increase associativity)
   - larger blocks (16 to 64 bytes typical)
   - victim cache – small buffer holding most recently discarded blocks

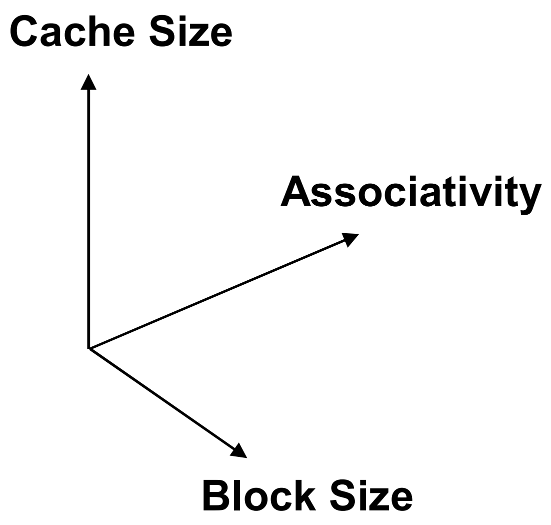
Summary: Improving Cache Performance

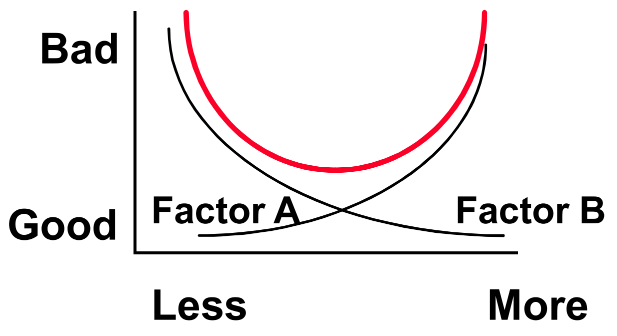
2. Reduce the miss penalty

- smaller blocks
- use a write buffer to hold dirty blocks being replaced so don’t have to wait for the write to complete before reading
- check write buffer (and/or victim cache) on read miss – may get lucky
- for large blocks fetch critical word first
- use multiple cache levels – L2 cache not tied to CPU clock rate
- faster backing store/improved memory bandwidth
  - wider buses
  - memory interleaving, DDR SDRAMs

Summary: The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins





Reminders

- Next:
  - Hardware support for virtual memory (TLBs)