CSE 431
Computer Architecture
Fall 2017
Exploiting the Memory Hierarchy: Caches

Mahmut Taylan Kandemir (www.cse.psu.edu/~kandemir)

[Adapted from *Computer Organization and Design, 5th Edition*,

Patterson & Hennessy, © 2014, MK

With additional thanks/credits to Mary Jane Irwin, Amir Roth,

Milo Martin, Onur Mutlu

Why Memory Hierarchy?

- We want both fast and large

- But, we cannot achieve both with a single level of memory

- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)

Memory Locality

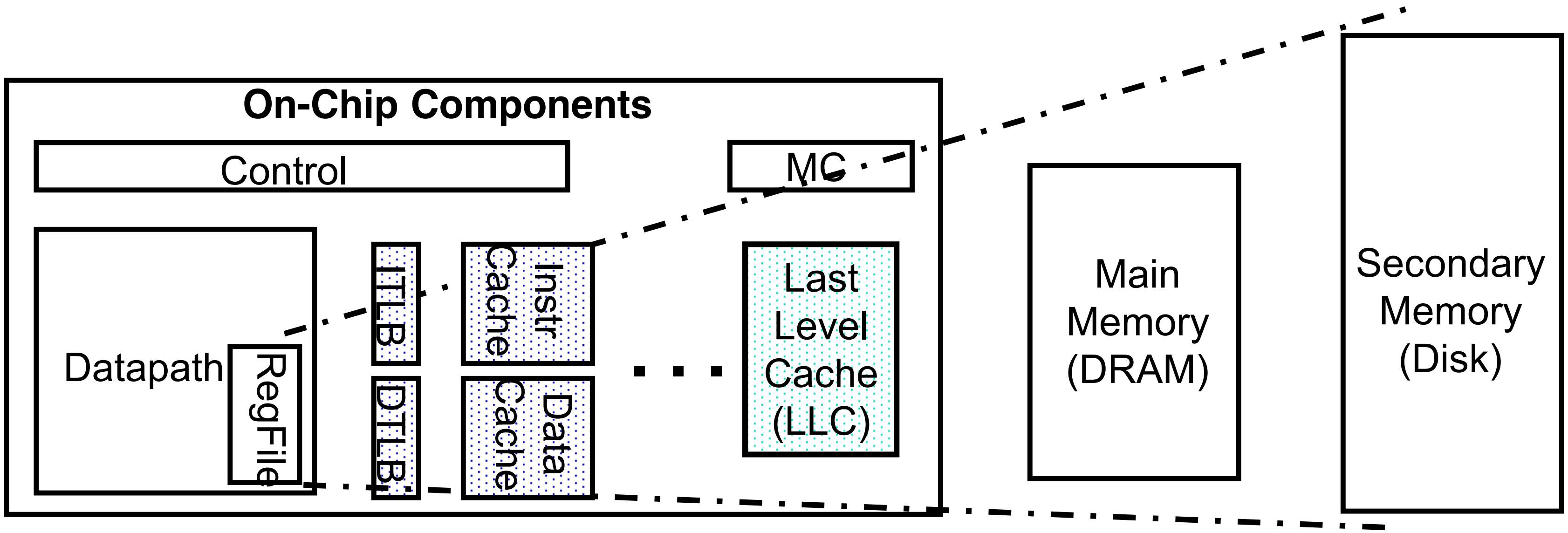
- A "typical" program has a lot of locality in memory references
  - typical programs are composed of "loops"

- Temporal: A program tends to reference the same memory location many times and all within a small window of time

- Spatial: A program tends to reference a cluster of memory locations at a time
  - most notable examples:
    - 1. instruction memory references
    - 2. array/data structure references

Review: A Typical Memory Hierarchy

- Take advantage of the principle of locality to present the user with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology



Speed (%cycles):	½'s	1's	10's	100's	10,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost:	highest				lowest

How is the Hierarchy Managed?

- registers $\leftrightarrow$ memory
  - by compiler (programmer?)
- cache $\leftrightarrow$ main memory
  - by the cache controller hardware
- main memory $\leftrightarrow$ disks
  - by the operating system (virtual memory)
  - virtual to physical address mapping assisted by the hardware (TLB)
  - by the programmer (files)

Cache Basics

- Two questions to answer (in hardware):
  - **Q1:** How do we know if a data item is in the cache?
  - **Q2:** If it is, how do we find it?

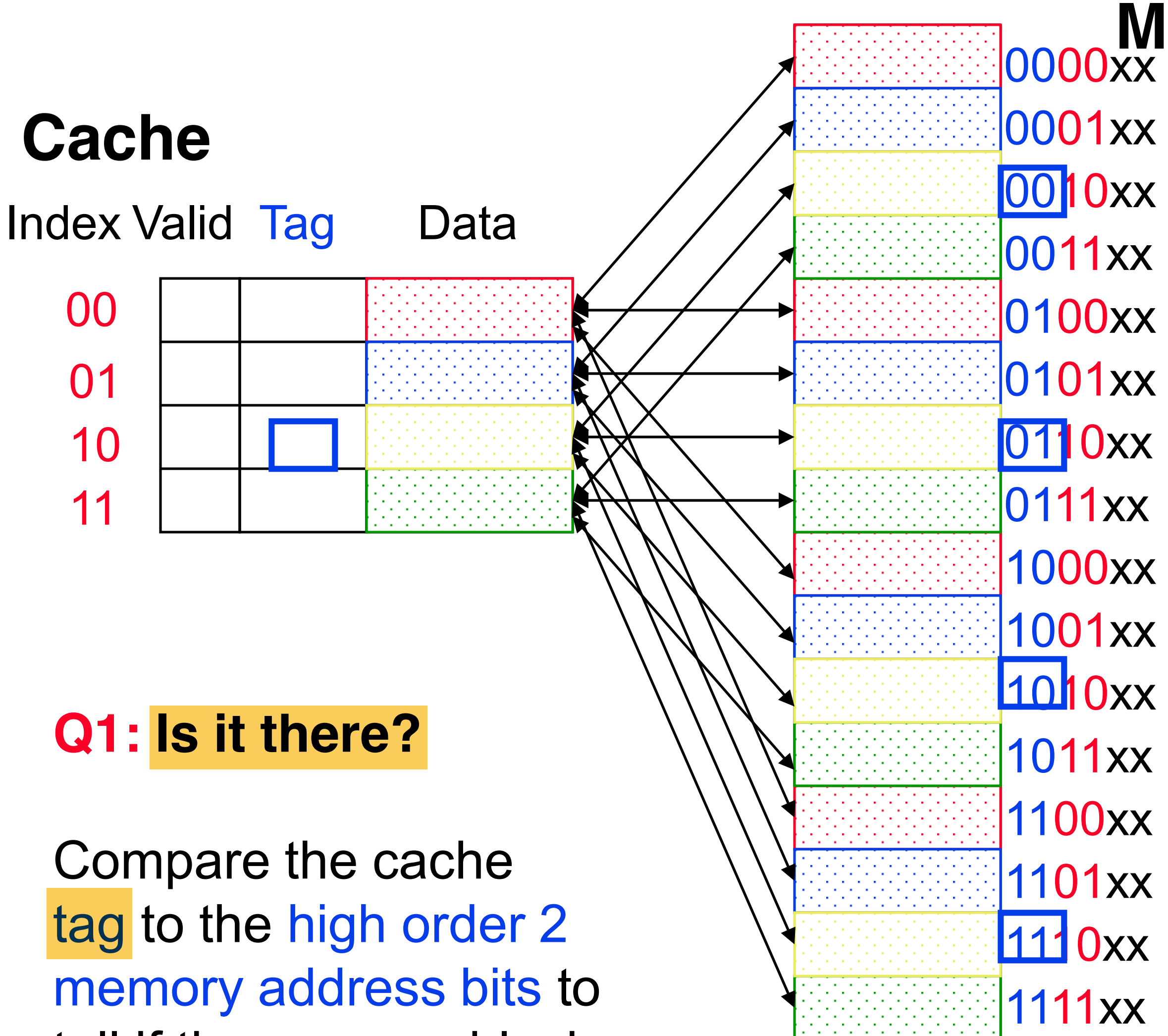
- Direct mapped cache
  - Each memory block is mapped to exactly one block in the cache
    - lots of memory blocks must share a block in the cache
  - Address mapping (to answer Q2):

    (block address) modulo (# of blocks in the cache)

  - Have a tag associated with each cache block that contains the address information (the upper portion of the address) required to identify the block (to answer Q1)

Caching: A Simple First Example

Cache



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

Q1: Is it there?

Compare the cache tag to the high order 2 memory address bits to tell if the memory block is in the cache

Q2: How do we find it?

Use next 2 low order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

(block address) modulo (# of blocks in the cache)

Direct Mapped Cache

- Consider the main memory word reference string

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

0 1 2 3 4 3 4 15

0 miss

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

01 ~~00~~	~~Mem(0)~~ 4
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

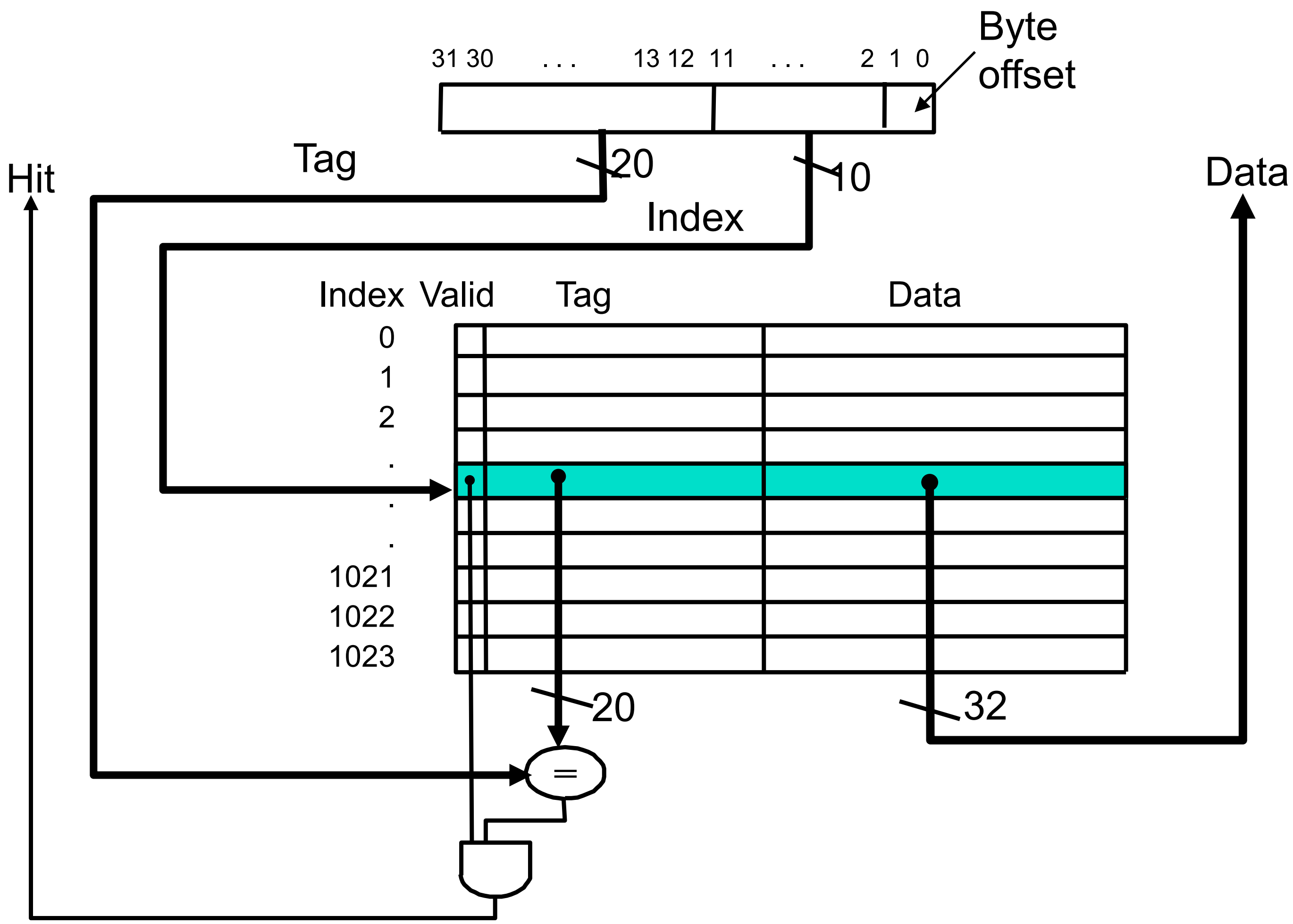
15 miss

01	Mem(4)
00	Mem(1)
00	Mem(2)
11 ~~00~~	~~Mem(3)~~ 15

- 8 requests, 6 misses

MIPS Direct Mapped 4KB Cache Example

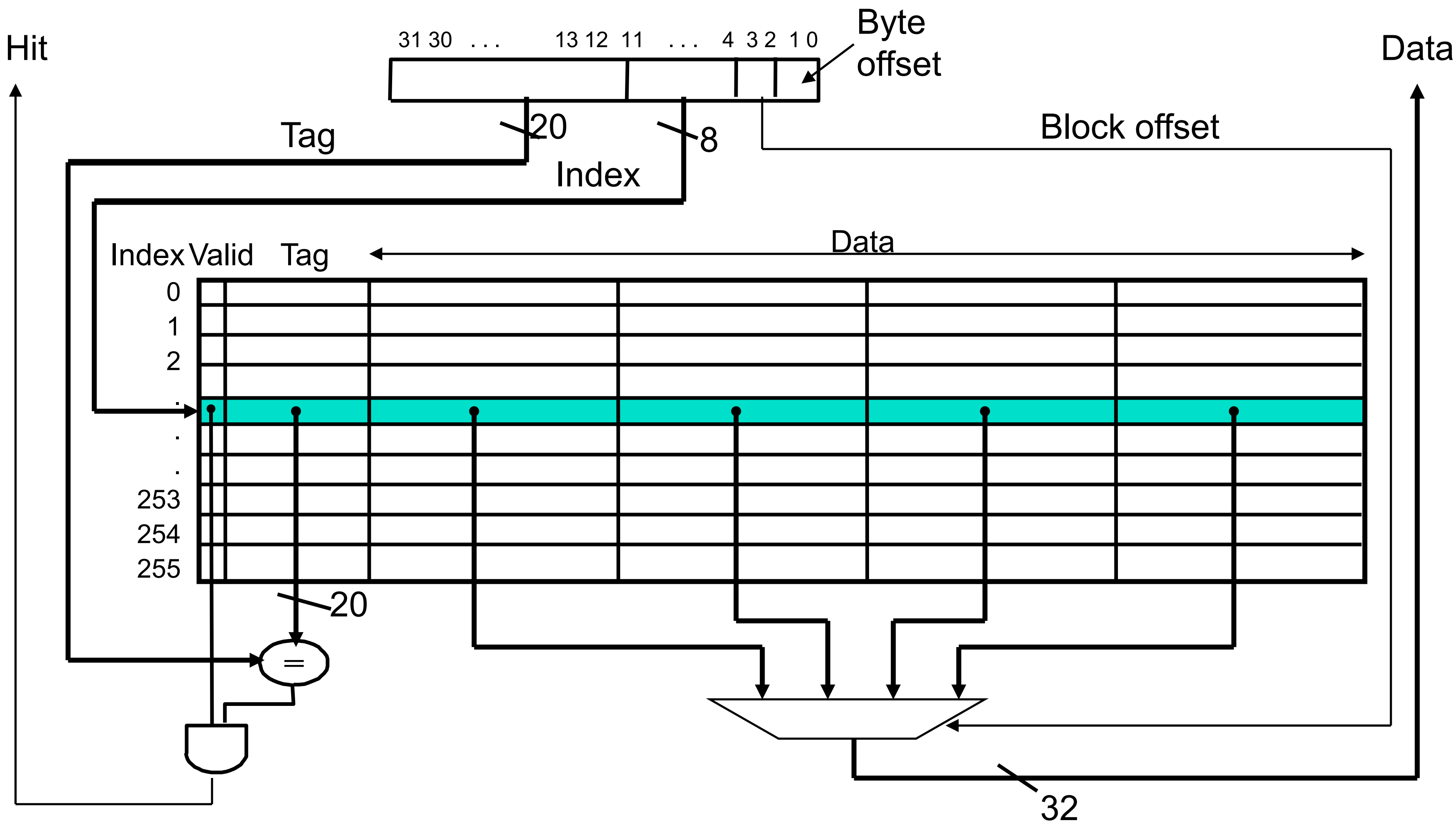
- One word blocks, cache size = 1K words (or 4KB)



What kind of locality are we taking advantage of?

Multiword Block Direct Mapped 4KB Cache

- Four words/block, cache size = 1K words



What kind of locality are we taking advantage of?

Taking Advantage of Spatial Locality

- Let cache block hold more than one word

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

0 1 2 3 4 3 4 15

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

4 miss

01 ~~00~~	~~Mem(1)~~ 5	~~Mem(0)~~ 4
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

01	Mem(5)	Mem(4)
11 ~~00~~	~~Mem(3)~~ 15	~~Mem(2)~~ 14

- 8 requests, 4 misses

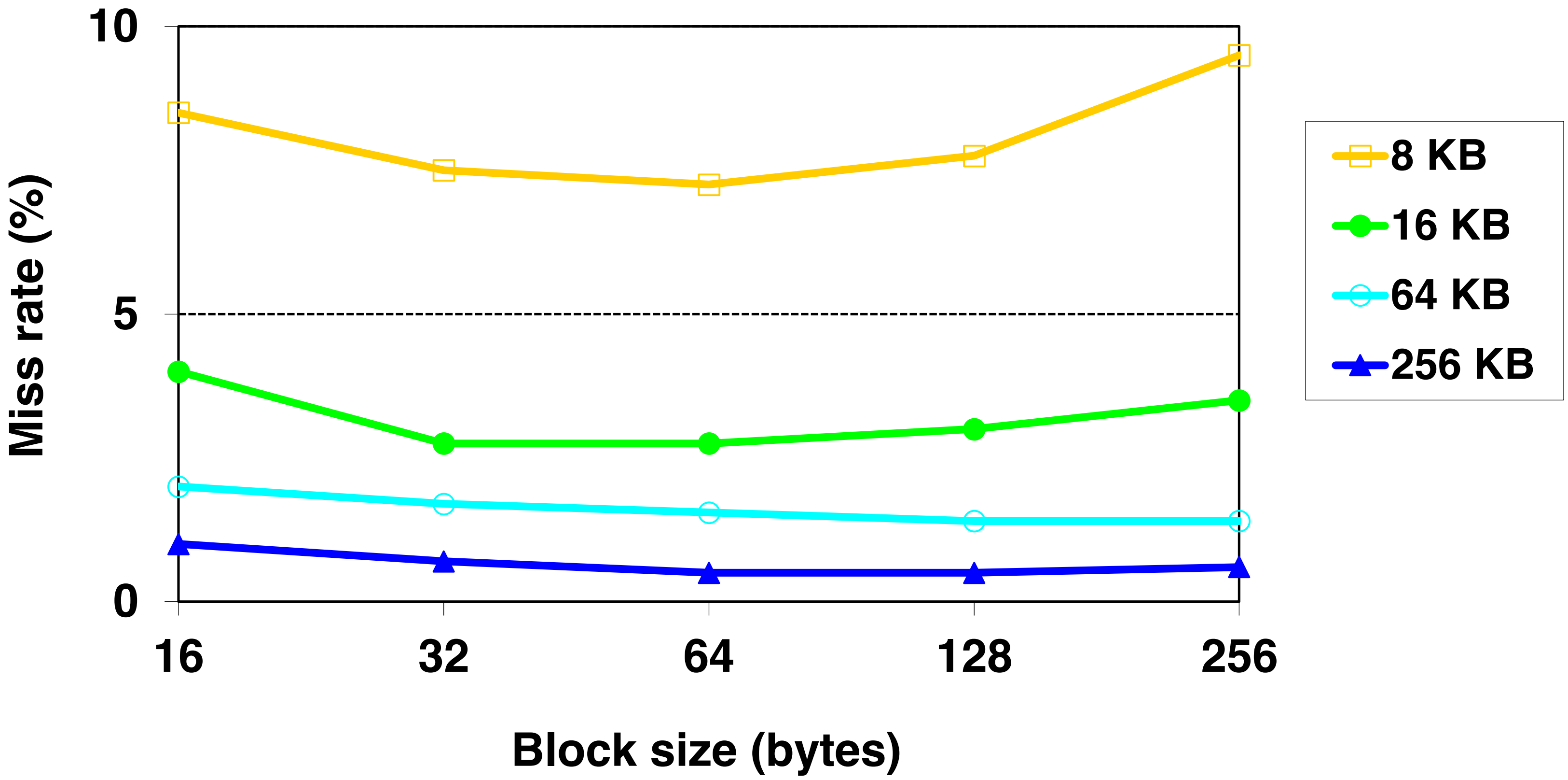
Cache Field Sizes

- The number of bits in a cache includes both the storage for data and for the tags
  - 32-bit byte address
  - A direct mapped cache with 2^n blocks has a *n* bits index
  - For a block size of 2^m words (2^{m+2} bytes), *m* bits are used to address the word within the block and 2 bits are used to address the byte within the word
- What is the size of the tag field? 32 – (n + m + 2)
- The total number of bits in a direct-mapped cache is then

  2^n x (block size + tag field size + valid field size)
- How many total bits are required for a direct mapped cache with 16KB of data and 4-word blocks assuming a 32-bit address? 16KB = 4KW (2^12) 1024 blocks (2^10)

2^10 [4x32b data + (32-10-2-2)b tag +1b valid] = 147Kb
…about 1.15 times as many as needed just for storage data

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)

Handling Cache Hits

Instruction Cache and Data Cache

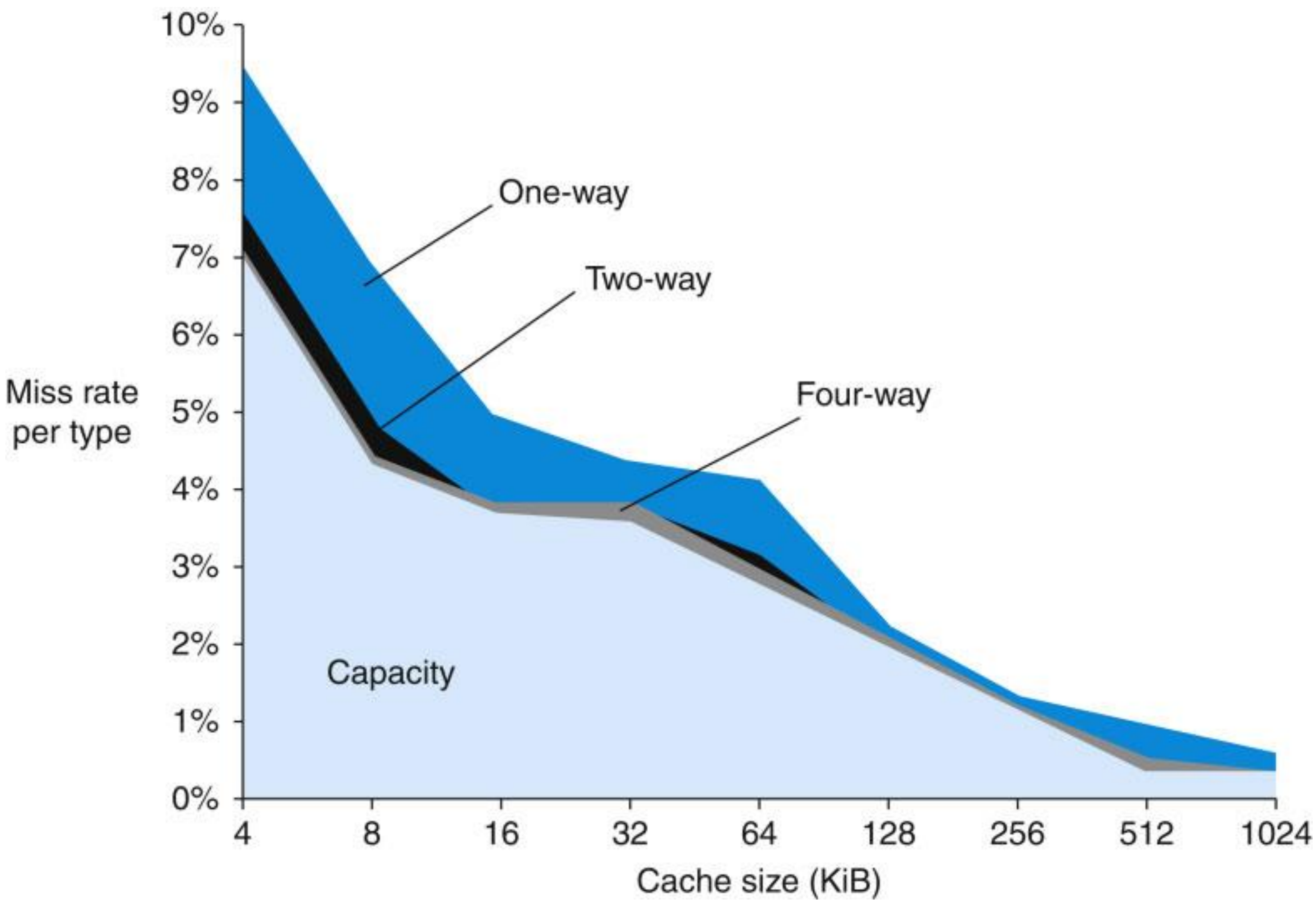
- Read hits (I$ and D$)
  - this is what we want!

- Write hits (D$ only)
  - If we require the cache and memory to be consistent
    - always write the data into both the cache block and the next level in the memory hierarchy (write-through)
    - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a write buffer and stall only if the write buffer is full
  - If we allow cache and memory to be inconsistent
    - write the data only into the cache block (write-back the cache block to the next level in the memory hierarchy when that cache block is "evicted")
    - need a dirty bit for each data cache block to tell if it needs to be written back to memory when it is evicted – can use a write buffer to help "buffer" write-backs of dirty blocks

Sources of Cache Misses (3 Cs)

- 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 Rates per Cache Miss Type



Handling Cache Misses (Single Word Blocks)

- Read misses (I$ and D$)
  - stall the pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), and send the requested word to the core, then let the pipeline resume
- Write misses (D$ only)
  1. stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the core to the cache, then let the pipeline resume
  2. Write allocate – just write the word (and its tag) into the cache (which may involve having to evict a dirty block if using a write-back cache), no need to check for cache hit, no need to stall
  3. No-write allocate – skip the cache write (but must invalidate that cache block since it will now hold stale data) and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn't full

Multiword Block Considerations

- Read misses (I$ and D$)
  - Processed the same as for single word blocks – a miss returns the entire block from memory
  - Miss penalty grows as block size grows
    - Early restart – core resumes execution as soon as the requested word of the block is returned
    - Requested word first – requested word is transferred from the memory to the cache (and core) first
  - Nonblocking cache – allows the core to continue to access the cache while the cache is handling an earlier miss
- Write misses (D$ only)
  - If using write allocate must *first* fetch the block from memory and then write the word to the block (or could end up with a “garbled” block in the cache (e.g., for 4 word blocks, a new tag, one word of data from the new block, and three words of data from the old block)

Measuring Cache Performance

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

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

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

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

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

$$\text{Memory-stall cycles} = 2\% \times 100 + 36\% \times 4\% \times 100 = 3.44$$

$$\text{So} \quad CPI_{stalls} = 2 + 3.44 = \mathbf{5.44}$$

more than twice the CPI_{ideal} !

Impacts of Cache Performance, Con't

- Relative cache penalty increases as core 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 core clock rate is doubled (doubling the miss penalty)?

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

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

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

- What is the AMAT for a core with a 20 psec clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?

$$\text{AMAT} = 1 + 0.02 \times 50 = 2 \text{ cycles}$$

Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches

- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches

Improving Cache Performance #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)

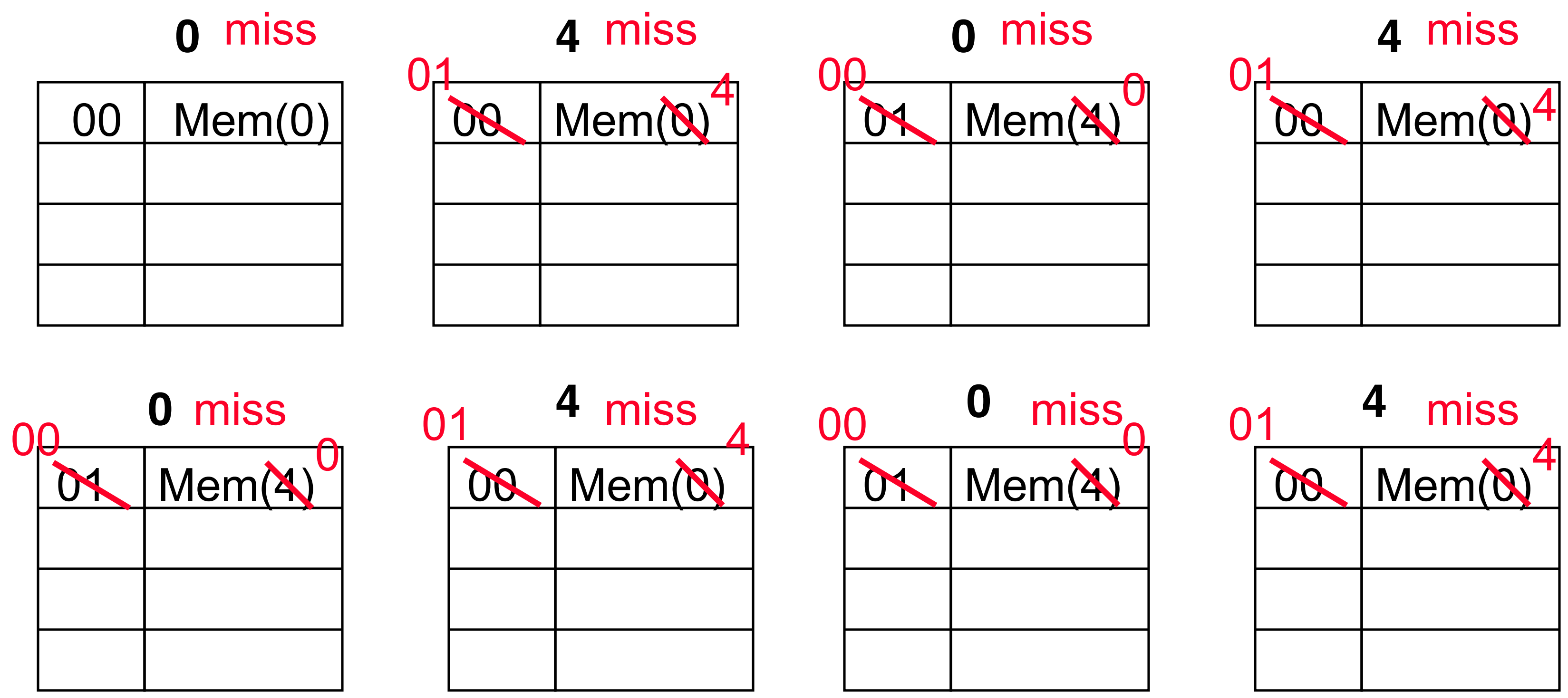
(block address) modulo (# sets in the cache)

Another Reference String Mapping

- Consider the main memory word reference string

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

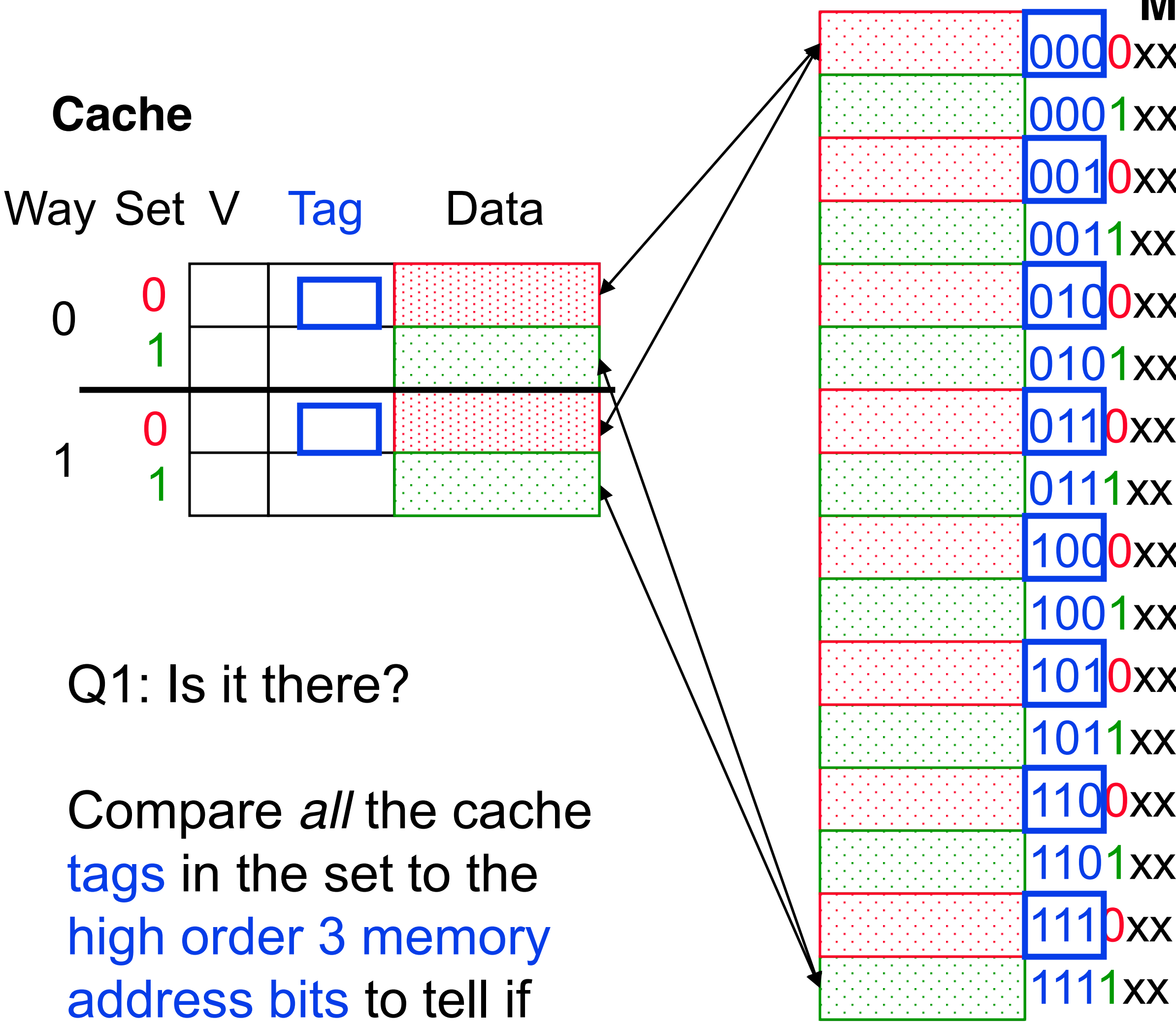
0 4 0 4 0 4 0 4



- 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)

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

Q2: How do we find it?

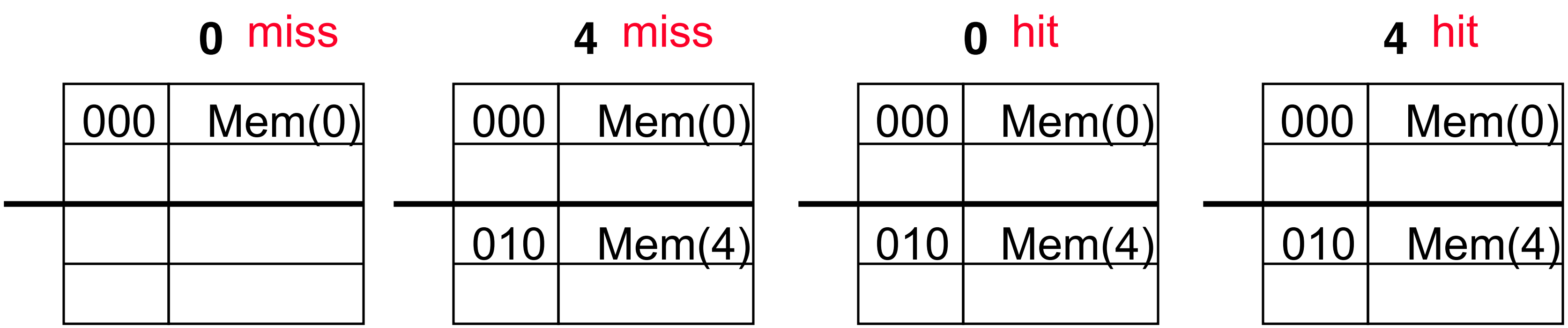
Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache)

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

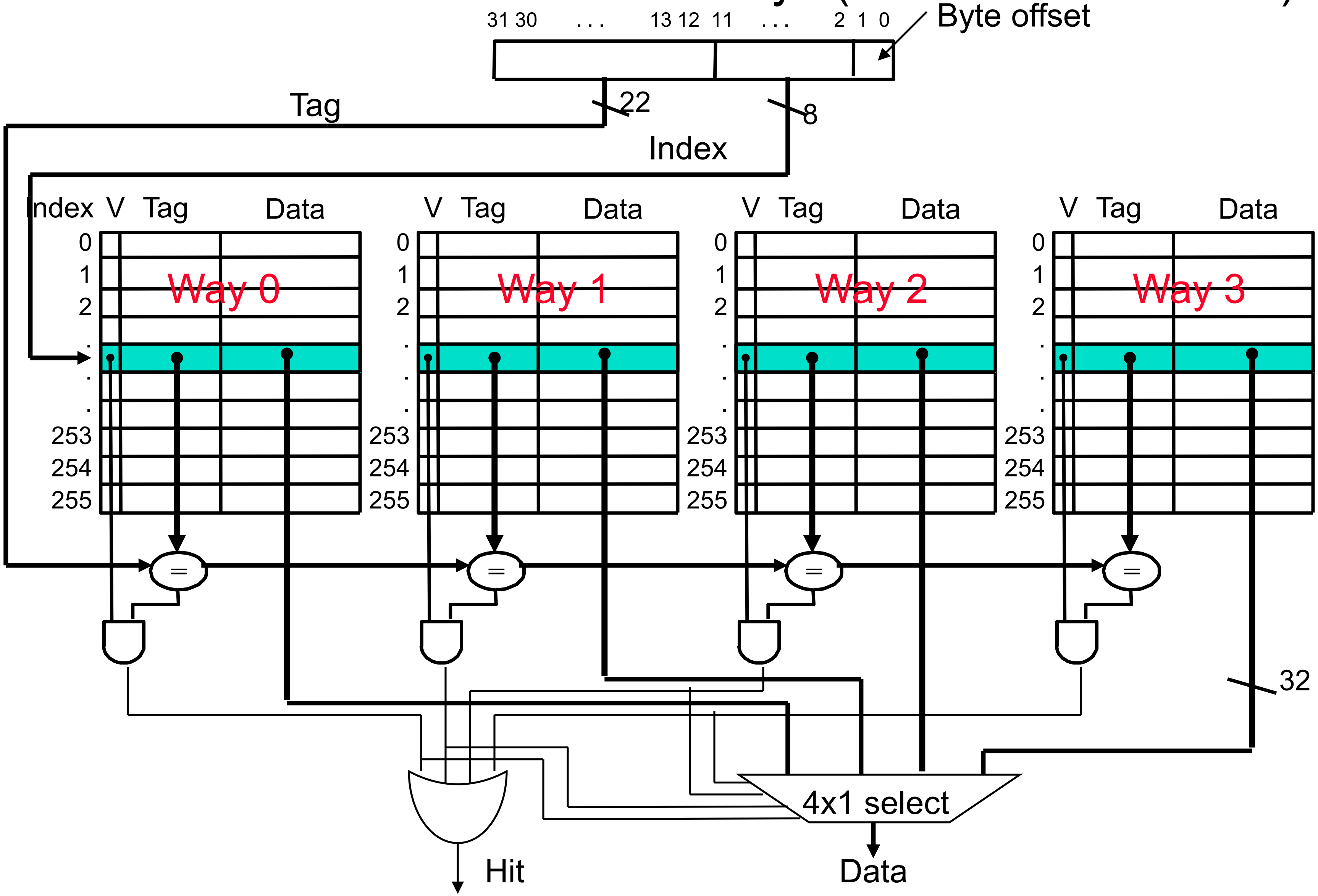


- 8 requests, 2 misses

- 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!

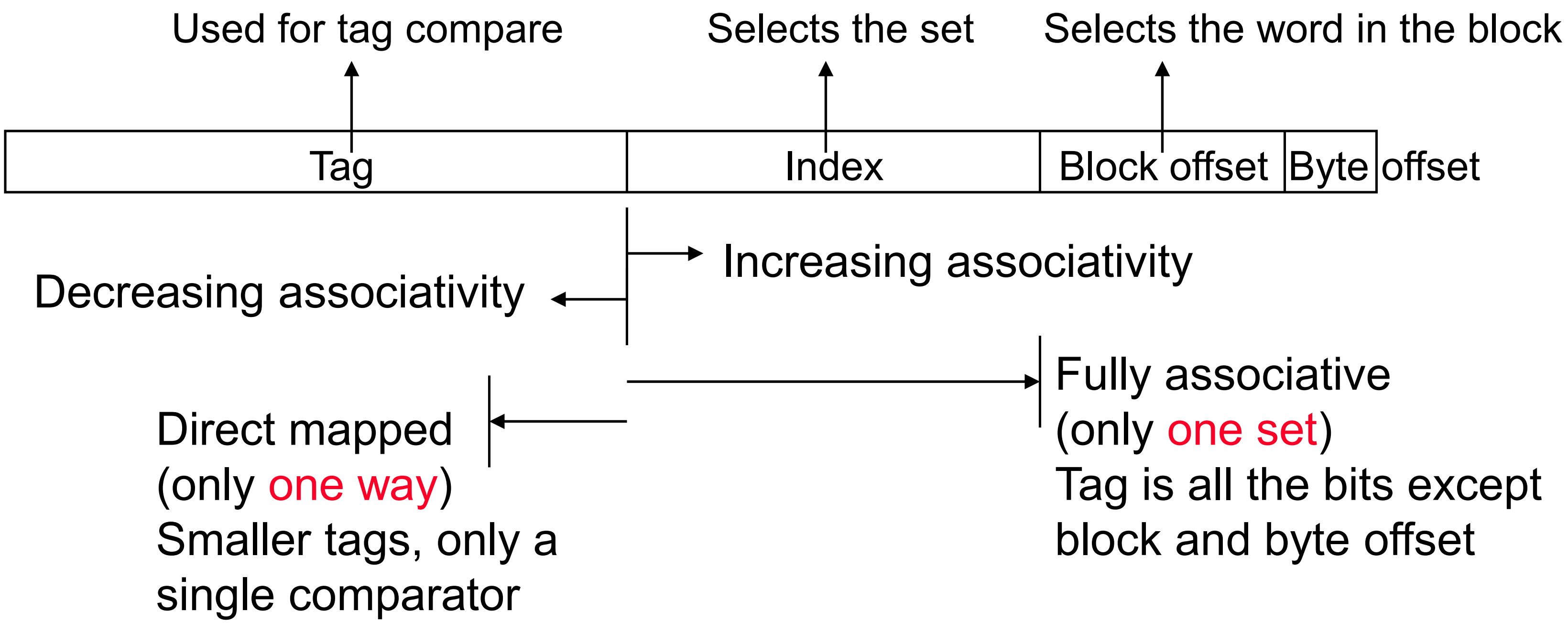
Four-Way Set Associative 4KB Cache

- 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 blocks per set (i.e., the number or ways) 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

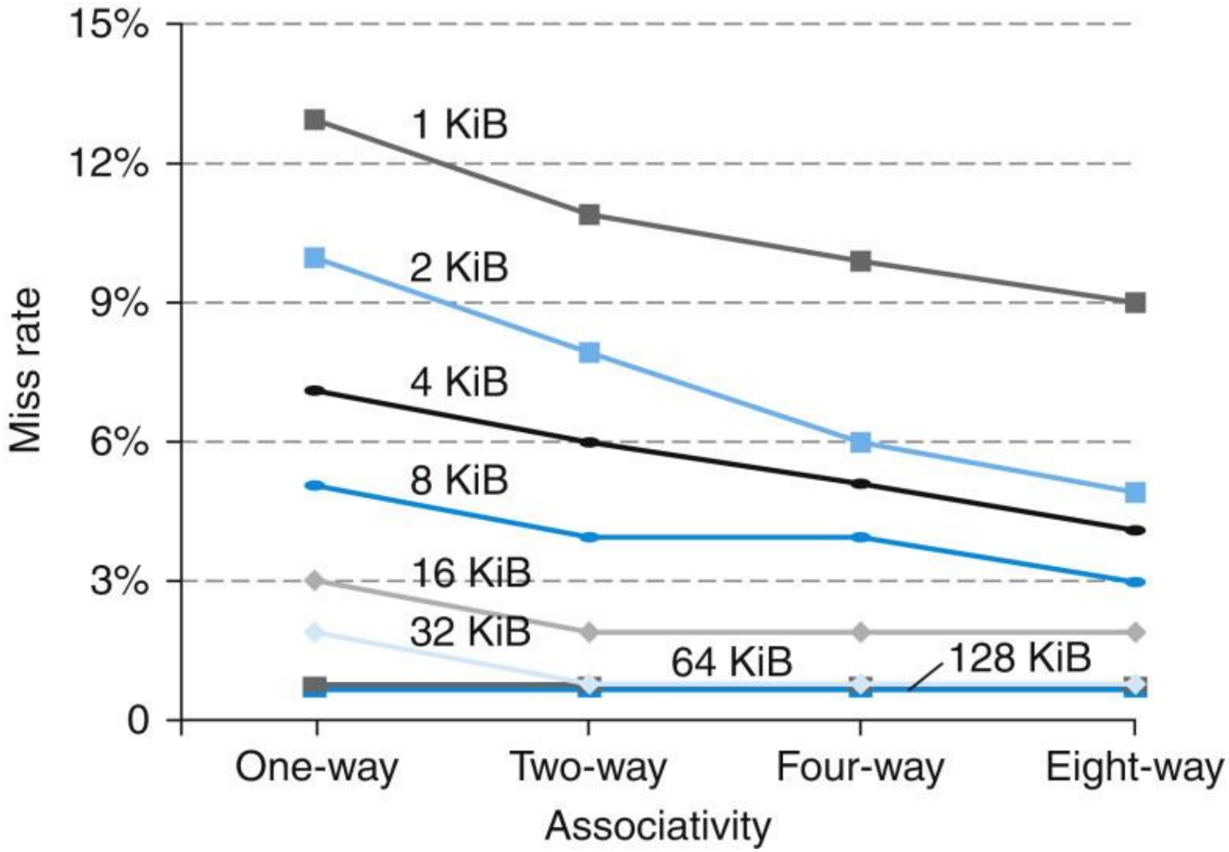


Costs of Set Associative Caches

- 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)
- N-way set associative cache additional costs
  - 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 to 2-way (20%+ reduction in miss rate)

Implementing LRU

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access ordering of blocks

- Question: 2-way set associative cache:
  - What do you need to implement LRU?

- Question: 4-way set associative cache:
  - How many different orderings possible for the 4 blocks in the set?
  - How many bits needed to encode the LRU order of a block?
  - What is the logic needed to determine the LRU victim?

Approximations of LRU

- Most modern processors do not implement "true LRU" in highly-associative caches

- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible replacement policy)

- Examples:
  - Not MRU (not most recently used)
  - Hierarchical LRU: divide the 4-way set into 2-way "groups", track the MRU group and the MRU way in each group
  - Victim-NextVictim Replacement: Only keep track of the victim and the next victim

Improving Cache Performance #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

$$CPI_{stalls} = 2 + .02\times25 + .36\times.04\times25 + .005\times100 + .36\times.005\times100 = 3.54$$

(as compared to 5.44 with no L2$)

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 >> the global miss rate

Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity
  - Tag store and data store accessed in parallel
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Tag store and data store accessed serially
- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter

Split vs Unified Caches

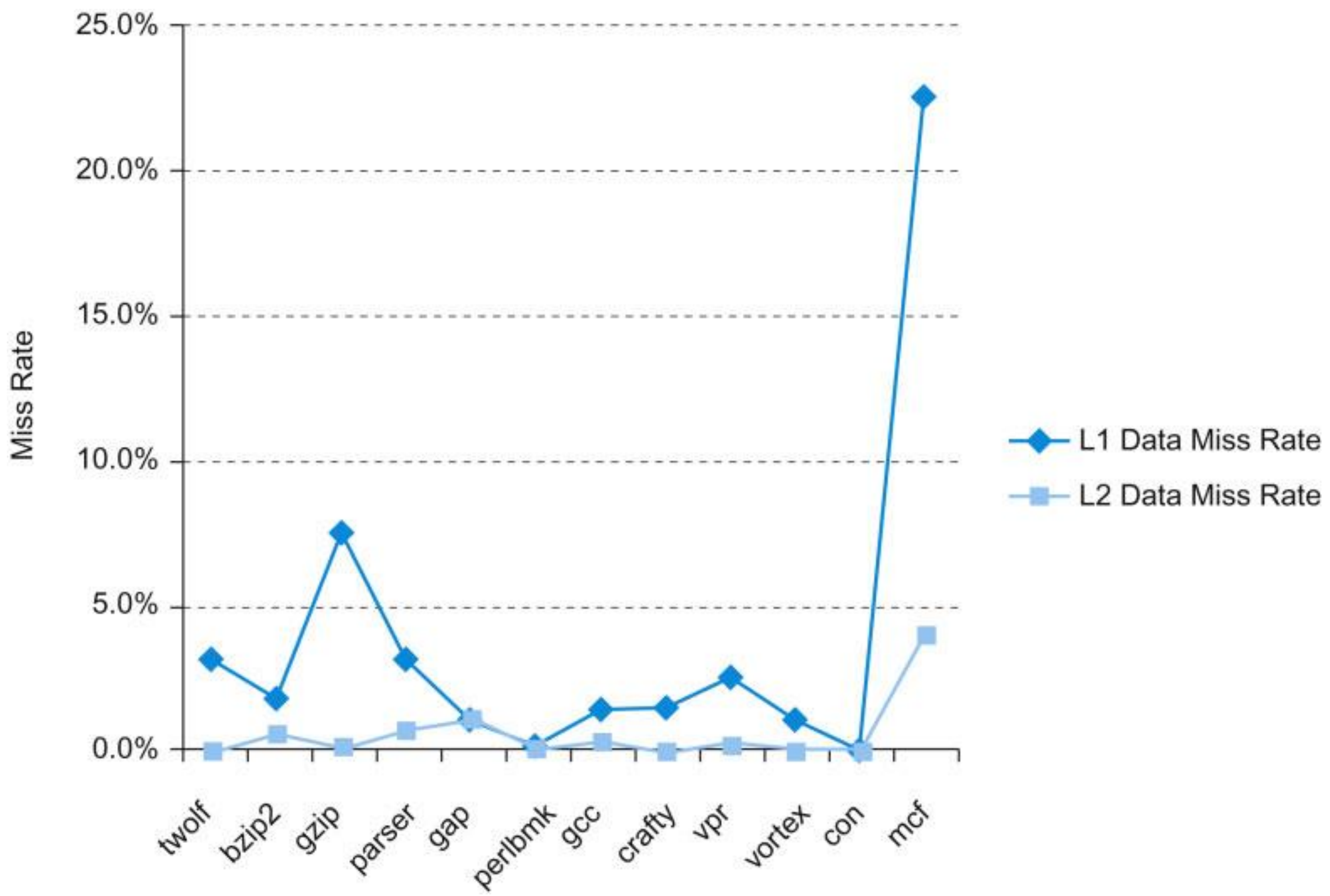
- Split L1I\$ and L1D\$: instr's and data in different caches at L1
  - To minimize structural hazards and t_{hit}
    - So low capacity/associativity (to reduce t_{hit})
    - So small to medium block size (to reduce conflict misses)
  - To optimize L1I\$ for wide output (superscalar) and no writes
- 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 effect in large caches)
  - Instr / data structural hazards are rare (would take a simultaneous L1I\$ and L1D\$ miss)

Comparing Cache Memory Architectures

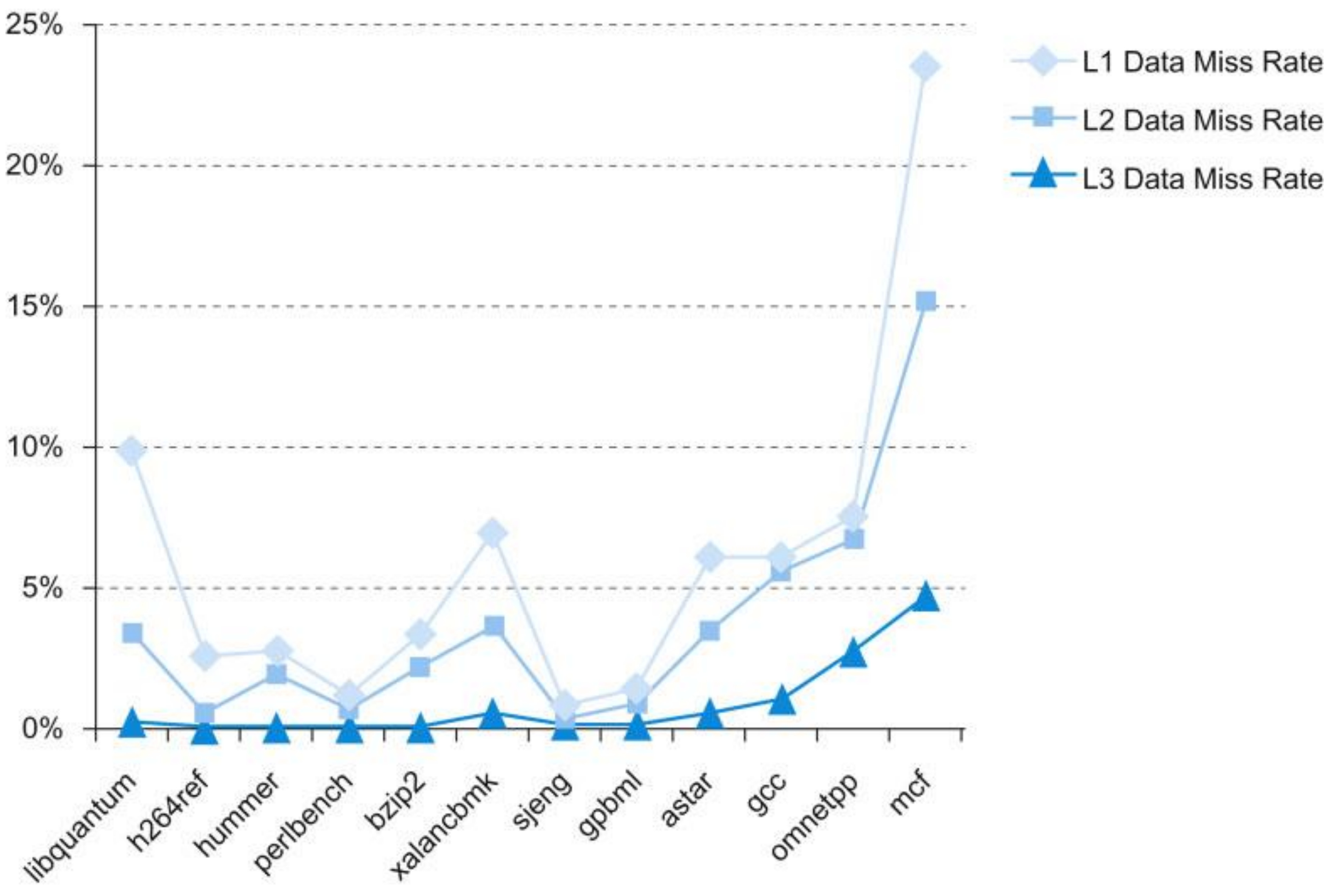
Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (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 (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacem		Approximated LRU
L2 block siz		64 bytes
L2 write poli		Write-back, Write-allocate
L2 hit time		10 clock cycles
L3 cache or		Unified (instruction and data)
L3 cache siz		8 MiB, shared
L3 cache as		16-way set associative
L3 replacem		Approximated LRU
L3 block siz		64 bytes
L3 write poli		Write-back, Write-allocate
L3 hit time		35 clock cycles

Intel i7
(Haswell, 4core, 2wayHT)
3.4GHz, 84W
L1D$ 4x32KB, 8-way
L1I$ 4x32KB, 8-way
L2 4x256KB, 8-way
L3 1x8MB, 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



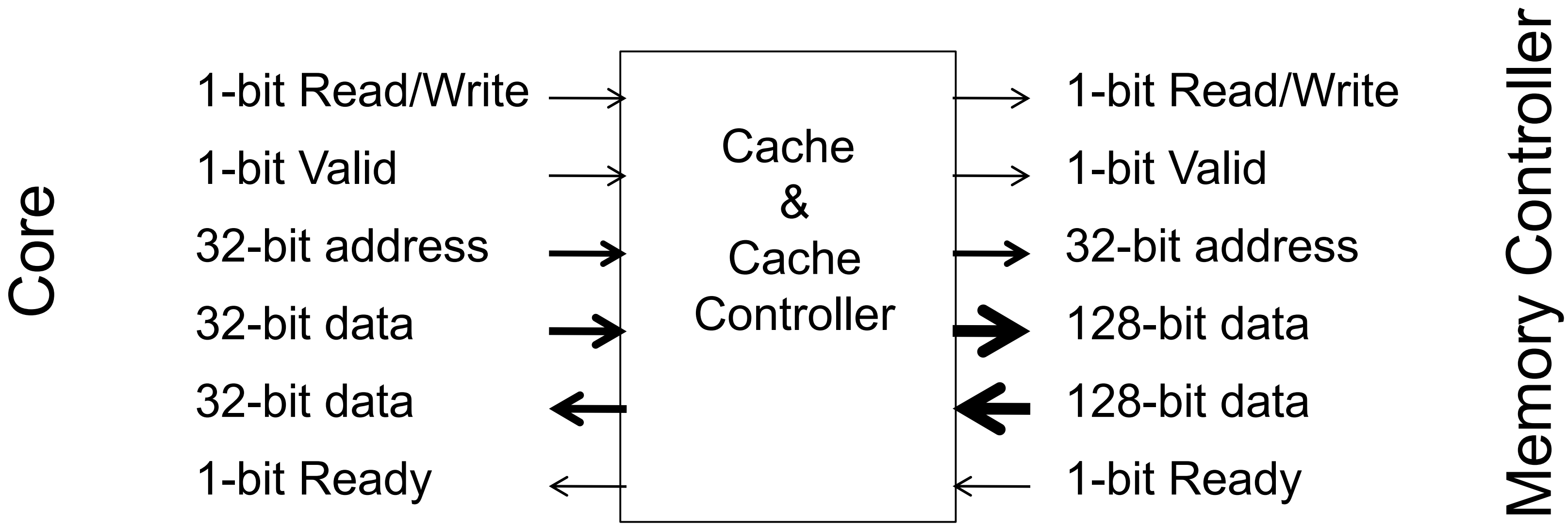
Improving Cache Performance #3

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

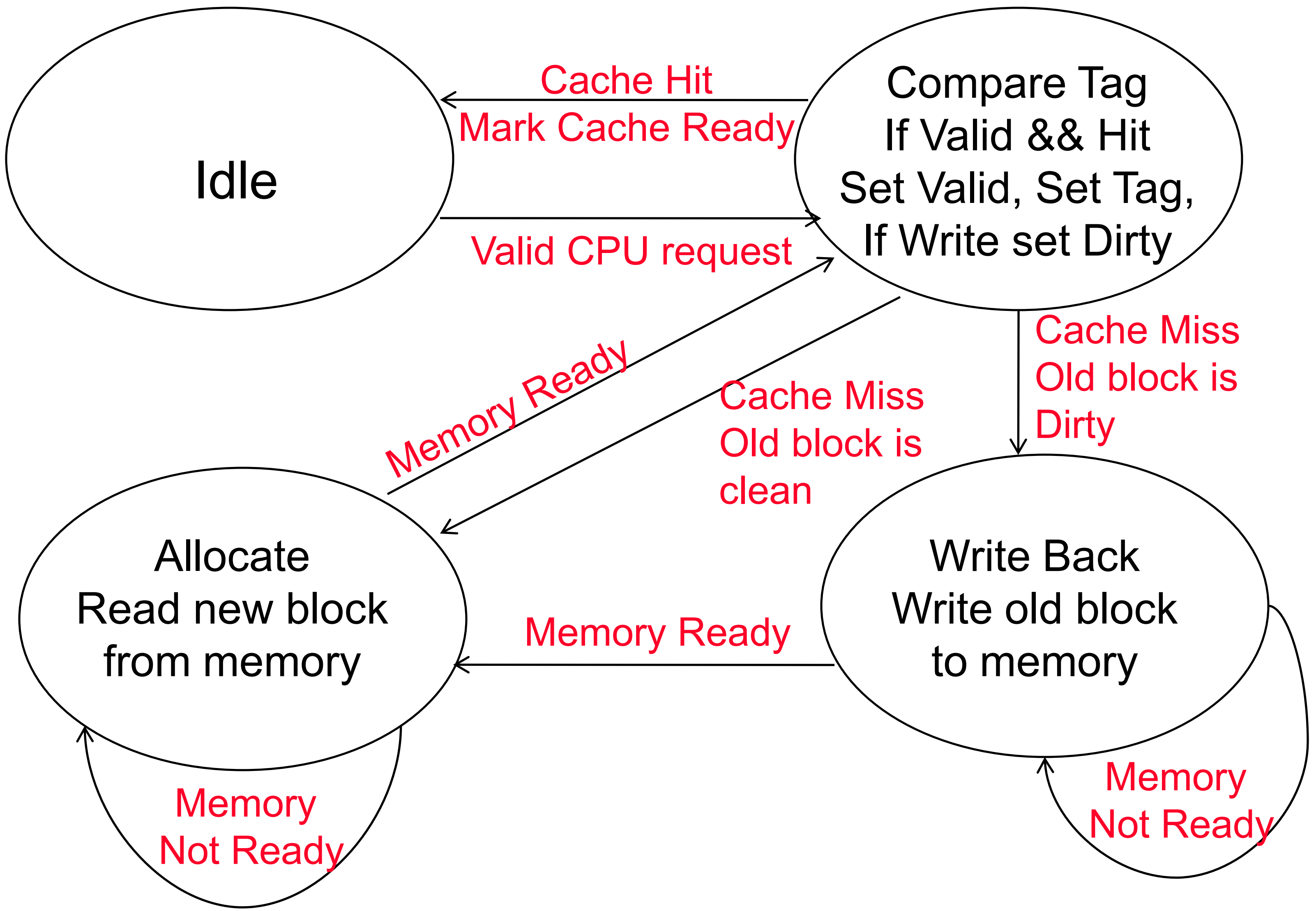
- A simple case is to use next block prefetching
  - Miss on address X → anticipate next reference miss on X + block-size
  - Works well for instr's (sequential execution) and for arrays of data
- Need to initiate prefetches sufficiently in advance
- If prefetch instr/data that is not going to be used then have polluted the cache with unnecessary data (possibly evicting useful data)

FSM Cache Controller

- Key characteristics for a simple L1 cache
  - Direct mapped
  - Write-back using write-allocate
  - Block size of 4 32-bit words (so 16B); Cache size of 16KB (so 1024 blocks)
  - 18-bit tags, 10-bit index, 2-bit block offset, 2-bit byte offset, dirty bit, valid bit, LRU bits (if set associative)



Four State Cache Controller



Cache Coherence Issues – More Details to Come

- 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 **<u>all</u>** 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

Impact of Code Transformations on Cache Performance

- Code transformations change data access pattern, influencing cache hits and misses
- A large body of compiler transformations designed to maximize cache performance
  - **Data Reuse => Data Locality => Cache Performance**
- Examples
  - Loop interchange
  - Iteration space tiling (aka code blocking)
  - Loop fusion
  - 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)
```

⇒

```
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) …
```

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

assuming ROW-MAJOR memory layout

- What type of locality do we exploit?

Restructuring Data Layout (I)

```
struct Node {
    struct Node* node;
    int key;
    char [256] name;
    char [256] school;
}

while (node) {
     if (node→key == input-key) {
          // access other fields of node
     }
     node = node→next;
}
```

- Pointer based traversal (e.g., of a linked list)
- Assume a huge linked list (1M nodes) and unique keys
- Why does the code on the left have poor cache hit rate?
  - "Other fields" occupy most of the cache line even though rarely accessed!

Restructuring Data Layout (II)

```
struct Node {
    struct Node* node;
    int key;
    struct Node-data* node-data;
}

struct Node-data {
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access node→node-data
    }
    node = node→next;
}
```

- Idea: separate frequently-used fields of a data structure and pack them into a separate data structure

- Who should do this?
  - Programmer
  - Compiler
    - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently used?

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

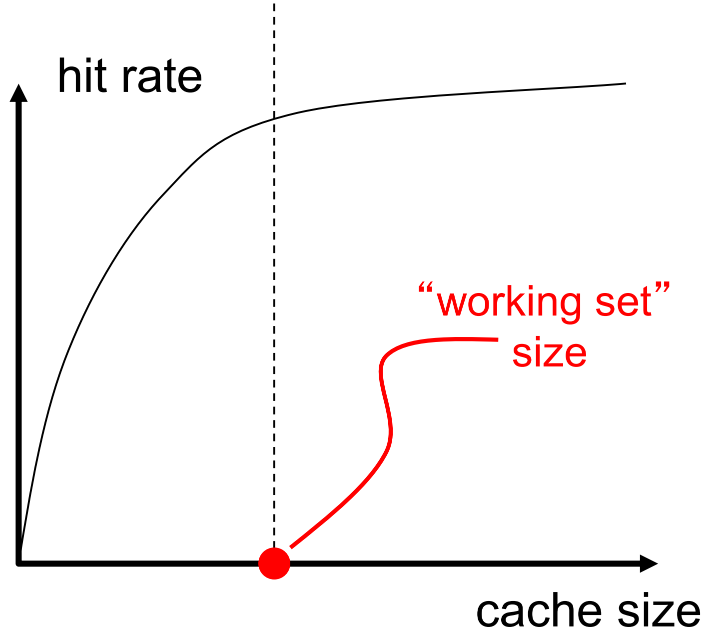
- smaller blocks
- use a write buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
- check write buffer (and/or victim cache) on read miss – may get lucky
- for large blocks fetch critical word first
- use multiple cache levels – L2 cache not tied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

Cache Performance Summary: Cache Parameters vs. Miss Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy

Cache Size

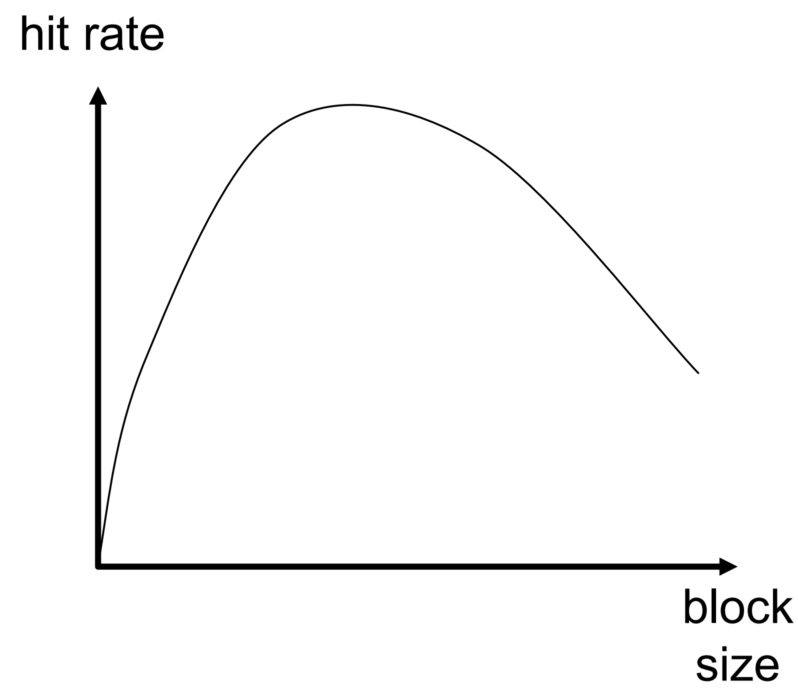
- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often



- Working set: the whole set of data the executing application references
  - Within a time interval

Block Size

- Block size is the data that is associated with an address tag
  - not necessarily the unit of transfer between hierarchies
    - Sub-blocking: A block divided into multiple pieces (each with V bit)
      - Can improve "write" performance



- Too small blocks
  - don't exploit spatial locality well
  - have larger tag overhead
- Too large blocks
  - too few total # of blocks
    - likely-useless data transferred
    - Extra bandwidth/energy consumed

Large Blocks: Critical-Word and Subblocking

- Large cache blocks can take a long time to fill into the cache
  - fill cache line critical word first
  - restart cache access before complete fill

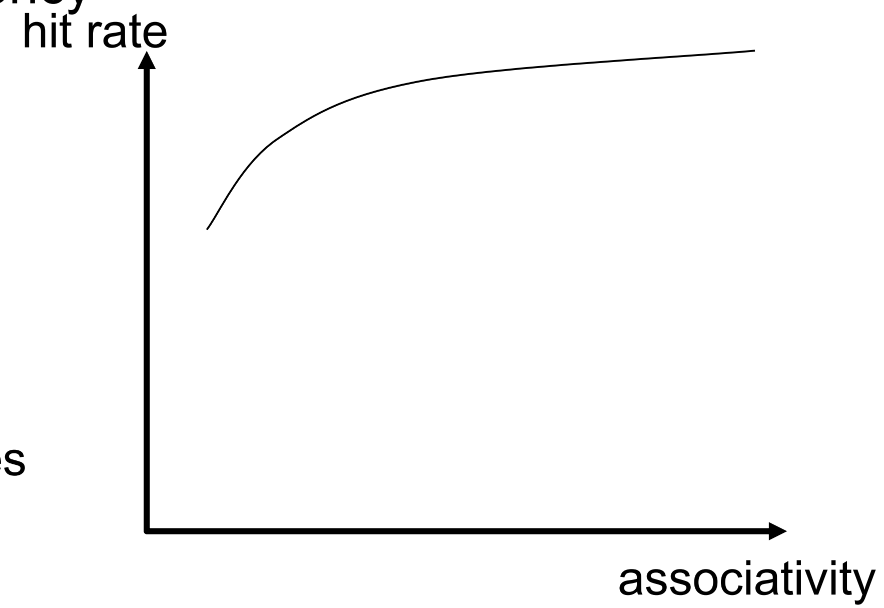
- Large cache blocks can waste bus bandwidth
  - divide a block into subblocks
  - associate separate valid bits for each subblock
  - When is this useful?

v	d	subblock	v	d	subblock	• • • •	v	d	subblock	tag

Associativity

- How many blocks can map to the same index (or set)?

- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns, higher hit latency



- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches

- Power of 2 associativity?