UNM SCHOOL of ENGINEERING
Department of Computer Science

Lecture #20

Ch. 6 The Memory Hierarchy (4)

Prof. Soraya Abad-Mota, PhD

From the index cards

- The answers to the questions posed were not satisfactory. (I marked most of the ones I read, you need to go over those again and make sure you study to be able to answer them correctly.)
- E.g. the number of the set is not stored in the line.

Questions

Cache organization

1. What is a line of cache?
2. What is the size of a line of cache? What is stored in a line?
3. How are memory addresses used when storing information in the cache?
4. What is `m`? how does it relate to the size of the cache?
5. List and describe the parameters that define a cache.

Two more questions

1. List the steps to see if the data stored at a memory address is in the cache (i.e. is it a cache hit or a cache miss).
2. How do we determine how many bits are needed for: CO, CI, and CT?

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 cycle for L1
    - 10 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - typically 50 cycles from L3, 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 * 100 cycles = **4 cycles** (0.97*1+0.03*100=3.97)
    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles** (0.99*1+0.01*100=1.99)
- This is why "miss rate" is used instead of "hit rate"

Performance impact of cache params.

- Section 6.4.7 (pp. 631-633)
  - impact of cache size
  - impact of block size
  - impact of associativity
  - impact of write strategy (-through or -back & -allocate or not)
- Read them (on your own) and understand the trade-offs
- Think of examples to make sure you truly understand what the compromises are.

Qualitative Tradeoff examples

- larger cache → increased hit rate, but larger hit time L1<L2<L3 ("it is hard to make large memories run faster")
- Associativity (parameter E)
  - large E → decreases vulnerability of cache to thrashing (we saw an example), but it is expensive to implement and hard to make fast.
  - tradeoff: hit time and miss penalty
  - E.g. L1 and L2 are 8-way cache but L3 is 16-way cache (miss penalty is higher)

Qualitative Tradeoff examples (2)

- Larger blocks → smaller number of cache lines which can reduce hit rate if there is more temporal than spatial locality
- Larger blocks→ larger miss penalty

Locality in the light of the measures

Programs with better locality (spatial or temporal)

- ➔ lower miss rates ➔ run faster than programs with higher miss rates

Writing code with *Good locality* makes the code *cache friendly*

Writing Cache Friendly Code (sec. 6.5)

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

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
    can be cached in the registers!
  - Stride-1 reference patterns are good (spatial locality)
    data are stored in contiguous blocks

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories and their parameters. Measure hits and misses and make decisions based on that.

Measuring hits and misses on a cache

```
int sumvec(int V[N]) {
    int i, sum = 0;
    for (i=0; i < N; i++)
          sum += v[i]
    return sum;
}
```

cache block size 16 bytes (4 ints fit in one line), v is block-aligned

v[i] (N=8) i=0 i=1 i=2 i=3 i=4 i=5 i=6 i=7

access order, 1[**m**] 2[h] 3[h] 4[h] 5[**m**] 6[h] 7[h] 8[h]

[h]it or [m]iss (only in this case, regardless of cache org.)

Example problems (pp.635-636)

Book explains in detail accesses to a 2D-array and provides which are hits and which misses.

and

To work with specific examples of locality and illustrate the tradeoffs.

For all these exercises, you do need to work with the cache org. and the cache parameters to see which addresses are placed where in the cache.

- Do 6.17 now and
- (at least one of) 6.18, 6.19, or 6.20 (during labs this week)

Problem 6.17: transposing a matrix

```
typedef int array[2][2]
void transpose1(array dst, array src) {
   int i,j;
   for (i=0; i<2; i++){
        for (j = 0; j<2; j++){
             dst[j][i] = src[i][j];
        }
   }
}
```

6.17 (cont.)

Assume this code runs on a machine with the following properties:

- `sizeof(int)` = 4.
- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, and write-allocate, with a block size of 8 bytes.
- The cache has a total size of 16 data bytes and the cache is initially empty.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

A. For each `row` and `col`, indicate whether the access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

	dst array			src array	
	Col. 0	Col. 1		Col. 0	Col. 1
Row 0	m	_____	Row0	m	_____
Row 1	_____	_____	Row 1	_____	_____

B. Repeat the problem for a cache with 32 data bytes.

Improving a program's performance

- What have we done so far?
  - Do not block the compiler
  - Do our own optimizations + Allow the optimizer to do its job
  - We have measured the performance in CPE (cycles per element)
  - Hardware helps, how?
    - pipelining and some parallelization in the microprocessor
    - cache memories help bridge the gap between main memory and CPU
  - Write cache-friendly code:
    - temporal and spatial locality
    - measure hits and misses with our knowledge of how cache works

Next topics

- **Conclude Ch. 6: The Memory Hierarchy**
  - Performance impact of caches
    - The memory mountain
    - Rearranging loops to improve spatial locality
    - Using blocking to improve temporal locality
- Chapter 7: Linking
- Chapter 8: Exceptional Control Flow

The Memory Mountain (section 6.6.1)

- *Read throughput* (or *read bandwidth*)
  - Number of bytes read from memory per second (MB/s) (rate at which a program reads data from memory)

- Memory mountain: Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.
  - Use the Test function (shown next) to build the mountain.

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`.

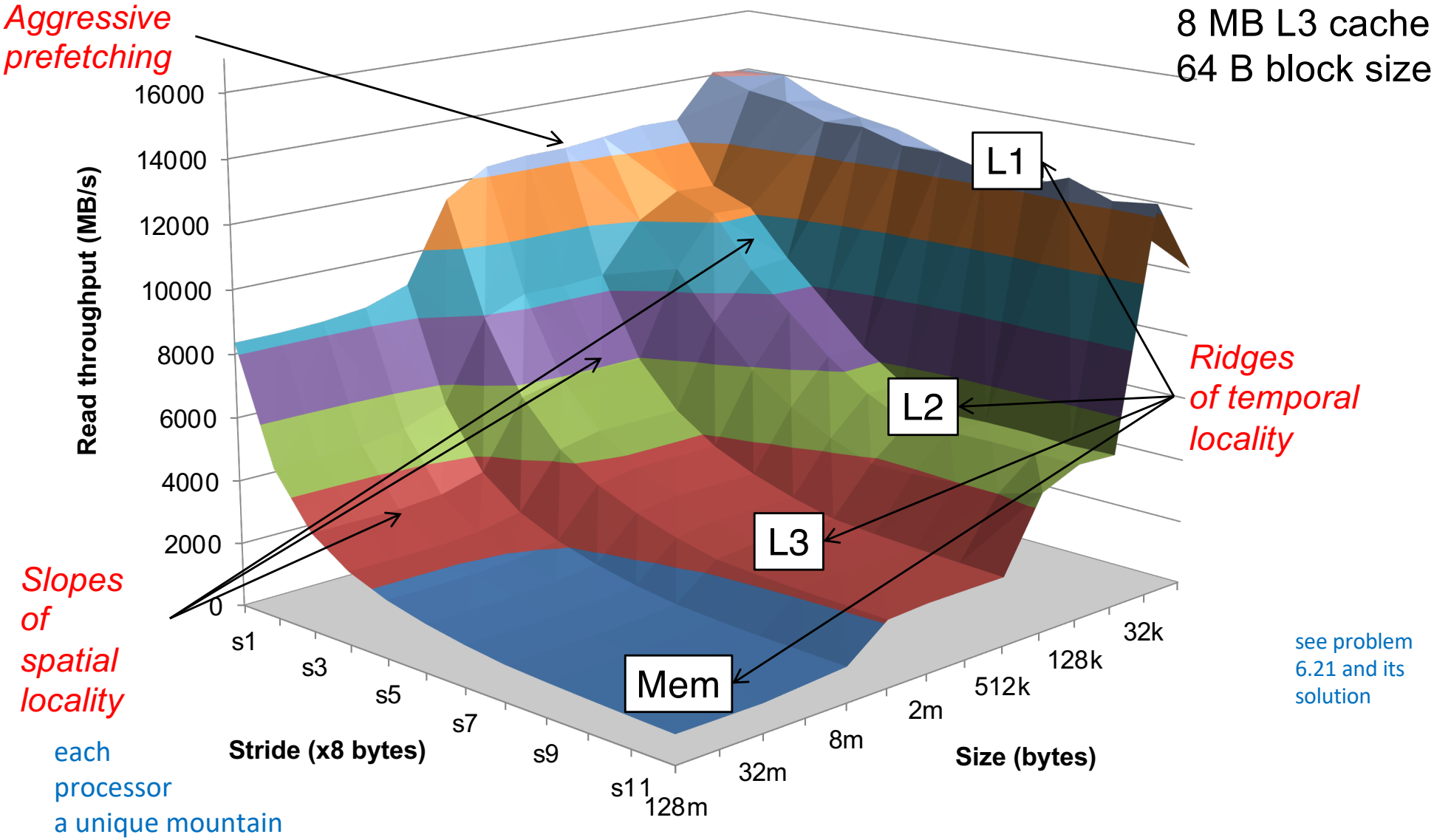
For each elems and stride:

1. Call test() once to warm up the caches. (seen in run function fig. 6.40, p.640!)

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

The Memory Mountain

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



each processor a unique mountain

see problem 6.21 and its solution

Memory Mountain Analysis

- Every computer has a unique memory mountain.
- size allows to control temporal locality
- stride allows to control spatial locality
- Read throughput can vary from 14GB/s in L1 to 900 MB/s in main memory
- Poor temporal locality (e.g. working set too large) can be compensated (somehow) by spatial locality.
- HW prefetching together with spatial locality also helps even with large size by fetching some blocks into cache, before they are accessed.

Use of the Memory Mountain

- "Wise programmers try to structure their programs so that they run in the peaks instead of the valleys."
- Goal:
  - to exploit temporal locality, i.e. heavily used words fetched from the L1 cache
  - to exploit spatial locality, i.e. as many words as possible accessed from a single line in L1 cache

Details of memory mountain analysis

May be studied in:

- Section 6.6.1 pp. 639-643 contains the full discussion of the metrics used and plotted in the graph.
- Figure 6.42 p. 642 shows the ridges of temporal locality in 2D and Fig. 6.43 shows a slope of spacial locality p. 643