ECE 454
Computer Systems Programming

Memory Hierarchy

Ashvin Goel, Ding Yuan
ECE Dept, University of Toronto

Content

- Cache basics and organization
- Understanding/Profiling Memory
- Optimizing for caches (later)
  - Loop reordering
  - Tiling/blocking

Matrix Multiply

```
double a[4][4];
double b[4][4];
double c[4][4]; // assume already set to zero

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         for (k = 0; k < n; k++)
           // actual work
           c[i][j] += a[i][k] * b[k][j];
}
```

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

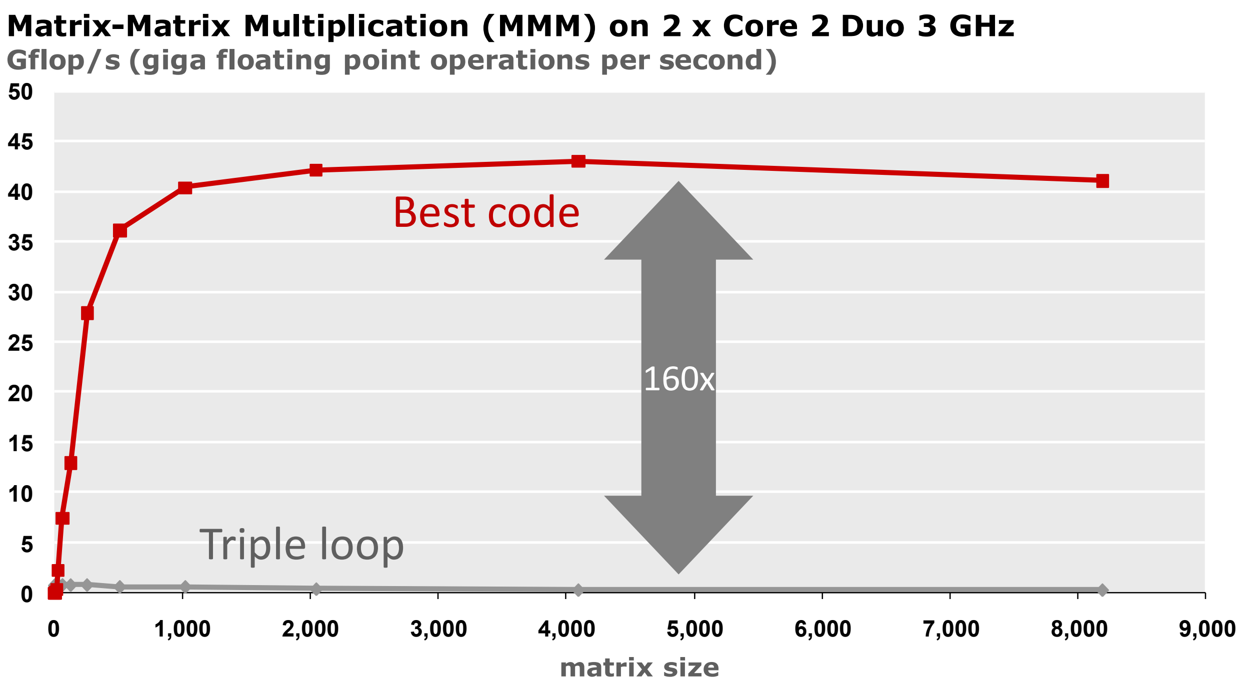
B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
c[0][0] = 1 * 17 +
          2 * 21 +
          3 * 25 +
          4 * 29
```

How much performance improvement can we get by optimizing this code?

MMM Performance

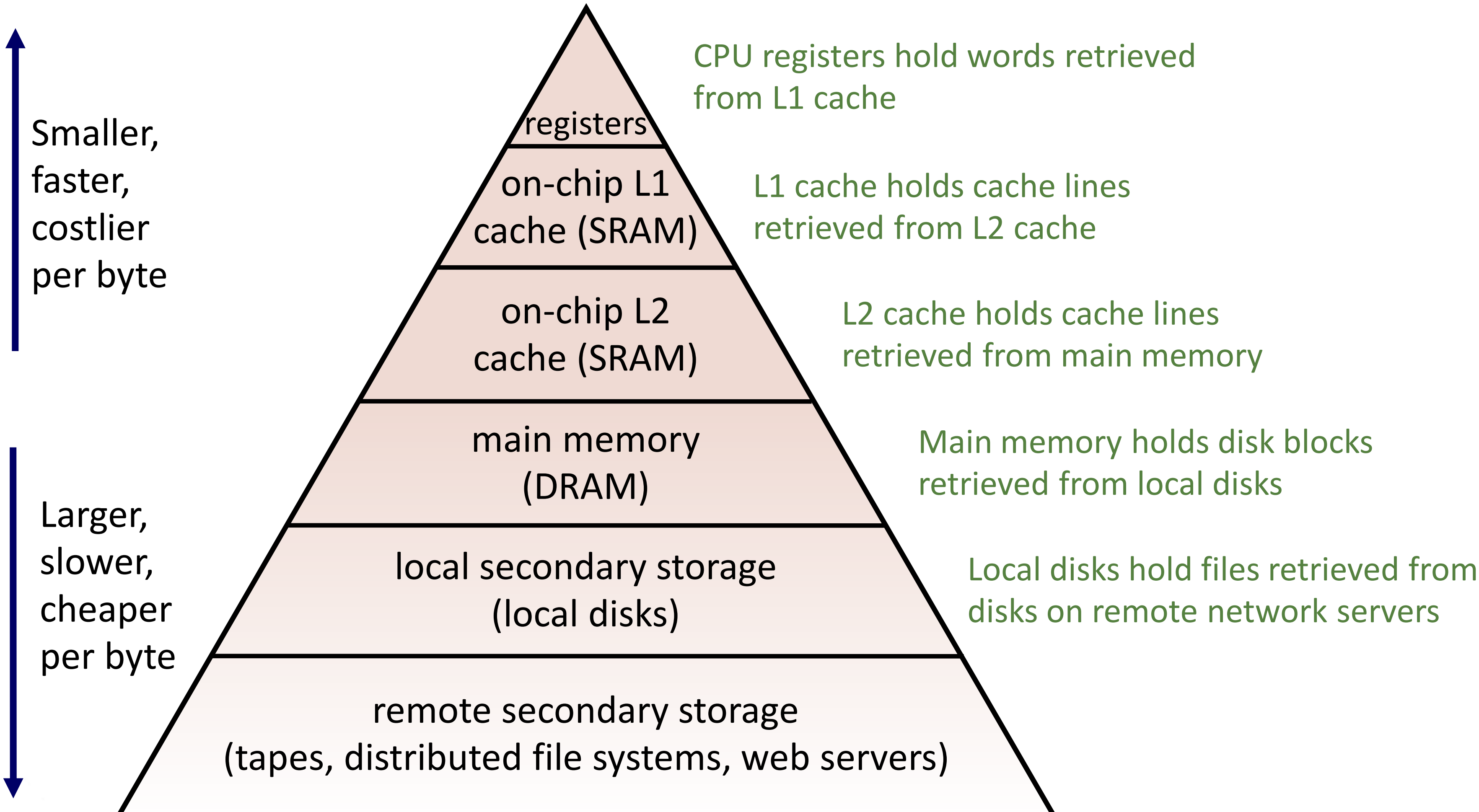


- Standard desktop computer
- Both versions compiled using optimization flags
- Both implementations have exactly the same # of operations ($2n^3$)
- What is going on?

Problem: Processor-Memory Bottleneck

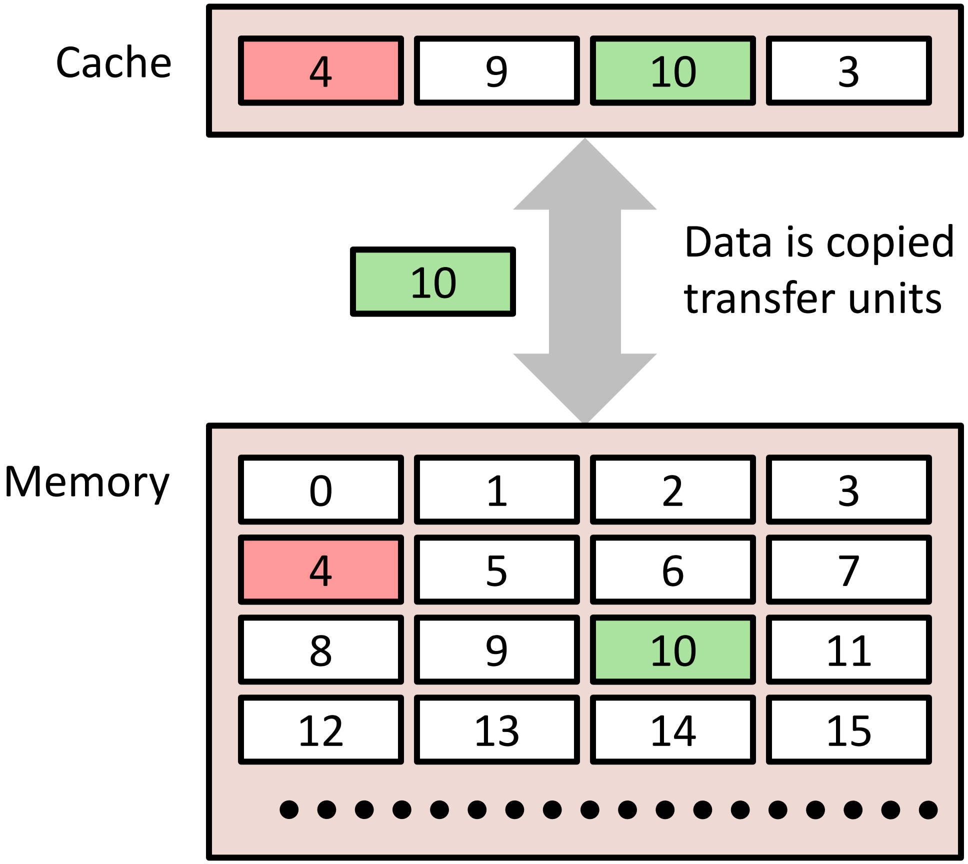
- L1 cache reference time = 1-4 ns
  - However, L1 cache size <= 64 KB
- Main memory reference time = 100 ns, 100X slower!
  - However, memory size >= GBs
- Some data:
  - 1 ns = 1/1,000,000,000 second
  - For a 2.5 GHz CPU (my laptop), 1 cycle = 0.4 ns

Memory Hierarchy



Cache Basics
(Review Hopefully!)

General Cache Mechanics

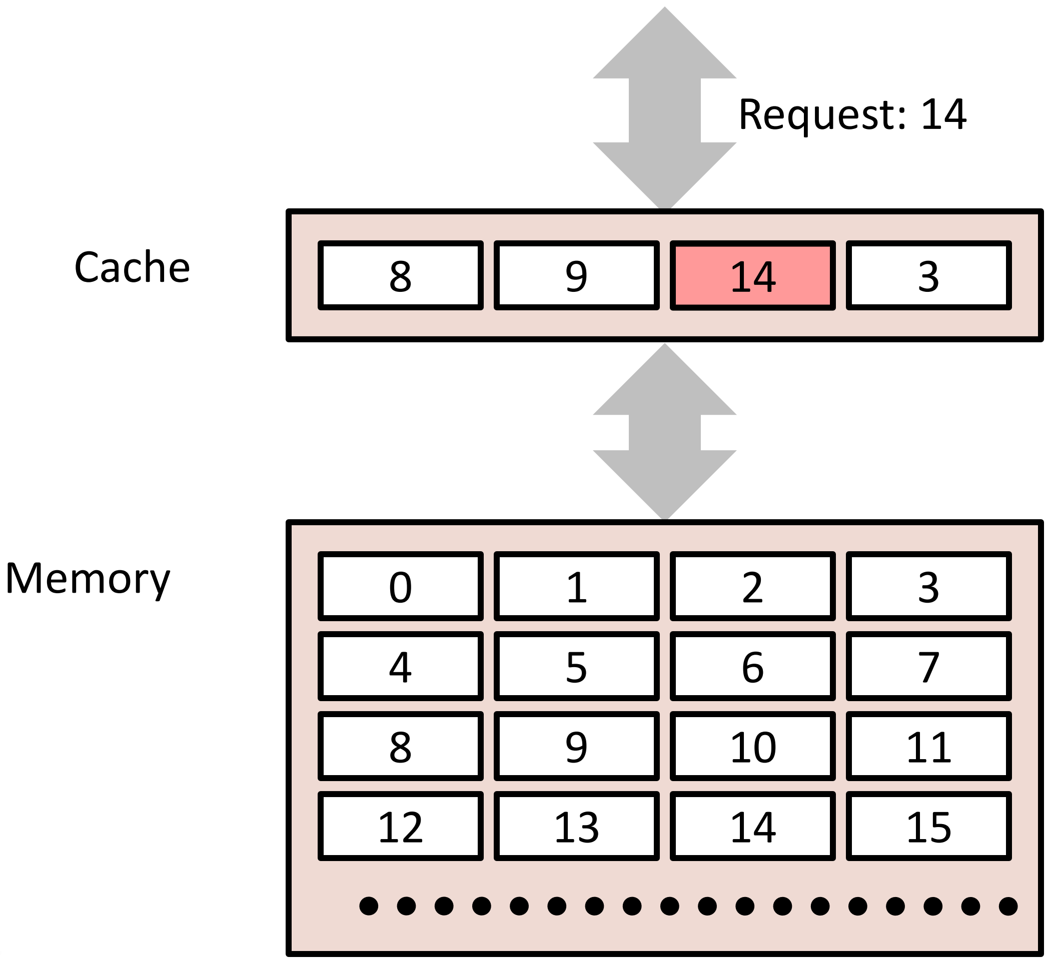


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into fixed size "blocks"

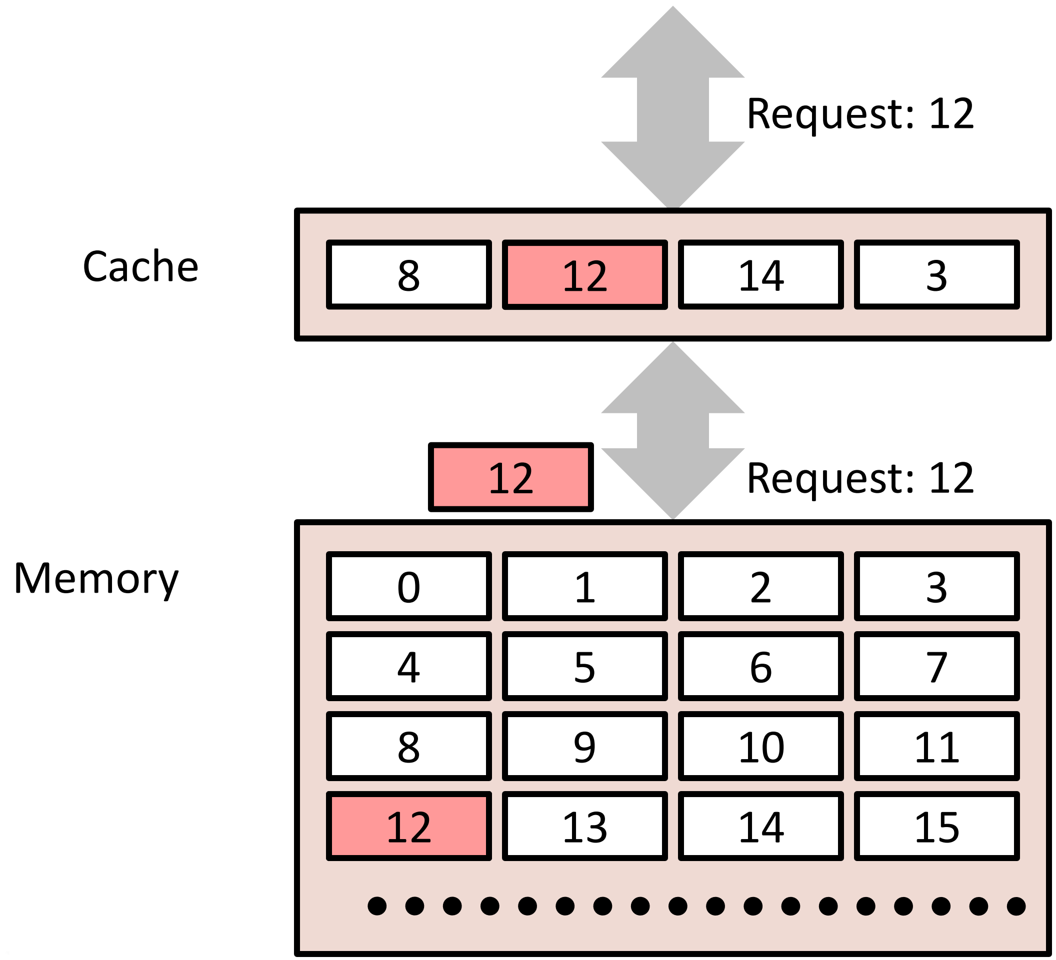
General Cache Concepts: Hit



Data in block 14 is needed

Block 14 is in cache: Hit!

General Cache Concepts: Miss



Data in block 12 is needed

Block 12 is not in cache: Miss!

Block 12 is fetched from memory

Block 12 is stored in cache:

- Placement policy: Chooses a set of blocks where 12 goes in cache
- Replacement policy: Determines which block in set gets evicted (victim)

Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache
  - miss rate = misses / accesses = 1 – hit rate
  - 3-10% for L1, small (e.g., < 1%) for L2, depending on size, etc.
- Hit Time
  - Time to deliver a line in the cache to the processor
    - Includes time to determine whether the line is in the cache
  - 1-4 clock cycles for L1, 5-20 clock cycles for L2
- Miss Penalty
  - Additional time required due to a miss
    - Typically 50-400 cycles for main memory

Let's Think About Those Numbers

- Huge difference between a hit and a miss
  - 100x between L1 and main memory
- Performance with 99% hit rate doubles compared to 97%!
  - Say cache hit time = 1 cycle, miss penalty of 100 cycles
  - Average access time:
  - 97% hits: `1 cycle + 0.03 * 100 cycles = 4 cycles`
  - 99% hits: `1 cycle + 0.01 * 100 cycles = 2 cycles`
- This is why miss (instead of hit) rate is used to think about cache performance
  - 3% is much worse than 1% miss rate

Types of Cache Misses (1)

- Three types
- Cold (compulsory) miss
  - Occurs on first access to a block
  - Can't do too much about these (except prefetching---more later)

Types of Cache Misses (2)

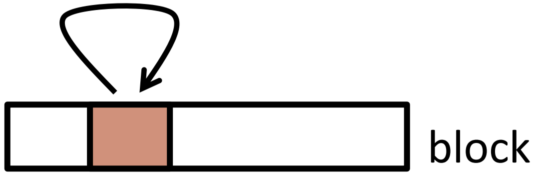
- Conflict miss
  - Placement policy of most hardware caches limit blocks to a small subset (sometimes a singleton) of the available cache slots
    - e.g., block i must be placed in slot (i mod 8)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, … would miss every time
  - Conflict misses are less of a problem today (more later)

- Capacity miss
  - Occurs when the set of active cache blocks is larger than the cache
    - Working set is larger than cache size
    - This is the most significant problem today

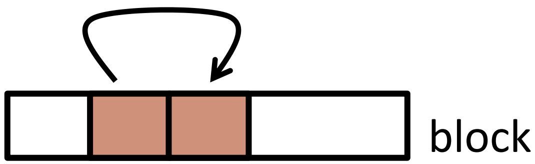
Why Caches Work

- Locality: Programs tend to use data and instructions with addresses equal or near to those they have used recently

- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time



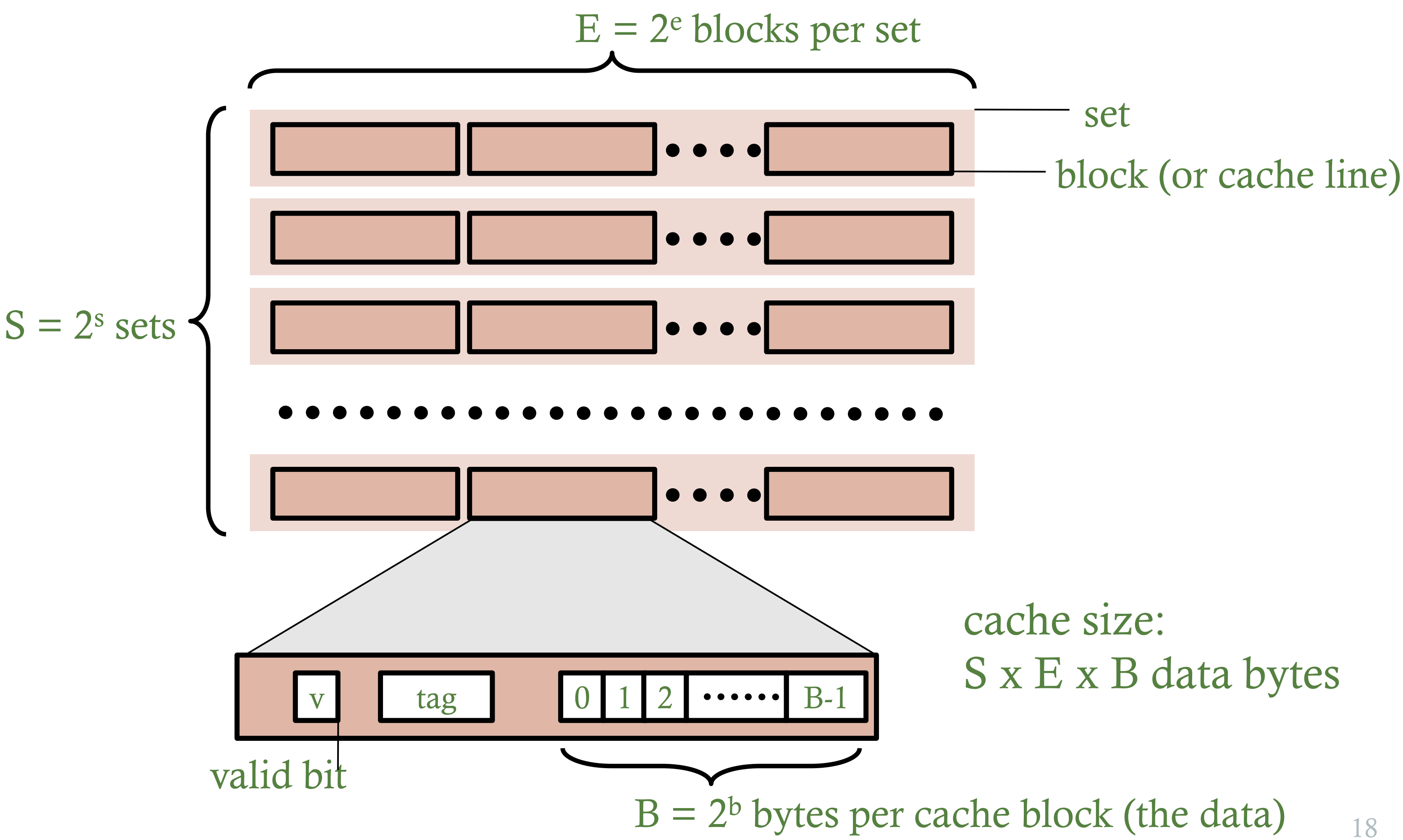
Example: Locality?

```
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```

- Data:
  - Temporal: `i, n, sum` are referenced in each iteration
  - Spatial: close by elements of array `a` accessed (in stride-1 pattern)
- Instructions:
  - Temporal: cycle through loop instructions repeatedly
  - Spatial: reference close by instructions in sequence
- Important to be able to assess the locality in your code!

Cache Organization

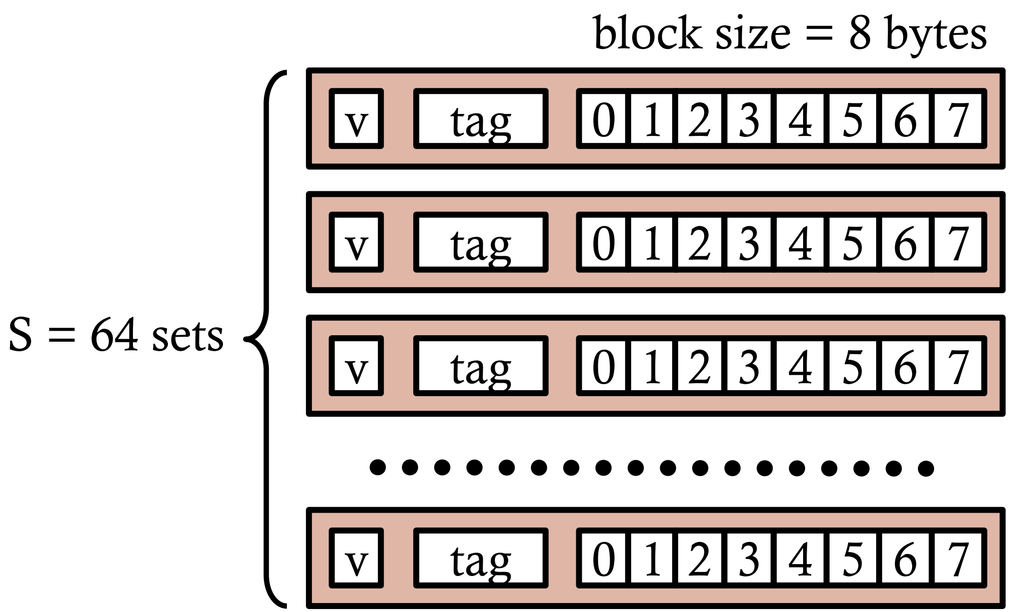
General Cache Organization (S, E, B)



cache size:
S x E x B data bytes

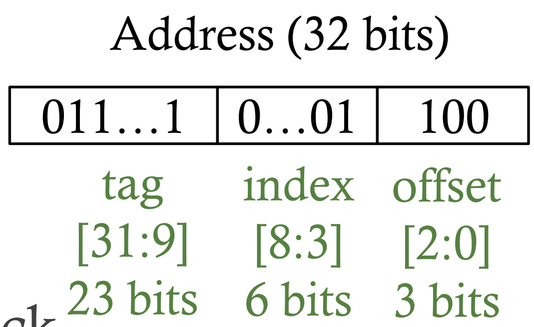
Direct Mapped Cache (E = 1)

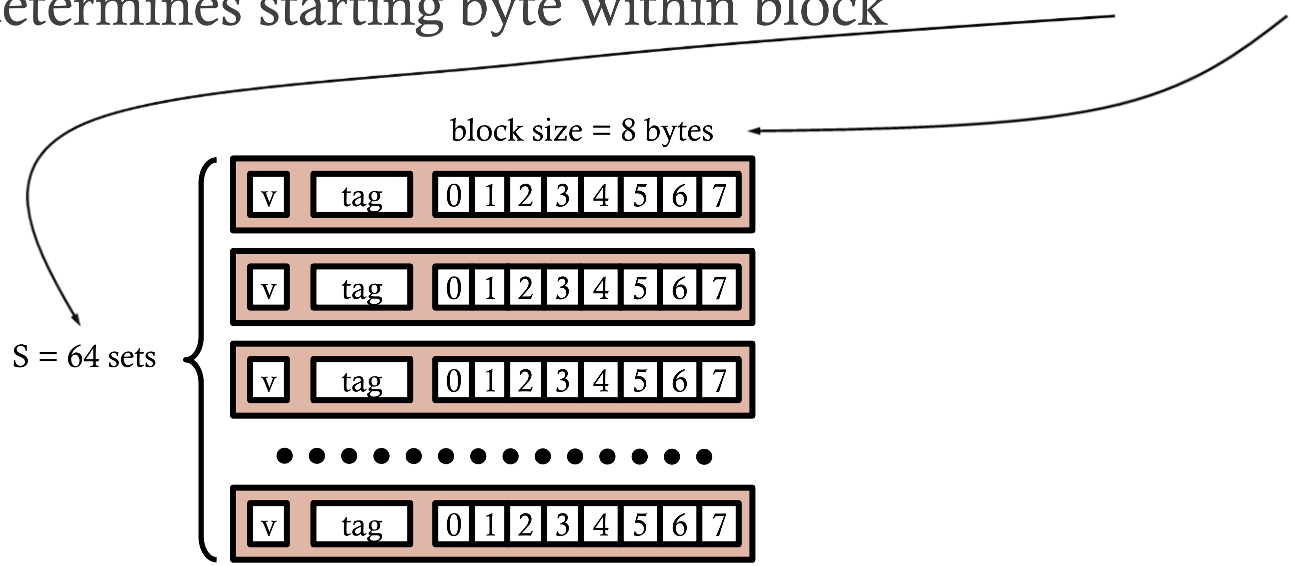
- Direct mapped: one block per set



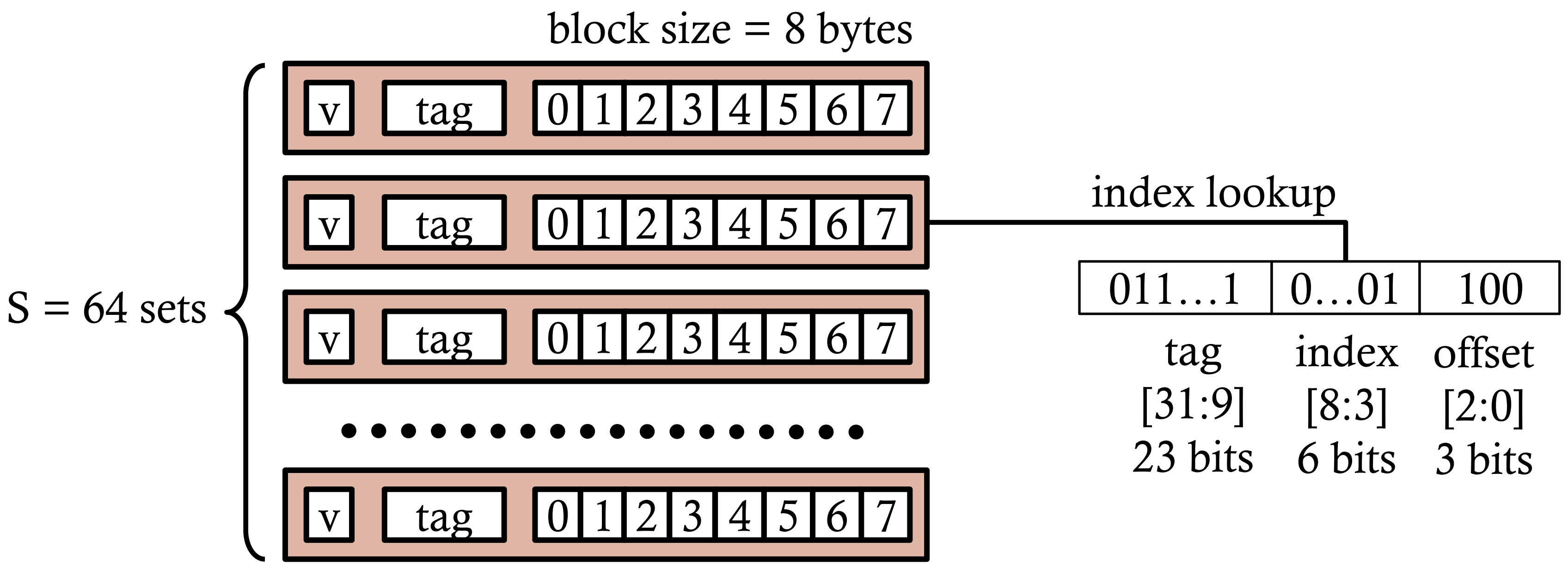
Direct Mapped Cache

- Incoming memory address divided into tag, index and offset bits
  - Index determines set
  - Tag is used for matching
  - Offset determines starting byte within block

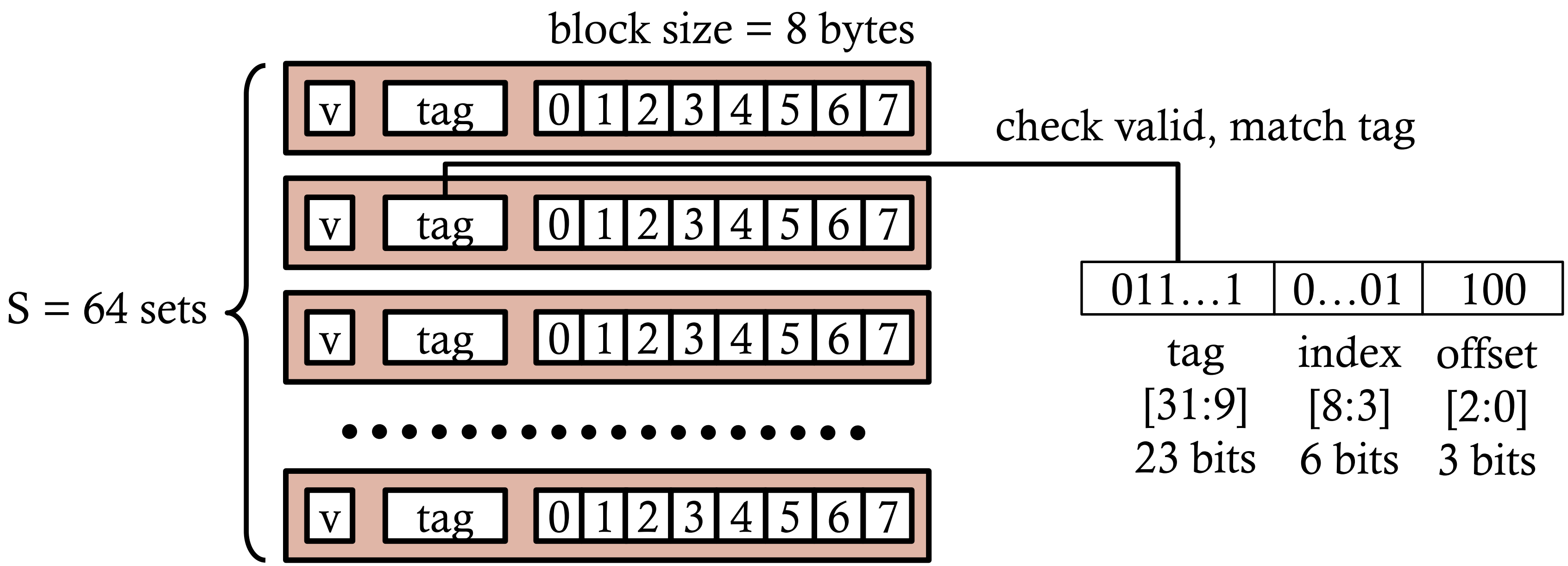




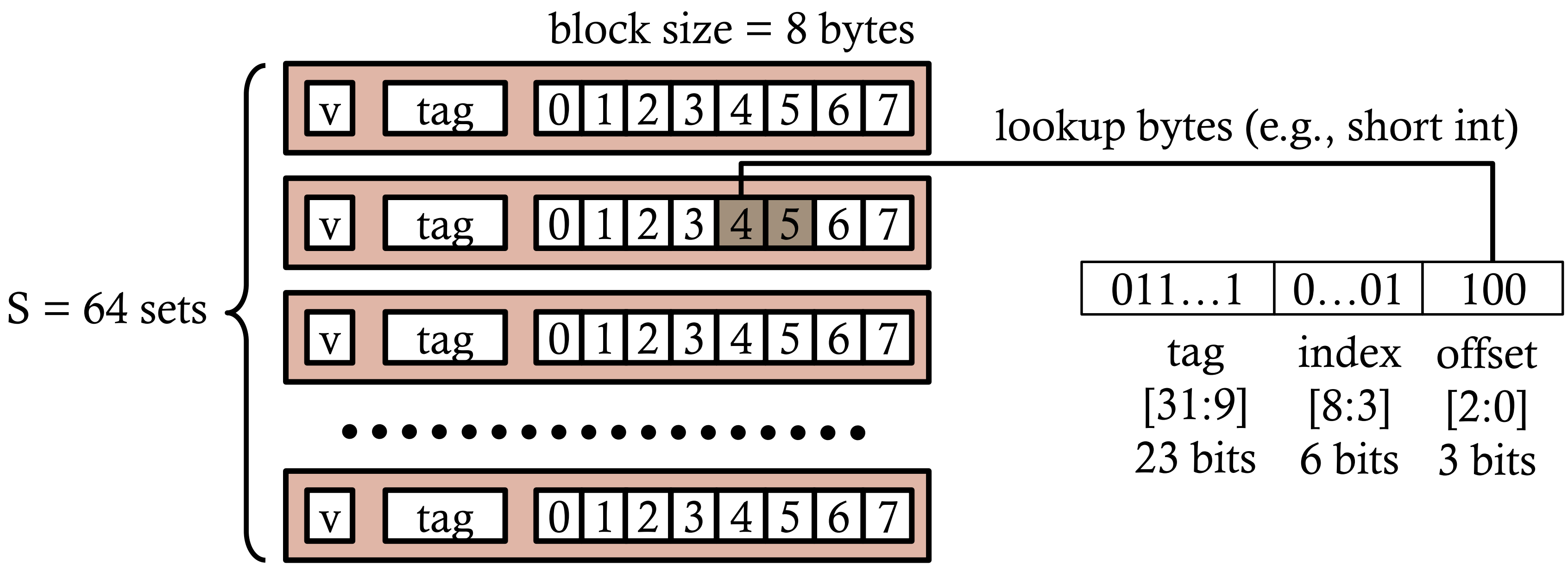
Direct Mapped Cache: Index Lookup



Direct Mapped Cache: Match Tag

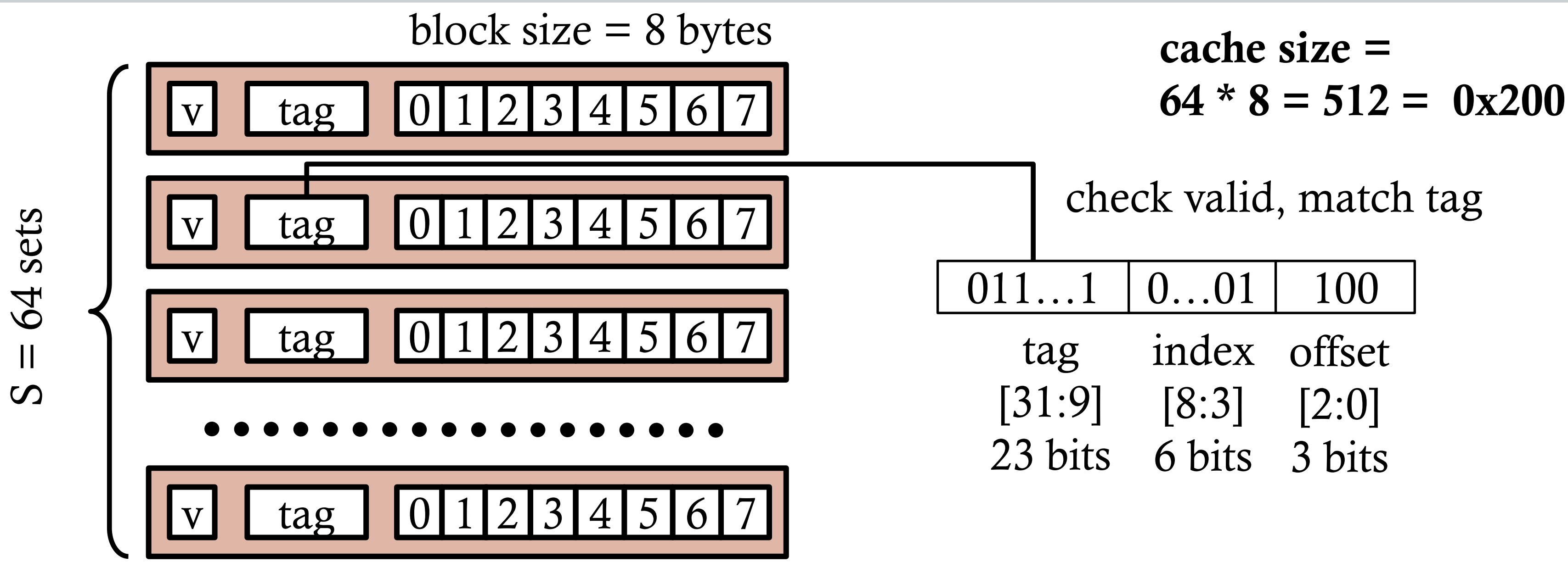


Direct Mapped Cache: Lookup Bytes



- Assume address being looked up is for a short int (2 bytes)
- If the tag doesn't match, old block is evicted and replaced with entire new block (i.e., 8 bytes are loaded from memory)

Direct Mapped Cache Example

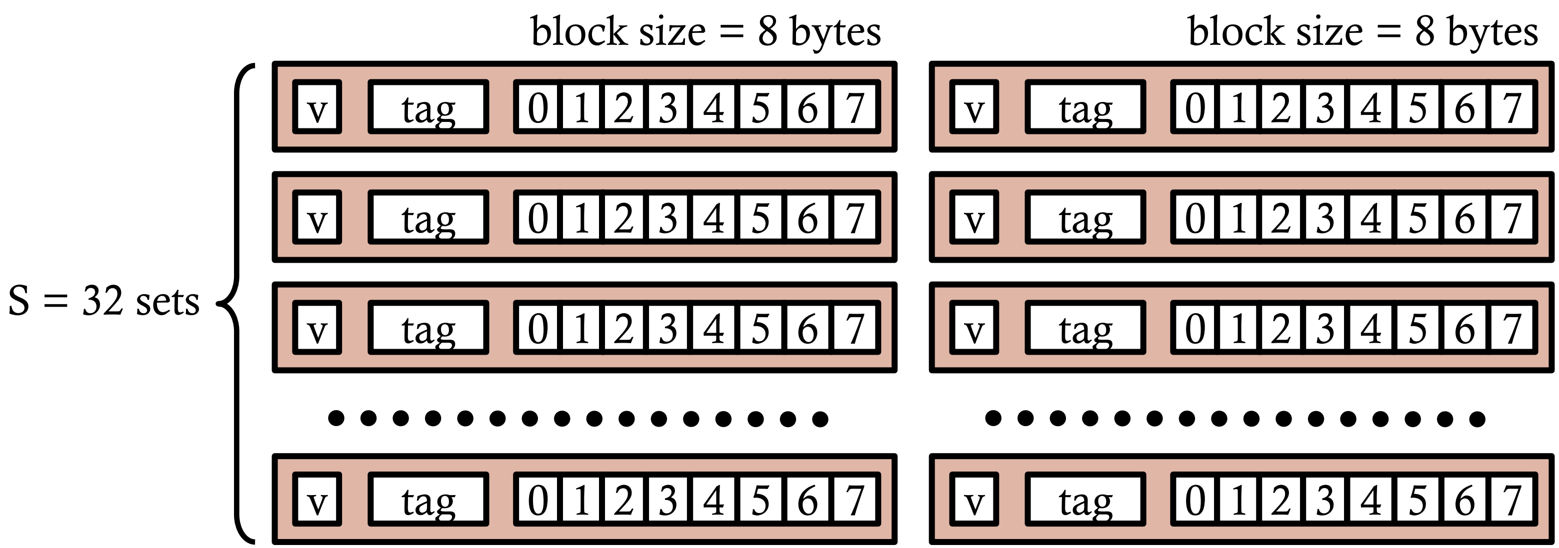



```
long a[100]; // each array element is 8 bytes

a[0]:  Addr 0x0   =            0b0 000 = (0,  0, 0)  maps to Set 0
a[1]:  Addr 0x8   =            0b1 000 = (0,  1, 0)  maps to Set 1
a[32]: Addr 0x100 = 0b0 100000 000 = (0, 32, 0) maps to Set 32
a[64]: Addr 0x200 = 0b1 000000 000 = (1,  0, 0)  maps to Set 0
```

Two-way Set Associative Cache (E = 2)

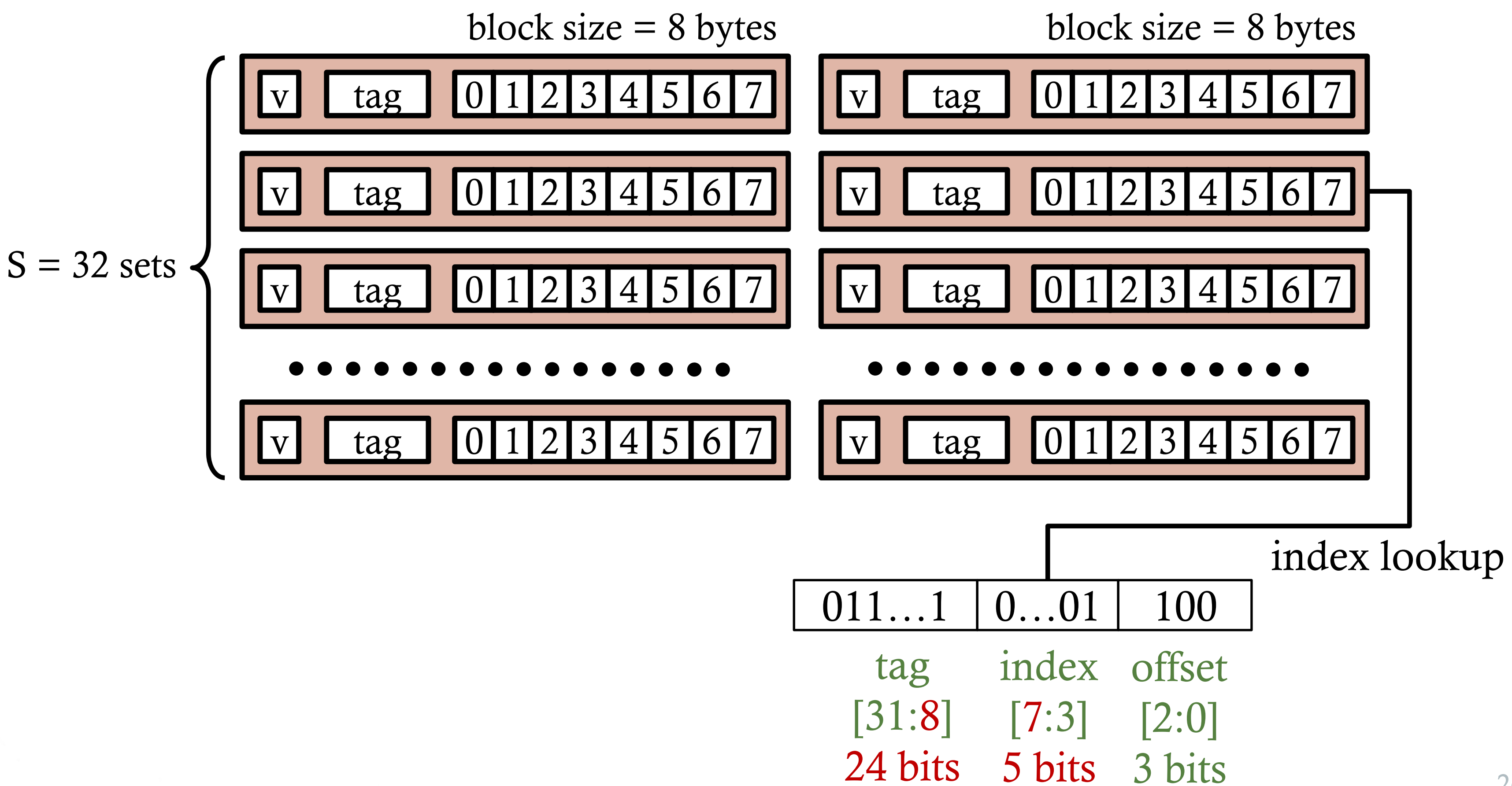
- 2-way set associative: two blocks per set



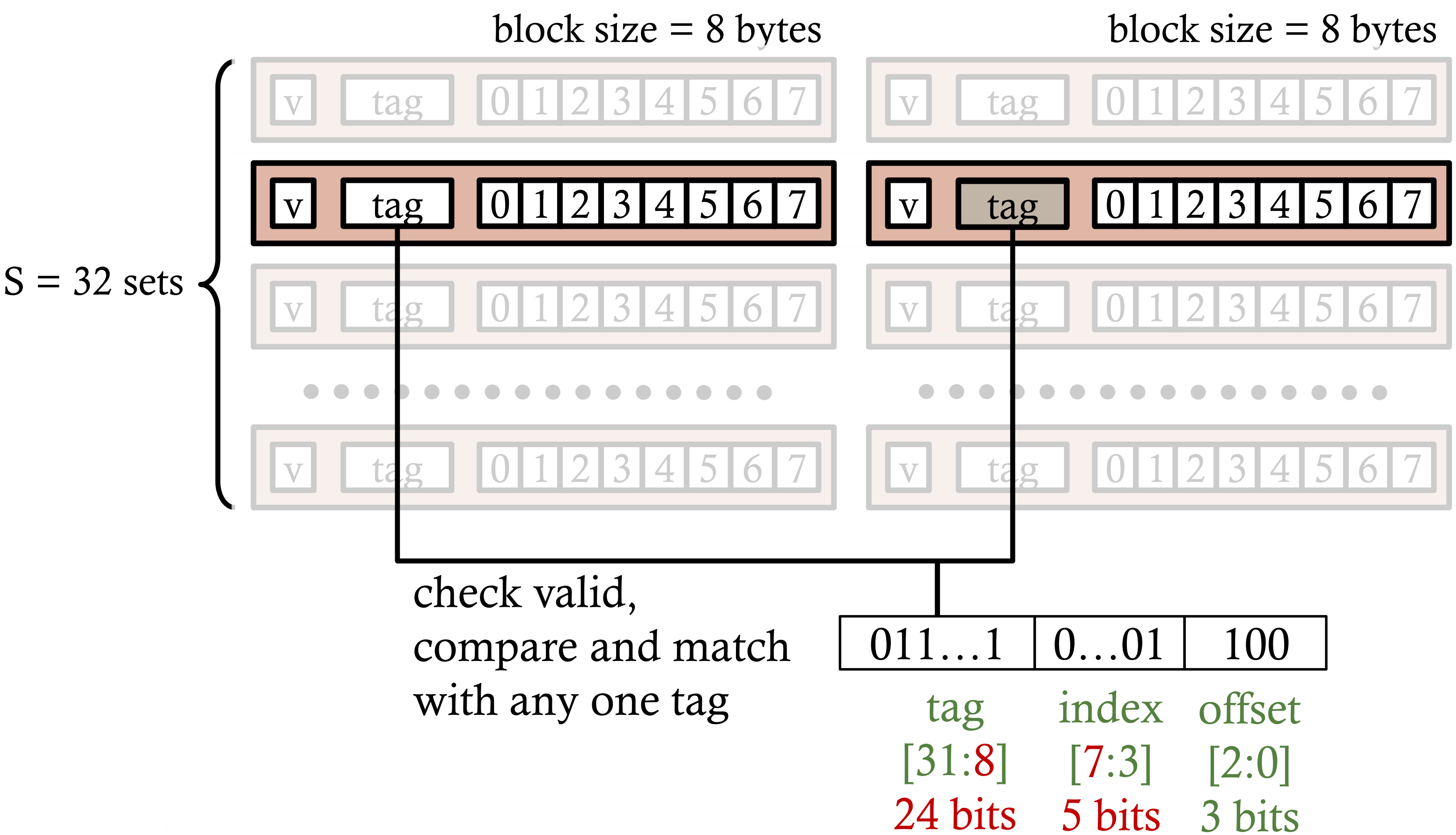
- Total cache size is same as direct mapped cache
- But number of sets is halved

011…1	0…01	100
tag	index	offset
[31:8]	[7:3]	[2:0]
24 bits	5 bits	3 bits

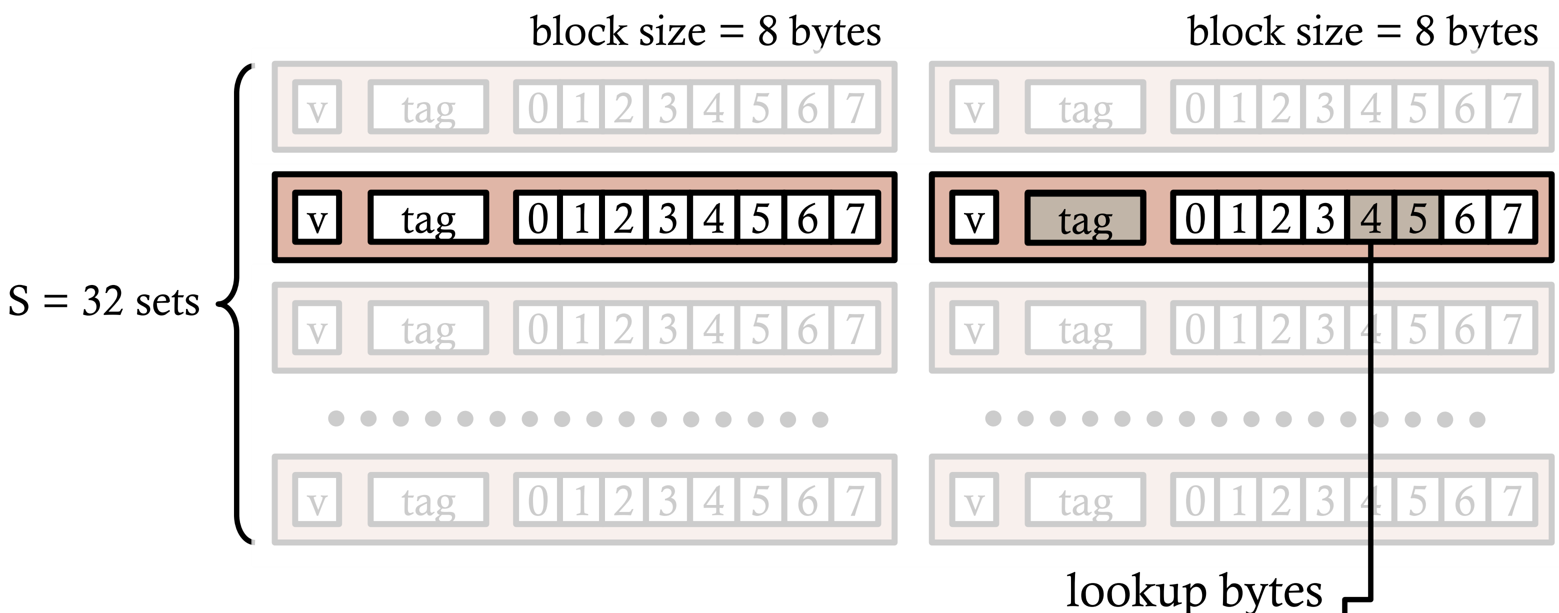
Two-way Set Associative Cache (E = 2)

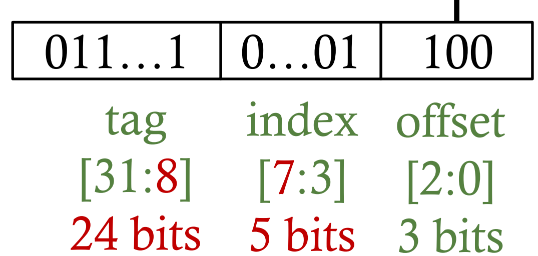


Two-way Set Associative Cache (E = 2)



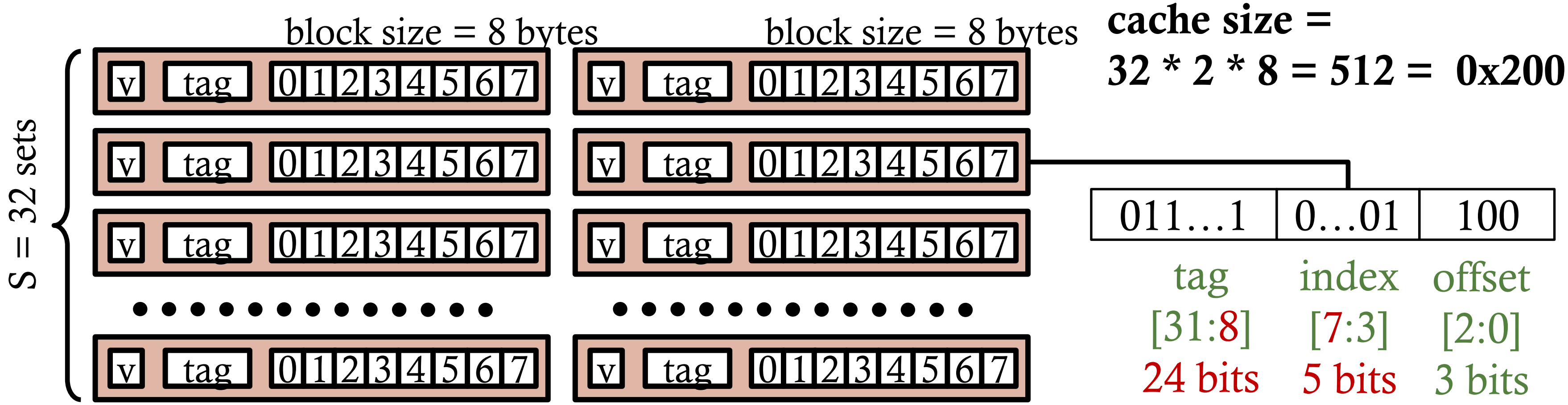
Two-way Set Associative Cache (E = 2)





- If no match then one line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

Two-way Cache Example



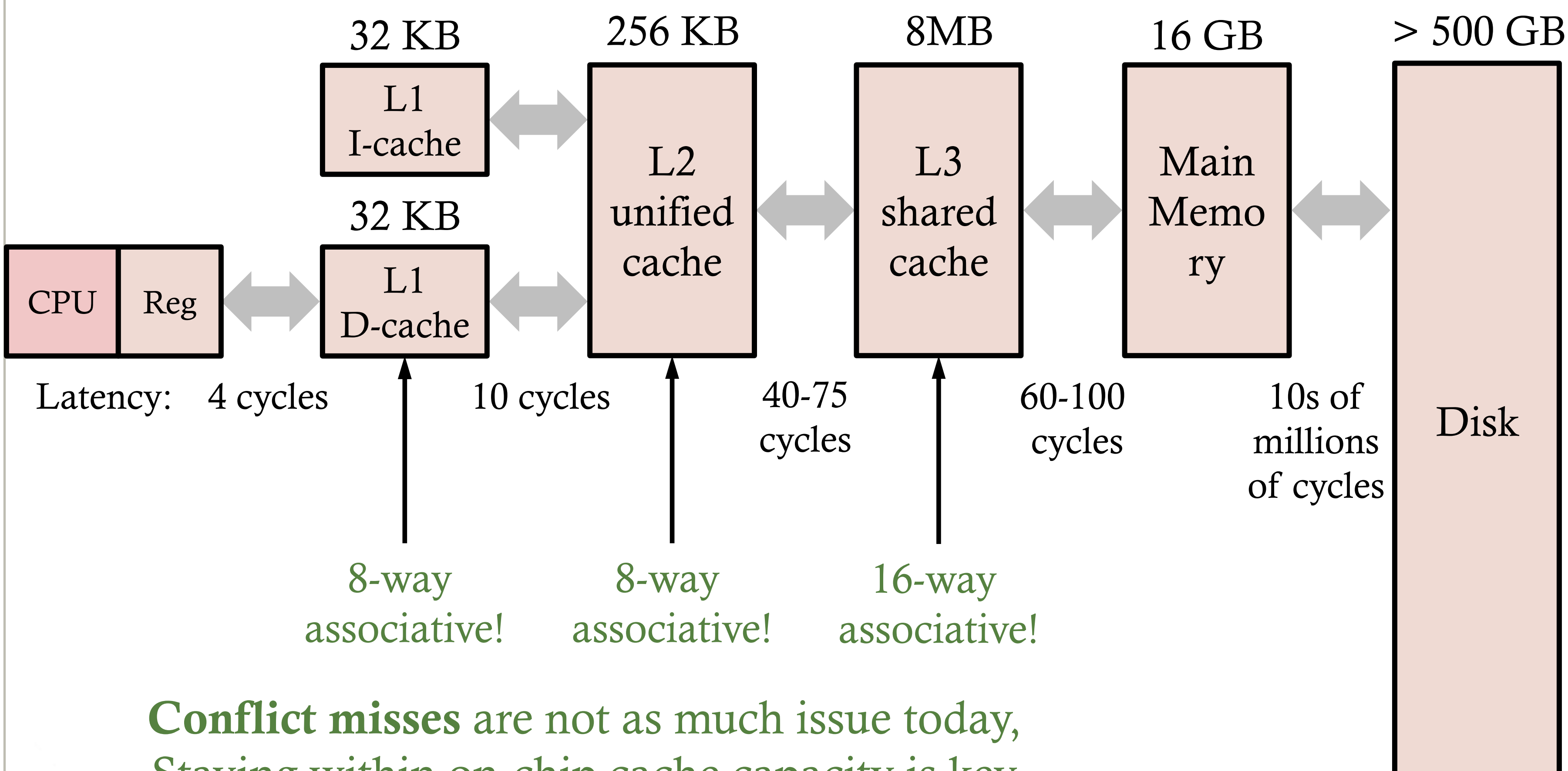
cache size =
32 * 2 * 8 = 512 = 0x200

```
long a[100]; // each array element is 8 bytes

a[0]:  Addr 0x0   =          0b0 000 = (0,  0, 0)  maps to Set 0
a[1]:  Addr 0x8   =          0b1 000 = (0,  1, 0)  maps to Set 1
a[32]: Addr 0x100 = 0b1  00000 000 = (1,  0, 0)  maps to Set 0
a[64]: Addr 0x200 = 0b10 00000 000 = (1,  0, 0)  maps to Set 0
```

Intel Core i7: Cache Associativity

L1/L2 cache have 64 B blocks



Conflict misses are not as much issue today,
Staying within on-chip cache capacity is key

What About Writes?

- Multiple copies of data exist in L1, L2, main memory, disk
  - Need to ensure consistency
- What to do on a write-hit?
  - Write-through (write to cache and immediately to memory)
  - Write-back (defer write to memory until line is replaced)
    - Need a dirty bit (cache line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more reads and writes to the location follow
  - No-write-allocate (write immediately to memory)
    - For streaming writes (write once and then no reads in the near future)

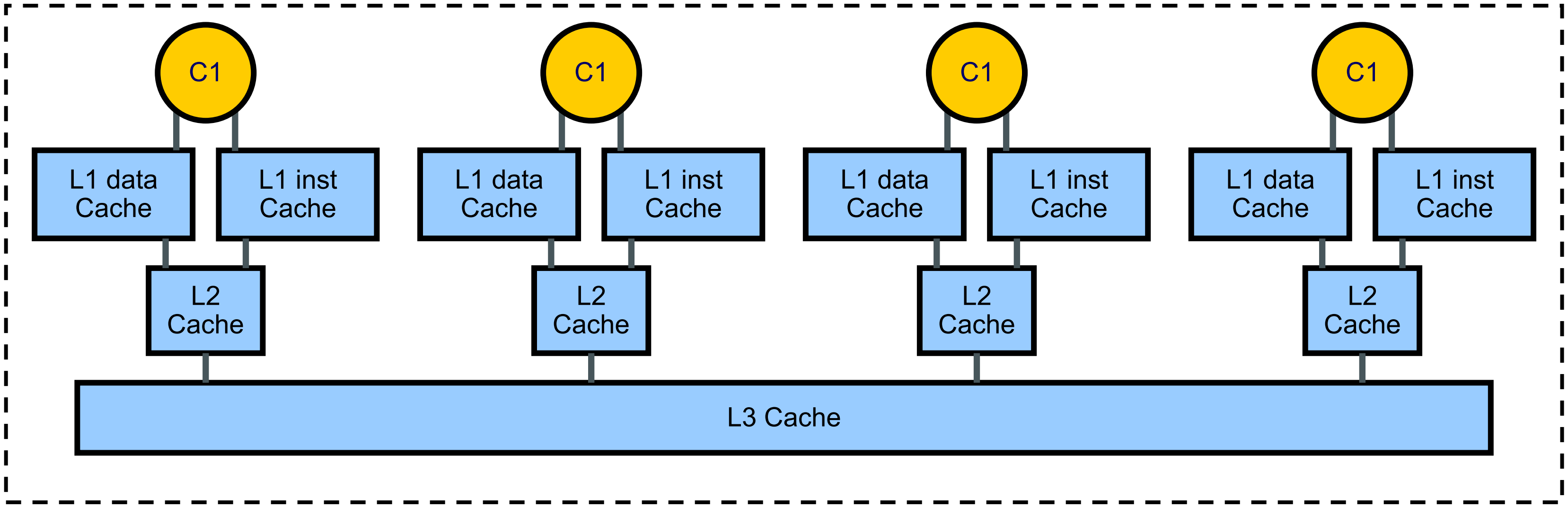
What About Writes?

- Multiple copies of data exist in L1, L2, main memory, disk
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (cache line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more reads and writes to the location follow
  - No-write-allocate (write immediately to memory)
    - For streaming writes (write once and then no reads in the near future)
- Typically:
  - Write-through + No-write-allocate
  - Write-back + Write-allocate

Understanding/Profiling Memory

UG Machines

Processor Chip



1 CPU – Intel Core i7-4790, 3.6 GHz, with 4 HT cores

Run `lscpu` on UG machine shows:

32KB, 8-way L1 data cache
32KB, 8-way L1 inst cache

256KB, 8-way L2 cache
8M, 16-way L3 cache

Get Memory Hierarchy Details: `lstopo`

- Running `lstopo` on UG machine shows:

Shared L3

```
Machine (16GB)
  Package L#0 + L3 L#0 (8192KB)
    L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
    L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#1)
    L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2 + PU L#2 (P#2)
    L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3 + PU L#3 (P#3)
```

4 cores per CPU

Get More Cache Details: L1 dcache

- ls /sys/devices/system/cpu/cpu0/cache/index0

  coherency_line_size: 64 // 64B cache lines

  level: 1 // L1 cache

  number_of_sets: 64

  physical_line_partition

  shared_cpu_list: 0 // shared by cpu0 only

  shared_cpu_map

  size: 32K

  type: data // data cache

  ways_of_associativity: 8 // 8-way set associative

Get More Cache Details: L2

- ls /sys/devices/system/cpu/cpu0/cache/index2

  coherency_line_size: 64 // 64B cache lines

  level: 2 // L2 cache

  number_of_sets: 512

  physical_line_partition

  shared_cpu_list

  shared_cpu_map

  size: 256K

  type: Unified // unified cache, means instructions and data

  ways_of_associativity: 8 // 8-way set associative

Access Hardware Counters: perf

- The `perf` tool allows you to access performance counters
  -
- To measure L1 data cache load misses for program `pi`, run:

```
perf stat -e L1-dcache-load-misses pi

7803  L1-dcache-load-misses    # 0.000 M/sec
```

- To see a list of all events you can measure:

```
perf list
```

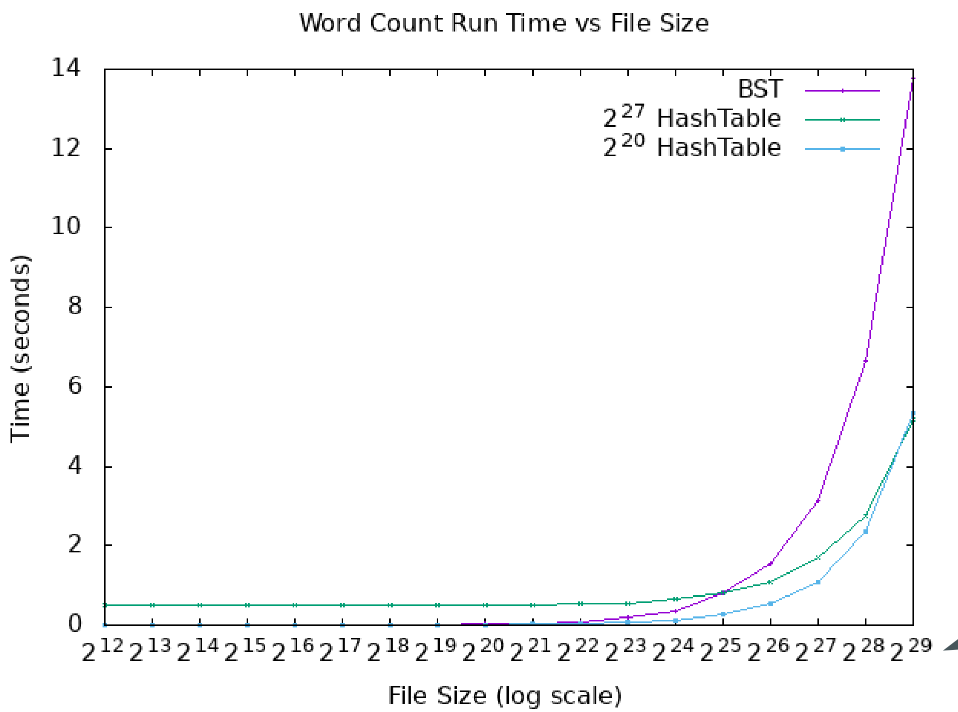
- Note: you can measure multiple events at once

BST and Hash Table Comparison

- A binary search tree (BST) and a hash table store 100 million items (~2^26)
- How much faster will a hash table be versus BST?
  - BST: Number of pointer traversals: $\log(2^{26}) = \sim 26$
    - However, the number is smaller for internal nodes
  - Hash Table: Number of pointer traversals: ~2
    - One to access hash table entry
    - One to access data item
  - Based on running code, BST traverses pointers 8 times more than hash table (expected is 26/2 = 13)
  - So we expect hash table to be roughly 8 times faster …

BST and Hash Table Comparison

- A binary search tree (BST) and a hash table store 100 million items (~2^26)
- How much faster will a hash table be versus BST?



Average word len ~= 5,
when file size = 2^{29} (~500M),
number of unique words is ~100M

Analysis

- While the hash table beats BST, the performance improvement is not what is expected from analysis of data accesses (8x improvement)
  - Hash table performance is **only** 2.5-2.8x better
- Why is that the case? Let's look at it in more detail.

Perf on BST and HashTable

- Initial hypothesis: memory hierarchy is the culprit, so run perf
  - E.g., Does hash table have a lot more LLC accesses than BST?

```
perf stat -e instructions -e L1-dcache-loads -e L1-dcache-misses -e LLC-loads -e LLC-misses {bst,hash}-program
```

	BST	HashTable	Ratio
Time	13.11 s	4.77 s	2.75
Instructions	50 b	21 b	2.4
L1 hits (4 cycles)	13000 m	4500 m	2.9
L1 misses (10 cycles)	1000 m	214 m	4.7
LLC loads (40-75 cycles)	333 m	101 m	3.3
LLC misses (60-100 cycles)	21 m	18 m	1.2

far lower than 8 (expected value)

Further Analysis

- Number of instructions executed and L1 cache hits is proportional to runtime
- Need to understand what instructions are being executed
  - Need to use gprof to see where time is being spent in code
  - Lesson: use code profiling before memory profiling
- Found that hash key calculation takes significant time, reducing the improvements we expect from the hash table!
  - Required disabling the inlining of this function (for gprof)!
- But really, can the hash key calculation slow down expected improvements by so much?

Digging Even Further

- Looking at assembly for each insertion of a word, the number of load/store operations is as follows:
  - BST: 31 (5 initial load/stores + 2 loads per iteration * 13 (roughly, depth of tree traversal))
  - Hash Table: 13 (10 initial load/stores (including hash key calculation) + 2 loads per iteration * 1.5 (for linked list traversal))
    - 13/1.5 ~= 8, which is the extra amount of pointer traversals that we measured that the BST code does over the hash table code
- Ratio of load/stores of BST to HashTable = 31/13 = 2.38
  - Roughly the same as observed performance
- So initialization has a significant impact on speedup!