Caches (cont.)

CSCI 237: Computer Organization
29th Lecture, Friday, November 15, 2024

Kelly Shaw

Administrative Details

- Read CSAPP 6.4-6.6
- Quiz due at 2:30pm today
- Lab #5 due Tuesday at 11pm
- **Watch short video before Monday**
- Colloquium on Friday at 2:35pm
  - Subhadeep Sarkar, Brandeis University
  - Building Deletion-Compliant Data Systems

Last Time

- Cache memory organization and operation (Ch 6.4)
- Caches in real systems
- Performance impact of caches
  - The memory mountain

Today

- Cache memory organization and operation (Ch 6.4)
- Caches in real systems
- Performance impact of caches
  - The memory mountain

What about writes?

- Challenge: Multiple copies of data exist throughout hierarchy.
  - L1, L2, L3, Main Memory, Disk
- Policy: 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” (is the line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load existing line into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- Typical
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate (most inline with current trends)**

Now that we've discussed how caches work

- How do caches exploit temporal locality?
- How do caches exploit spatial locality?
- What is the benefit of increasing set associativity?
- What is the disadvantage of increasing set associativity?

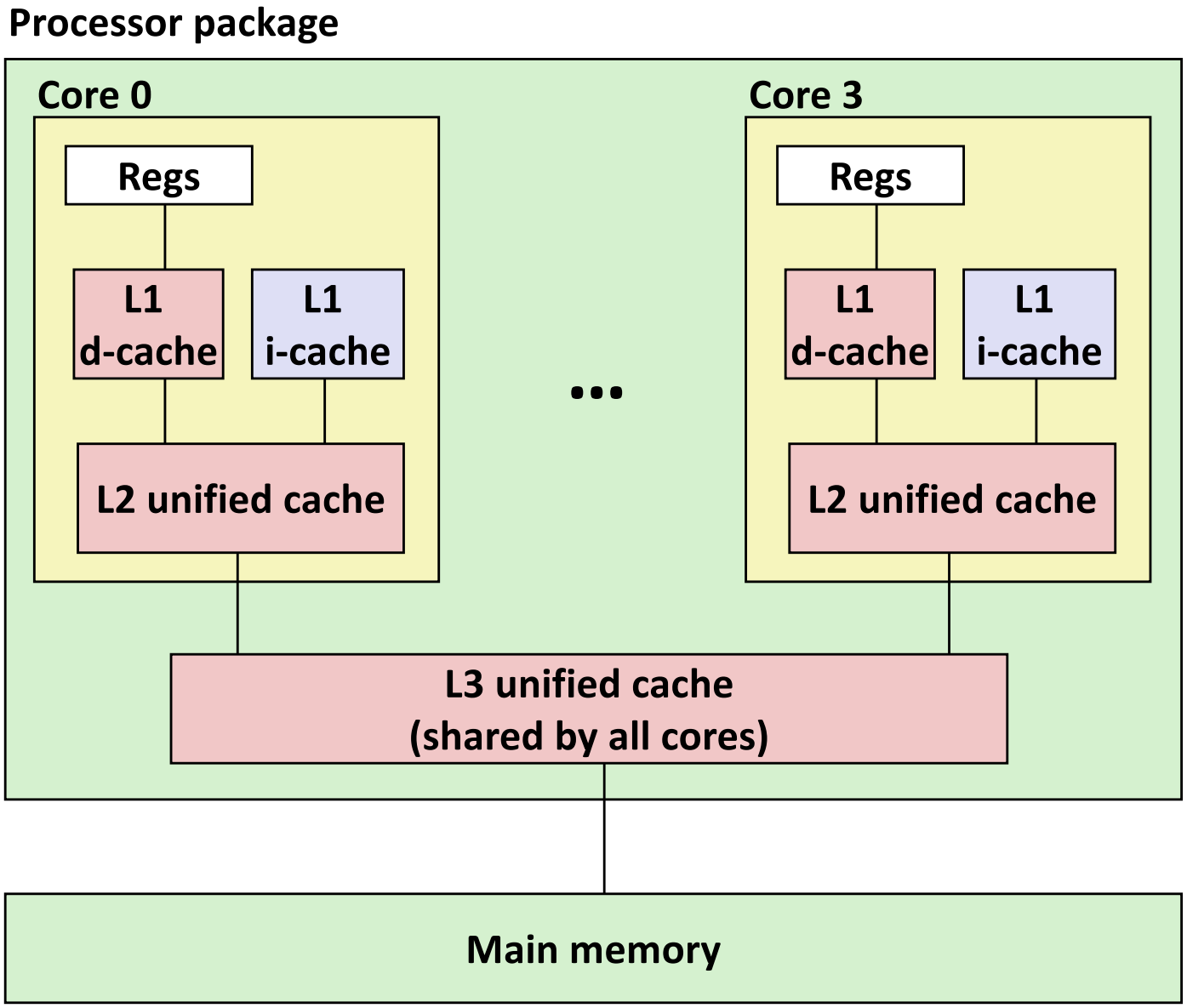
Today

- Cache memory organization and operation (Ch 6.4)
- Caches in real systems
- Practice Problem in Pairs
- Performance impact of caches
  - The memory mountain

How are real L1/L2/L3 caches organized?

- We often create separate caches for code and data
  - Why?
- CPUs with multiple cores often have private and shared caches
  - Smaller caches on each core (e.g., L1 and L2)
  - Larger caches shared by whole CPU (e.g., L3)

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

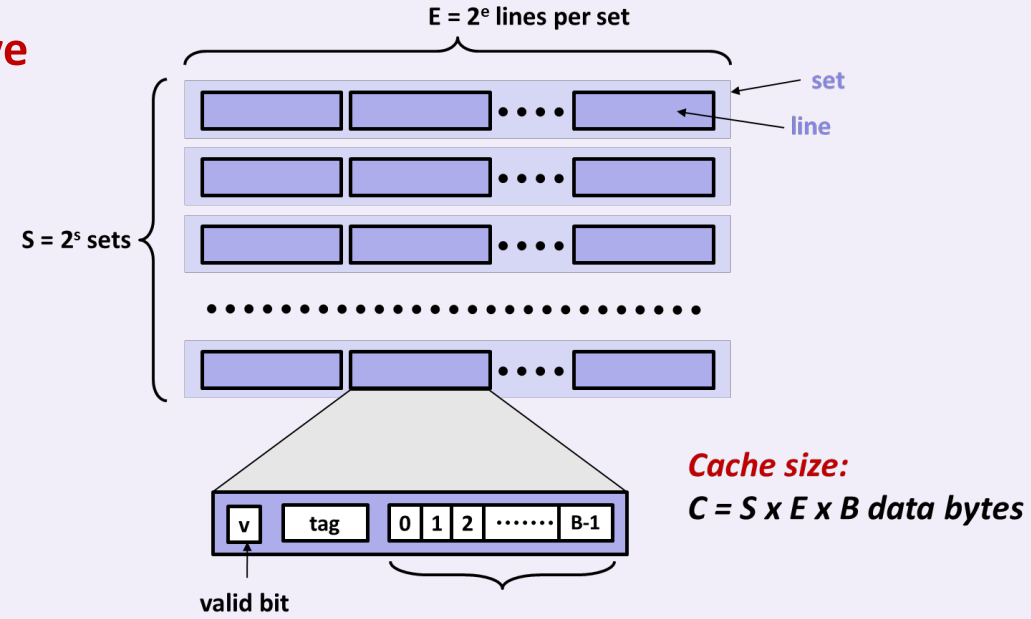
L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

Example: Core i7 L1 Data Cache

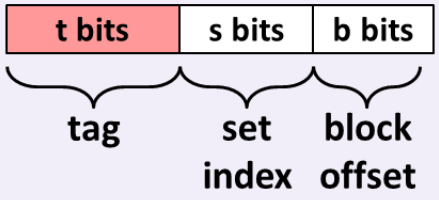
32 kB 8-way set associative
64 bytes/block
47 bit address range

B =
S = , s =
E = , e =
C =



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



Block offset: . bits
Set index: . bits
Tag: . bits

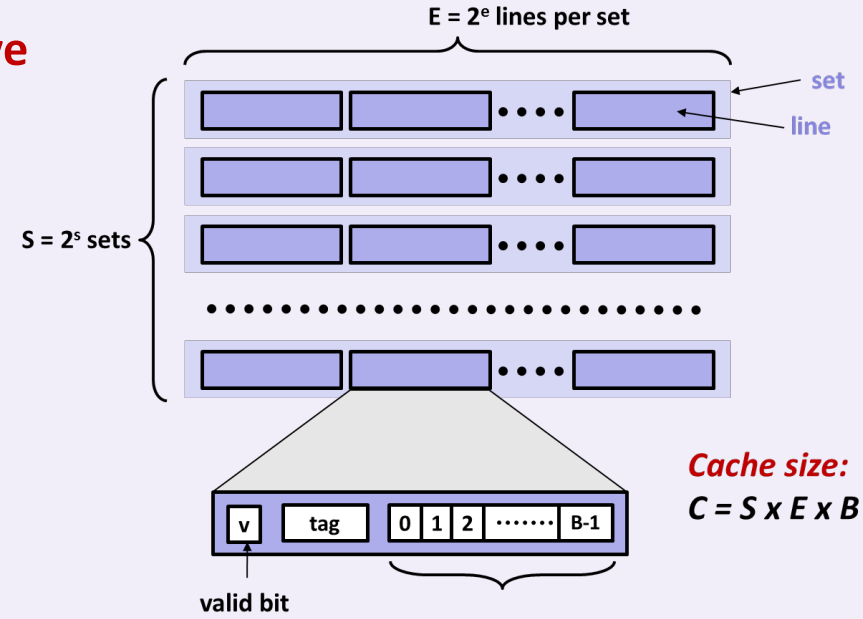
Stack Address:
0x00007f7262a1e010

Block offset: 0x??
Set index: 0x??
Tag: 0x??

Example: Core i7 L1 Data Cache

32 kB 8-way set associative
64 bytes/block
47 bit address range

B =
S = , s =
E = , e =
C =



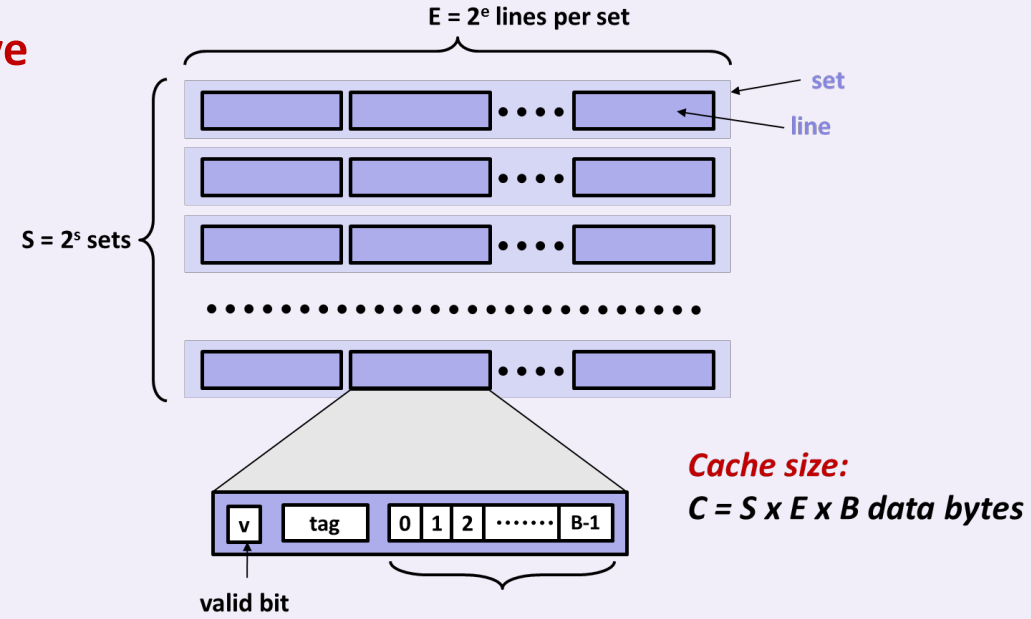
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Blocks in cache: **C / B**
Blocks in cache: **32 * 1024 / 64 = 512**

Example: Core i7 L1 Data Cache

32 kB 8-way set associative
64 bytes/block
47 bit address range

B =
S = , s =
E = , e =
C =



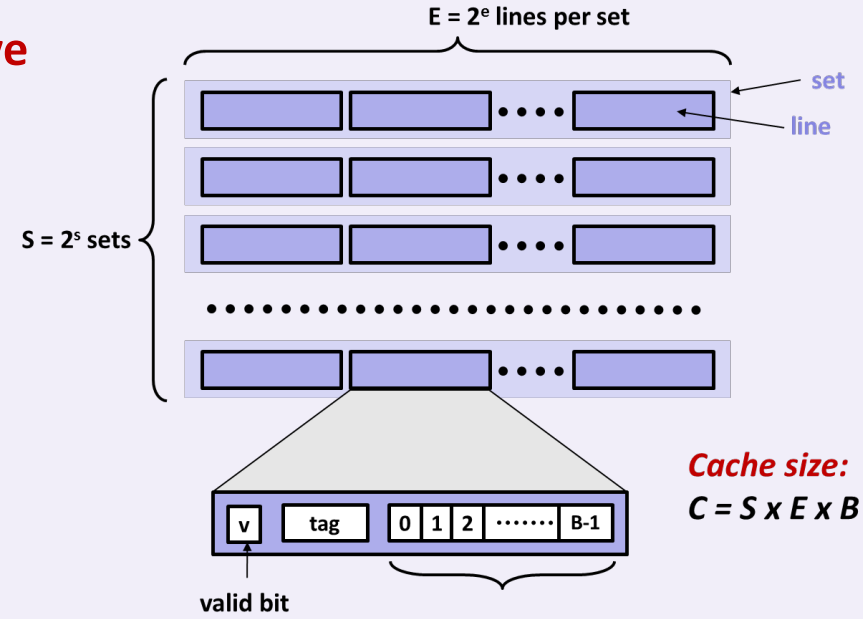
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Blocks in cache: C / B
Blocks in cache: 32 * 1024 / 64 = 512
Sets in cache: (Blocks in cache) / E
Sets in cache: (C / B) / E
Sets in cache: 512 / 8 = 64

Example: Core i7 L1 Data Cache

32 kB 8-way set associative
64 bytes/block
47 bit address range

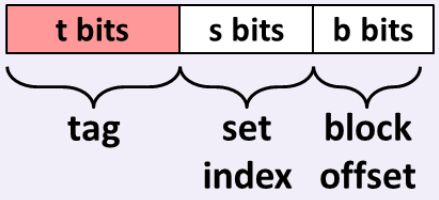
B = 64
S = 64, s = 6
E = 8, e = 3
C = 64 x 64 x 8 = 32,768



Cache size:
C = S x E x B data bytes

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



Block offset: 6 bits
Set index: 6 bits
Tag: 35 bits

Stack Address:
0x00007f7262a1e010

0000 0001 0000

Block offset: 0x10
Set index: 0x0
Tag: 0x7f7262a1e

Lab #5

Write a configurable cache simulator in C

- Inputs:
  - s, E, b
  - Address trace
- Create data structures to store **meta data** for specified cache configuration
  - Not actually storing the data block that stores application data
- Read every address
  - Determine if hit/miss in cache
  - Update cache's meta data for this access
- Print statistics about miss rates

Practice On Your Own

- Suppose a cache's capacity C = 32 KB (1 KB = 1024 Bytes) and its block size B = 64 B. Suppose addresses are specified in 32 bits.
- If the cache is direct mapped, how many bits would be used for the tag, set index, and block offset bits?

- If the cache is 4-way set associative, how many bits would be used for the tag, set index, and block offset bits?

Today

- Cache memory organization and operation (Ch 6.4)
- Caches in real systems
- Performance impact of caches
  - The memory mountain

Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses)
    = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite 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
  - Typical numbers:
    - 4 clock cycles for L1
    - 10 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

Let's think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Average access time:

    97% hits: 1 cycle + 0.03 x 100 cycles = **4 cycles**

    99% hits: 1 cycle + 0.01 x 100 cycles = **2 cycles**

- This is why "miss rate" is used instead of "hit rate"

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

The Memory Mountain

- Read throughput (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

- Memory mountain: Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.

Memory Mountain Test Function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array "data" with stride of "stride", using
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Call `test()` with many combinations of `elems` and `stride`.

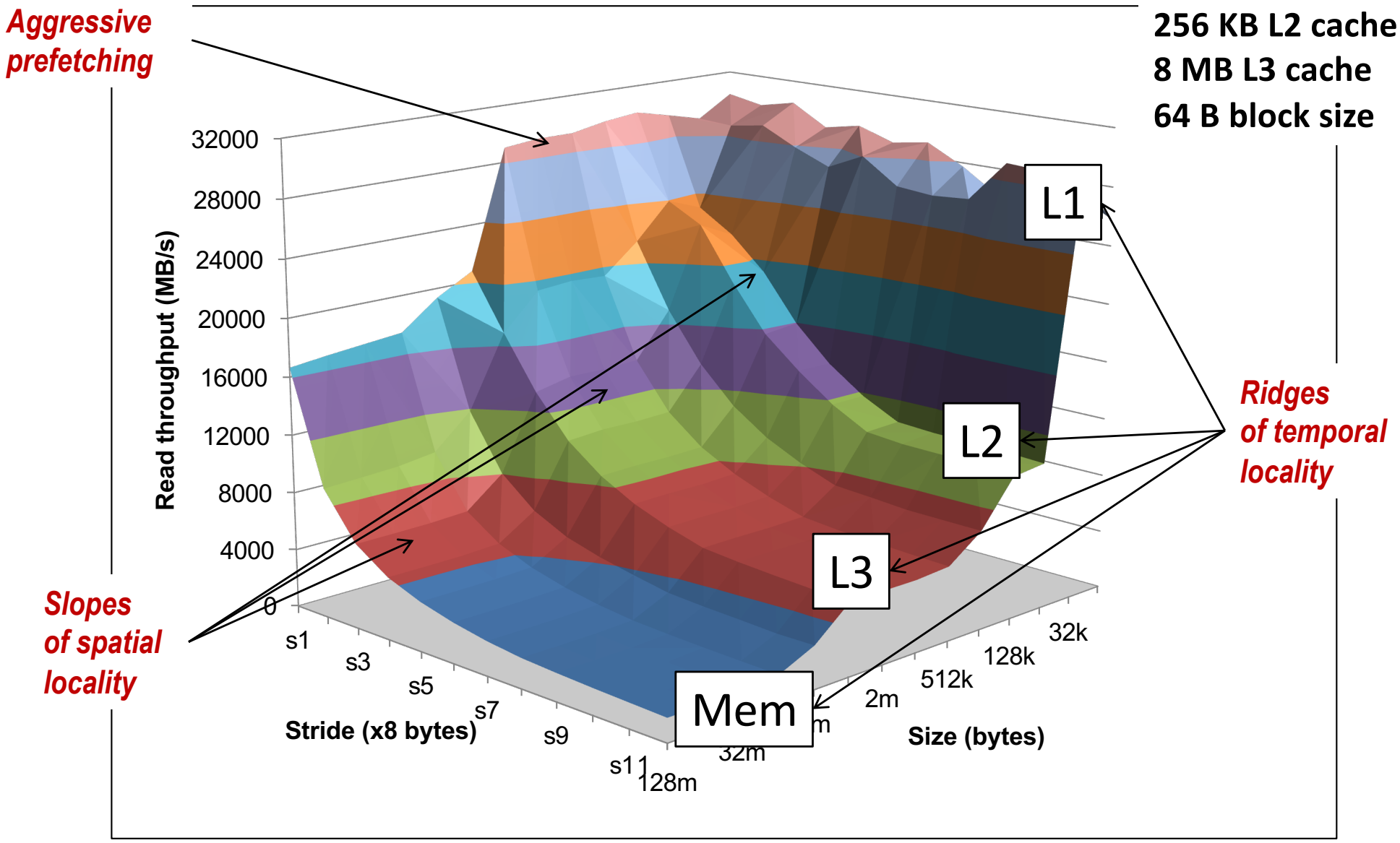
For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

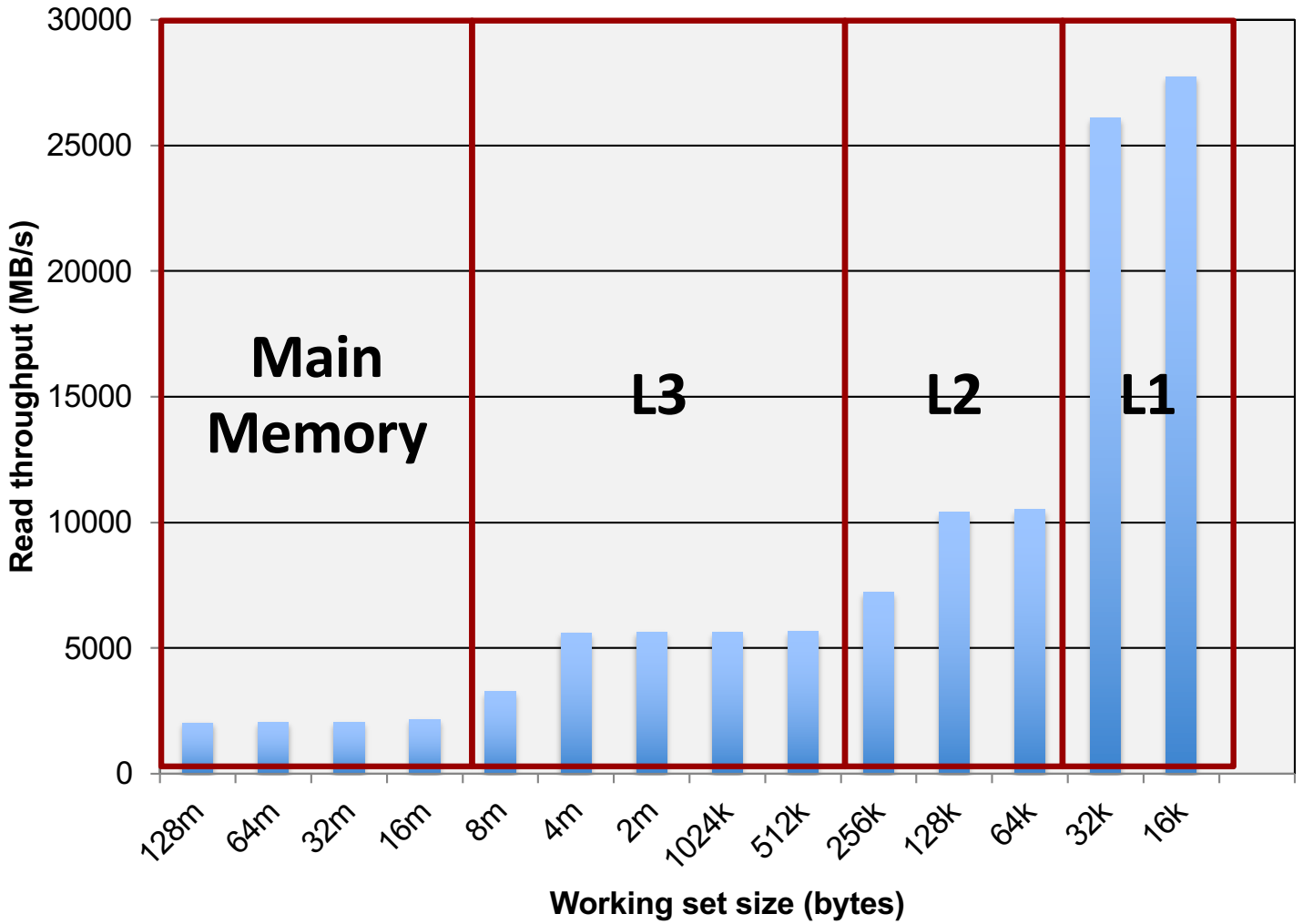
2. Call `test()` again and measure the read throughput(MB/s)

The Memory Mountain

Core i5 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Cache Capacity Effects from Memory Mountain

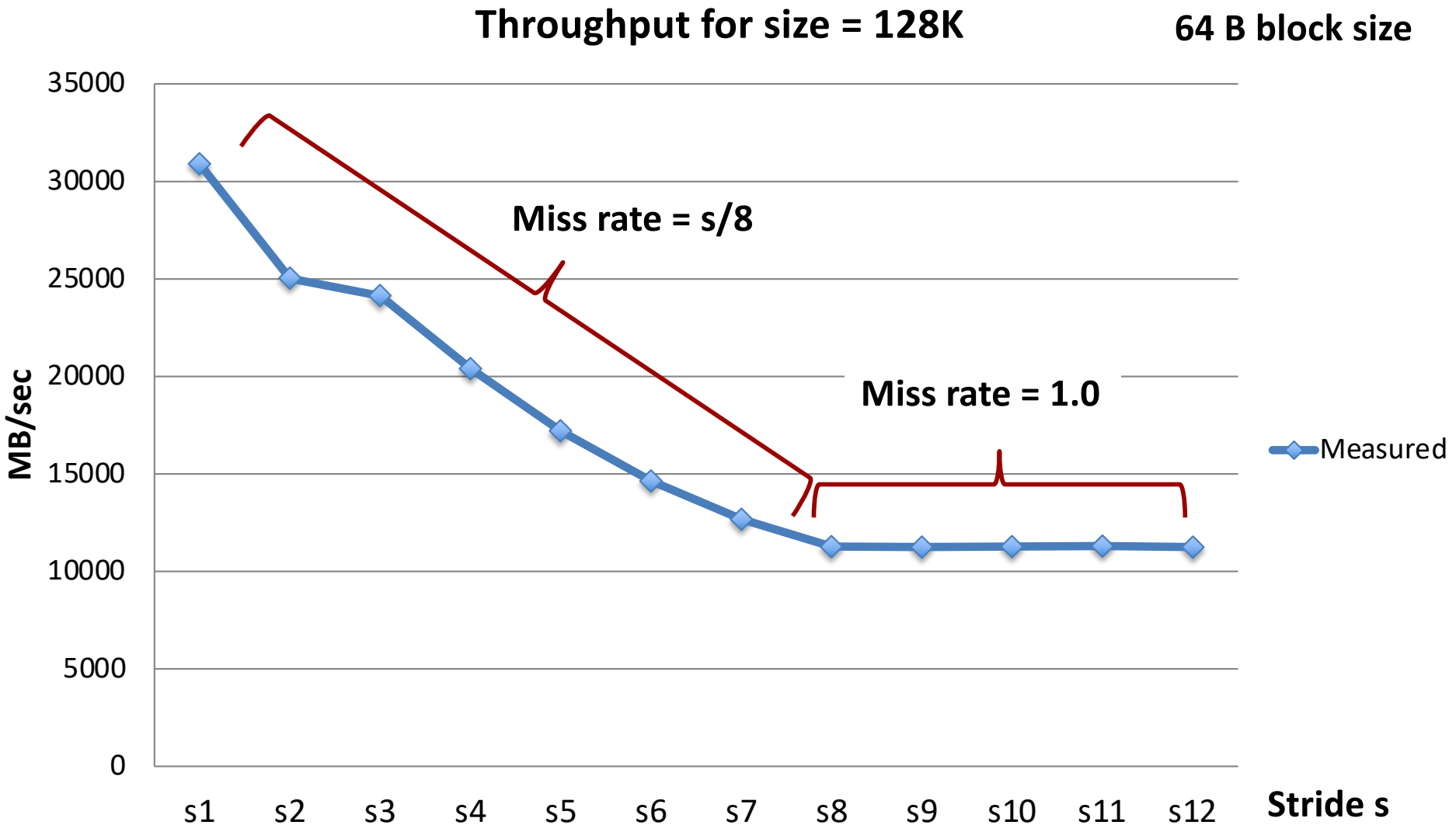


Core i7 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Slice through memory mountain with stride=8

Cache Block Size Effects from Memory Mountain

Core i7 Haswell
2.26 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Writing Cache Friendly Code

- Decrease size of working set
  - Reduce individual element size by using smaller type
    - Use a short instead of an int if values fit in 2 bytes
  - Reduce wasted space in structs due to alignment
- Nest loops in an order that improves spatial locality when accessing array elements
- Combine parallel arrays into single arrays of structs

```
int array1[1000;
int array2[1000];
```

```
struct combined{
    int element1;
    int element2;
};
struct combined array[1000];
```

Writing Cache Friendly Code (cont.)

- Reorganize computation to access subsets of arrays that fit into caches
  - Blocking
    - Ex. Matrix multiplication with large matrices

Practice on your own

- Suppose a cache has S=4 sets, a set associativity E = 2, and a block size B = 4 B. For the following addresses specified with 8 bits each, determine whether they hit or miss in the cache and determine the final state of the cache (tag bits and memory addresses stored in each cache block).
  - 0x22
  - 0x17
  - 0x75
  - 0x15
  - 0x26
  - 0x1E
  - 0x10
  - 0x24