CS 261
Fall 2017

Mike Lam, Professor

Caching

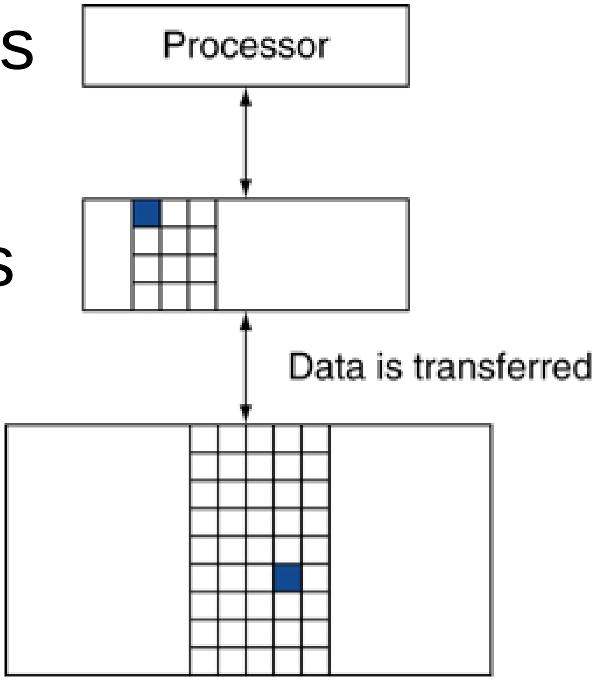
(get it??)

Topics

- Caching
- Cache policies and implementations
- Performance impact
- General strategies

Caching

- A cache is a small, fast memory that acts as a buffer or staging area for a larger, slower memory
  - Fundamental CS system design concept
  - Data is transferred in blocks
  - Slower caches use larger block sizes
  - Cache hit vs. cache miss
  - Hit ratio: # hits / # memory accesses



Caches

- A cache always begins cold (empty)
  - Every request will be a cold miss initially
- As the cache loads data, it is warmed up
  - This effect can cause performance measurement variation during experiments if not controlled for
- A working set is a collection of elements needed repeatedly for a particular computation
  - If the working set doesn't fit in cache, this is called a capacity miss

Cache implementations

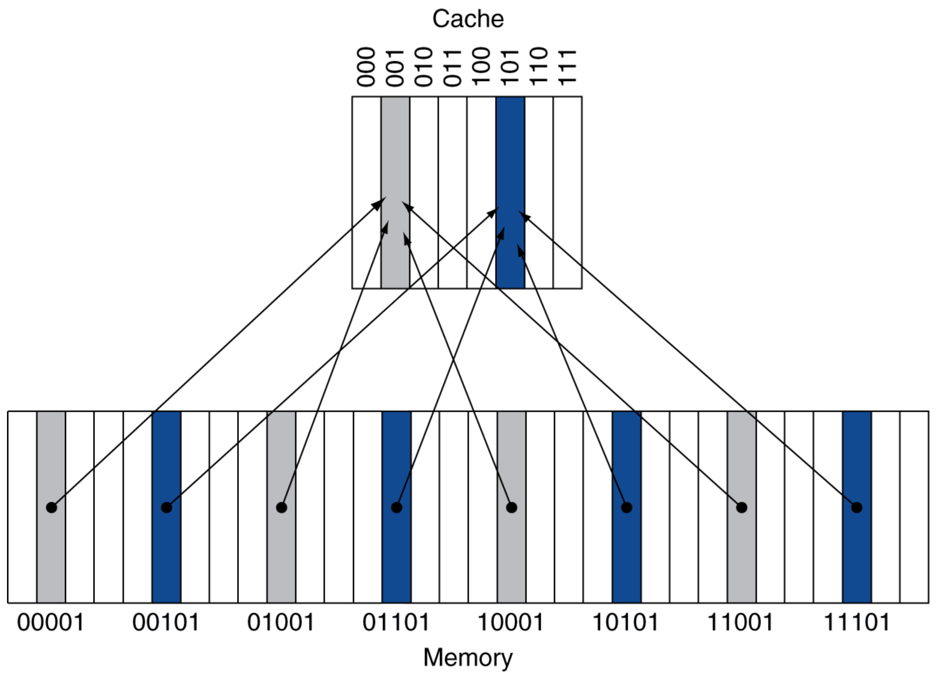
- What data structure can we use to implement caches?
  - Need **FAST** lookups and containment checks

Cache implementations

- What data structure can we use to implement caches?
  - Need **FAST** lookups and containment checks
  - From CS 240: use a hash table!
  - Cache address = "real address" % CACHE_SIZE

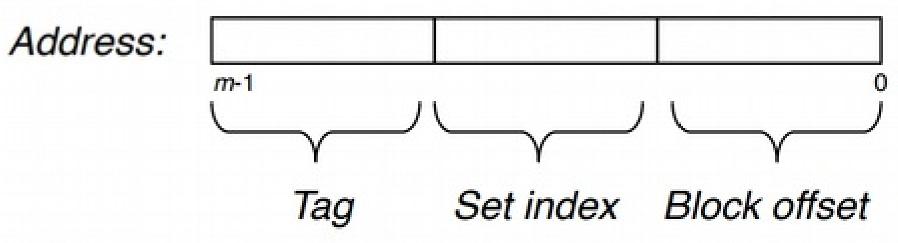
What if we wanted our cache to store blocks longer than a single byte?

What if multiple "real" addresses map to the same cache slot?



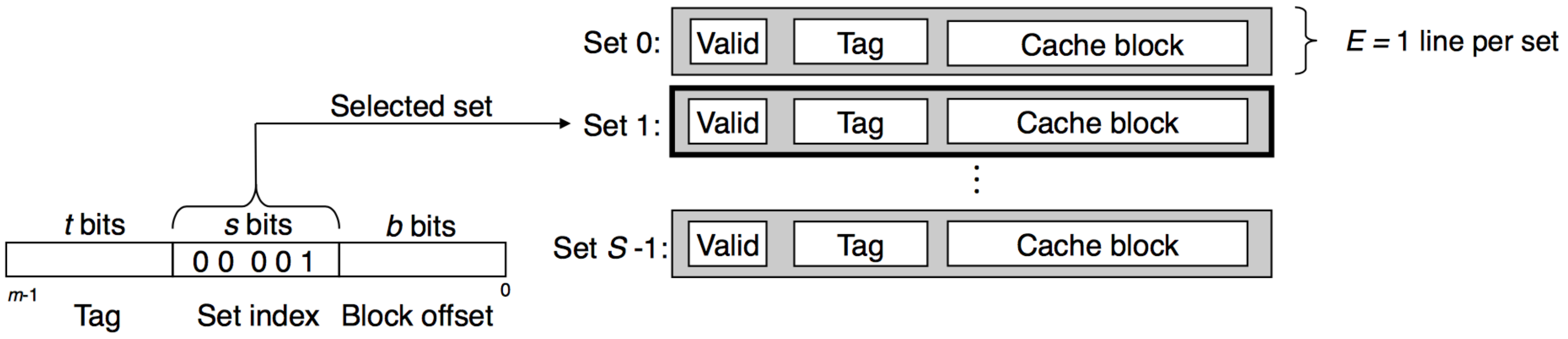
Cache implementations

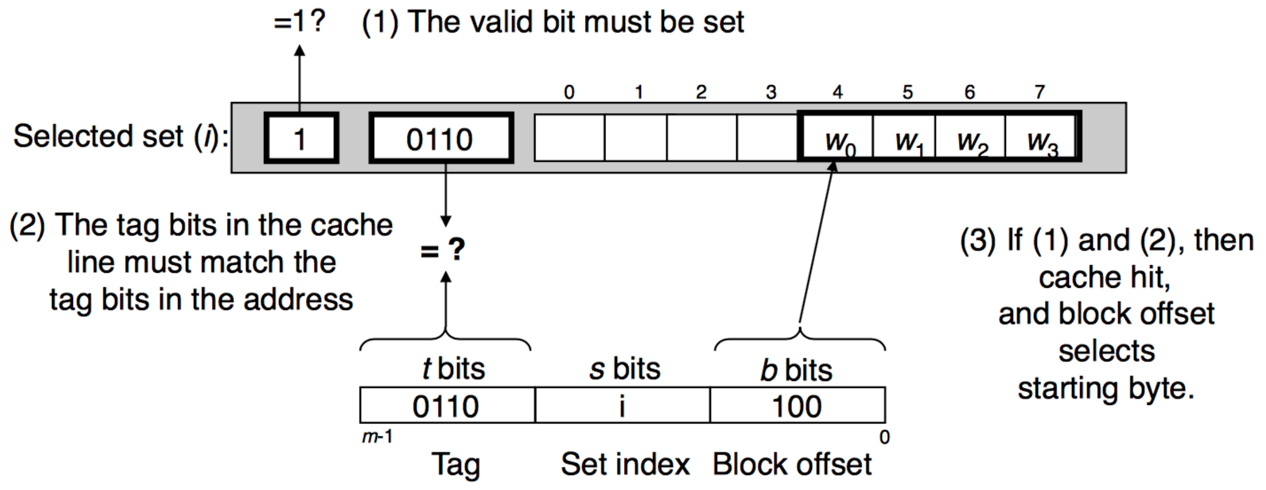
- A cache line is a block or sequence of bytes that is moved between memory levels in a single operation
- A cache set is a collection of one or more cache lines
  - Each cache line contains a tag to identify the source address and a valid flag/bit indicating whether the value is up-to-date
- Cache parameters (S, E, B, m):
  - **S** = # of cache sets = 2^s
    - s = # of bits for set index
  - **E** = # of lines per cache set
  - **B** = block (cache line) size = 2^b
    - b = # of bits for block offset
  - **m** = # of bits for memory address
    - M = size of memory in bytes = 2^m
  - C = total cache capacity = S x E x B
  - t = # of tag bits = m - s - b



Cache implementations

- Direct-mapped (E = 1) caches

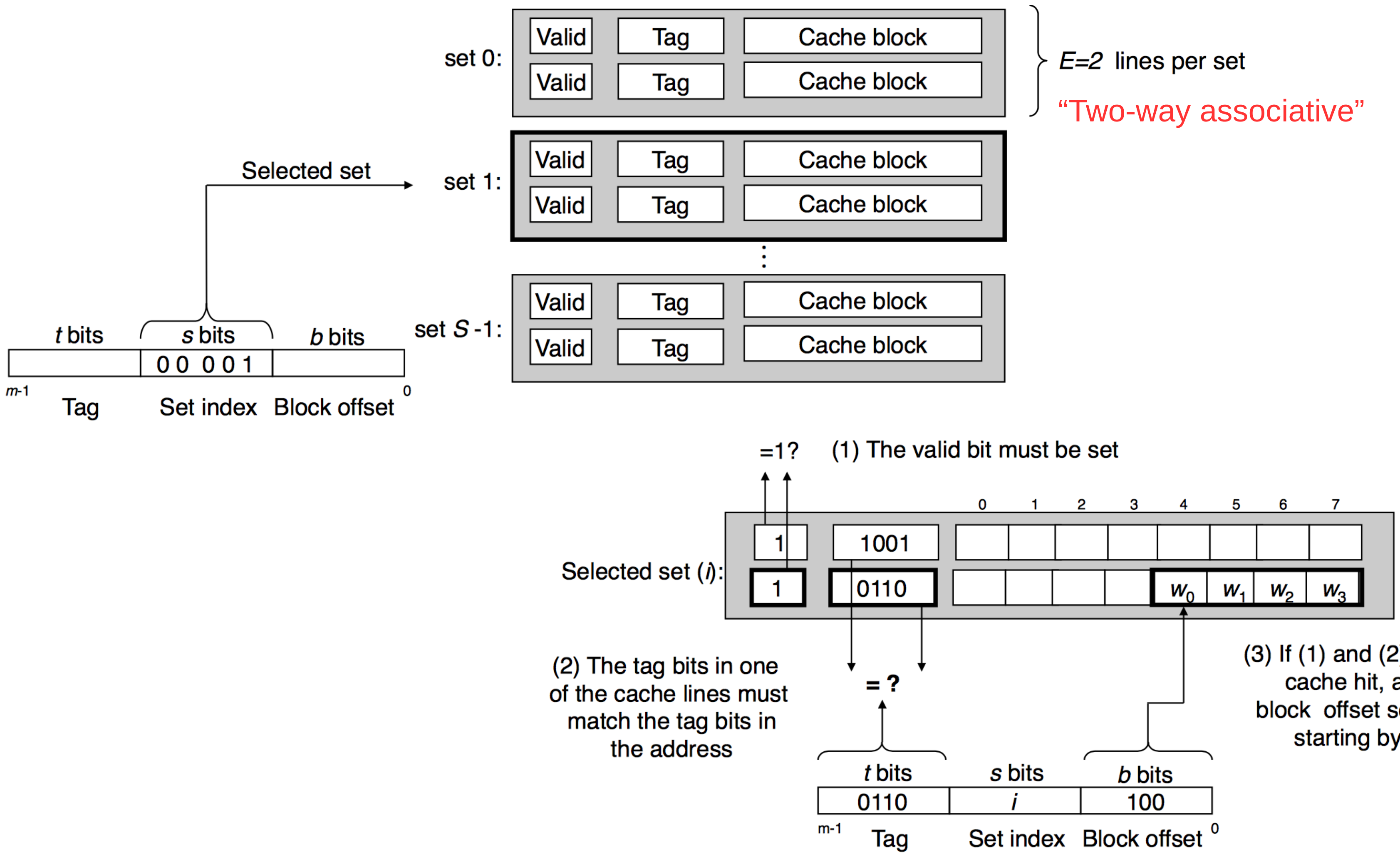




=1?
(1) The valid bit must be set
Selected set (i):
1
0110
0 1 2 3 4 5 6 7
w0 w1 w2 w3
(2) The tag bits in the cache line must match the tag bits in the address
= ?
(3) If (1) and (2), then cache hit, and block offset selects starting byte.
t bits
s bits
b bits
0110
i
100
m-1
0
Tag
Set index
Block offset

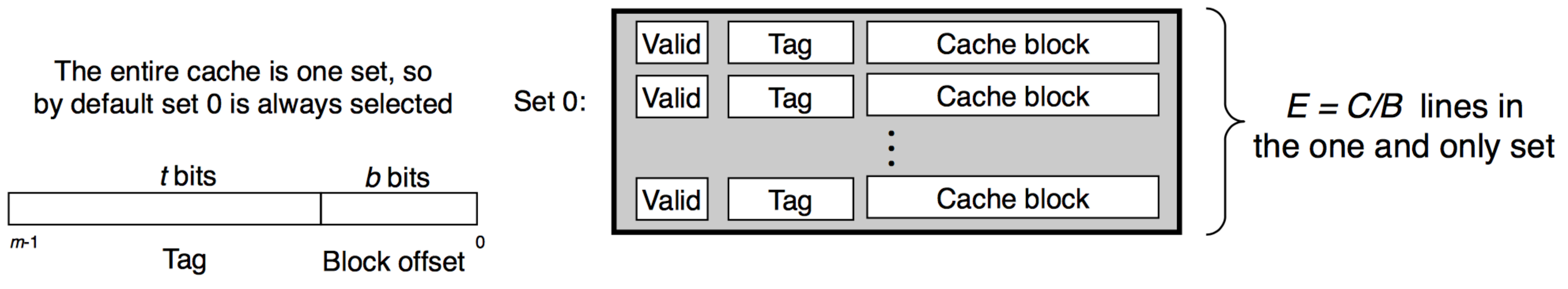
Cache implementations

- Set-associative (1 < E < C/B) caches

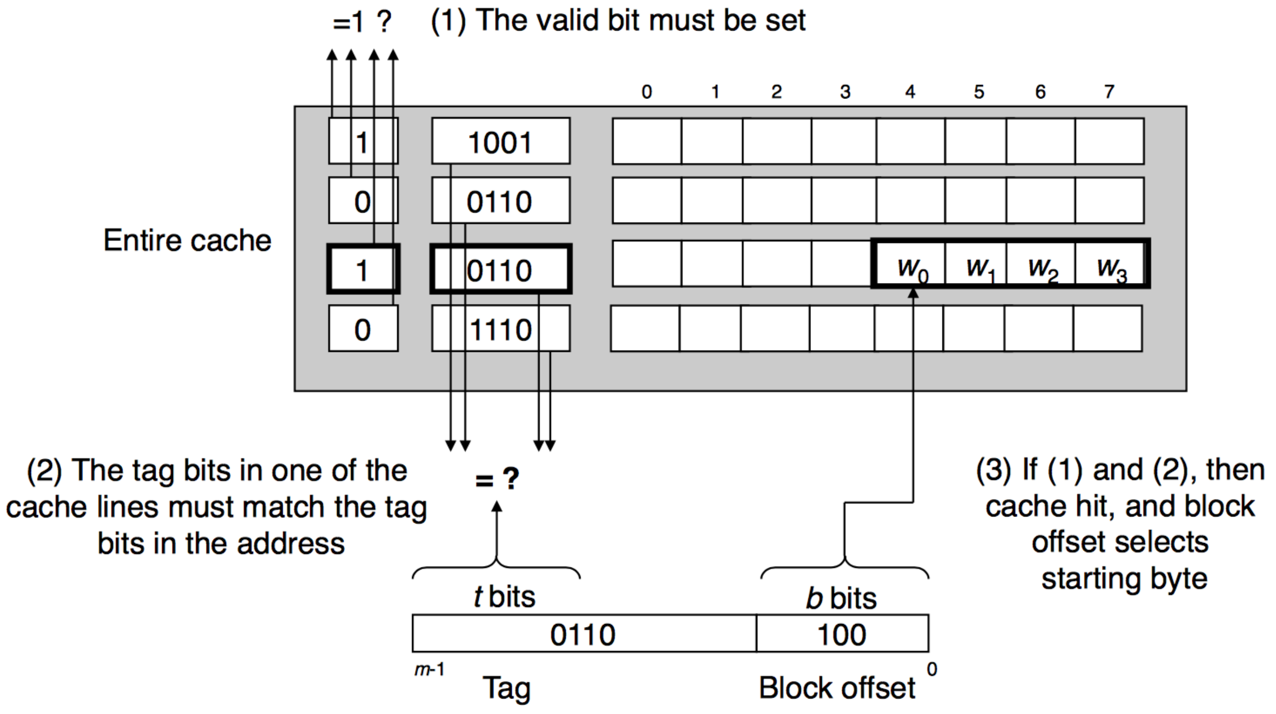


Cache implementations

- Fully-associative (E = C/B) caches



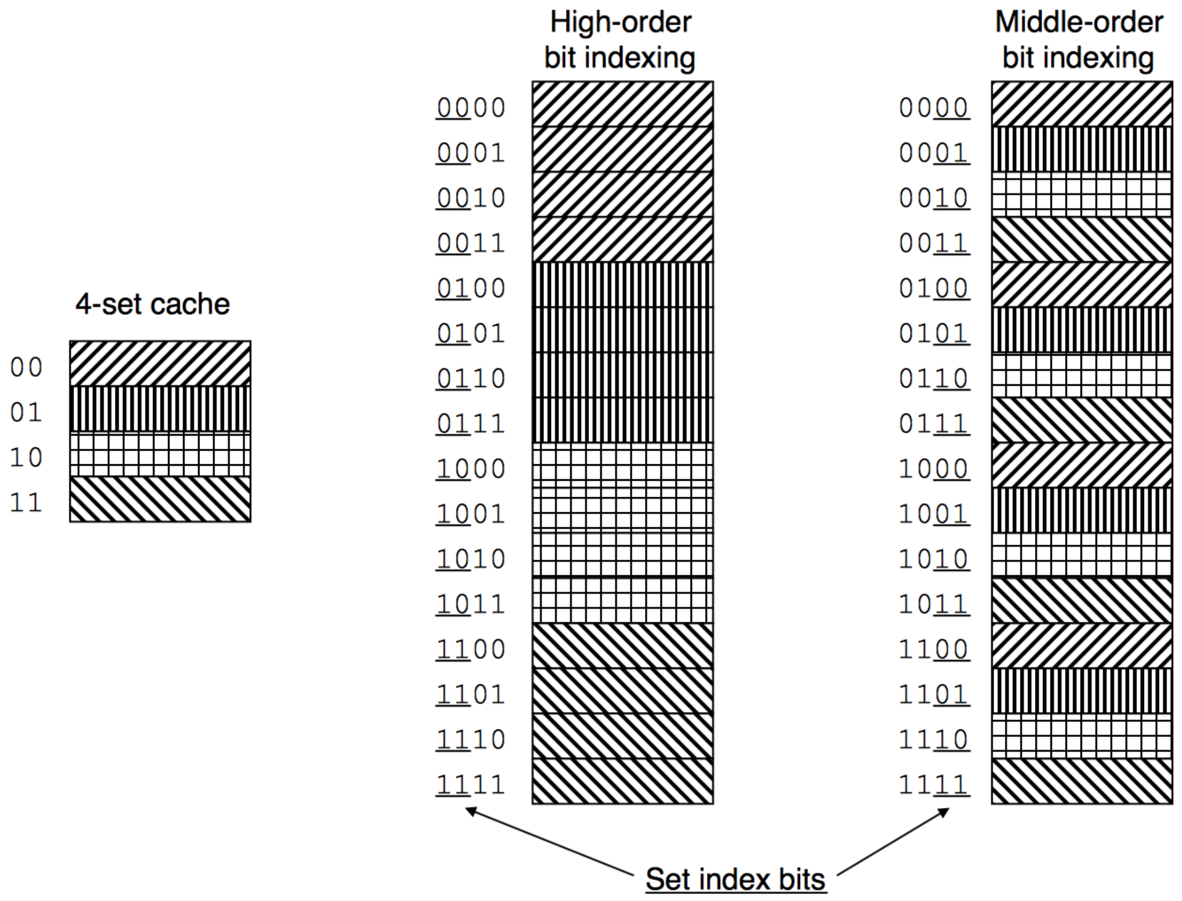
The entire cache is one set, so by default set 0 is always selected
t bits
b bits
m-1
Tag
Block offset
0
Set 0:
Valid
Tag
Cache block
E = C/B lines in the one and only set



=1 ?
(1) The valid bit must be set
0 1 2 3 4 5 6 7
Entire cache
1 1001
0 0110
1 0110
0 1110
w0 w1 w2 w3
(2) The tag bits in one of the cache lines must match the tag bits in the address
= ?
(3) If (1) and (2), then cache hit, and block offset selects starting byte
t bits
b bits
0110
100
m-1
Tag
Block offset
0

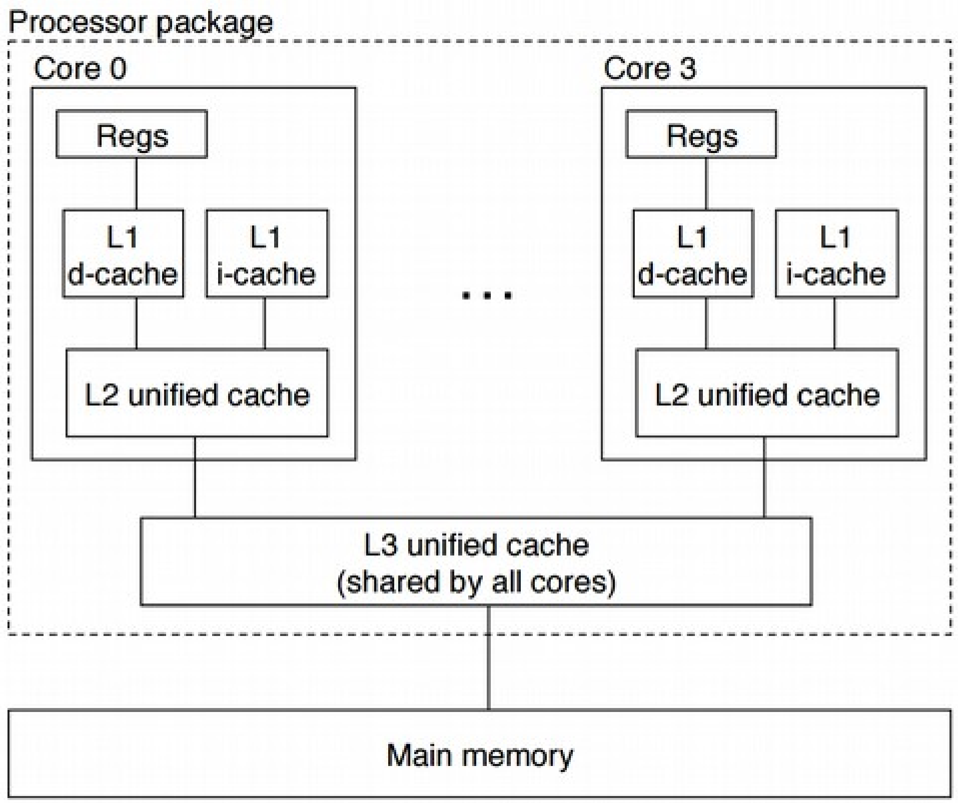
Cache implementations

- Why use the middle bits for the set index?
  - Contiguous memory blocks should map to different cache sets



Cache architecture

- Example: Intel Core i7
- Per-core:
  - Registers
  - L1 d-cache and i-cache
    - Data and instructions
  - L2 unified cache
- Shared:
  - L3 unified cache
  - Main memory



Cache policies

- If a cache set is full, a cache miss in that set requires blocks to be replaced or evicted
- Policies:
  - Random replacement
  - Least recently used
  - Least frequently used
- These policies require additional overhead
  - More important for lower levels of the memory hierarchy

Cache policies

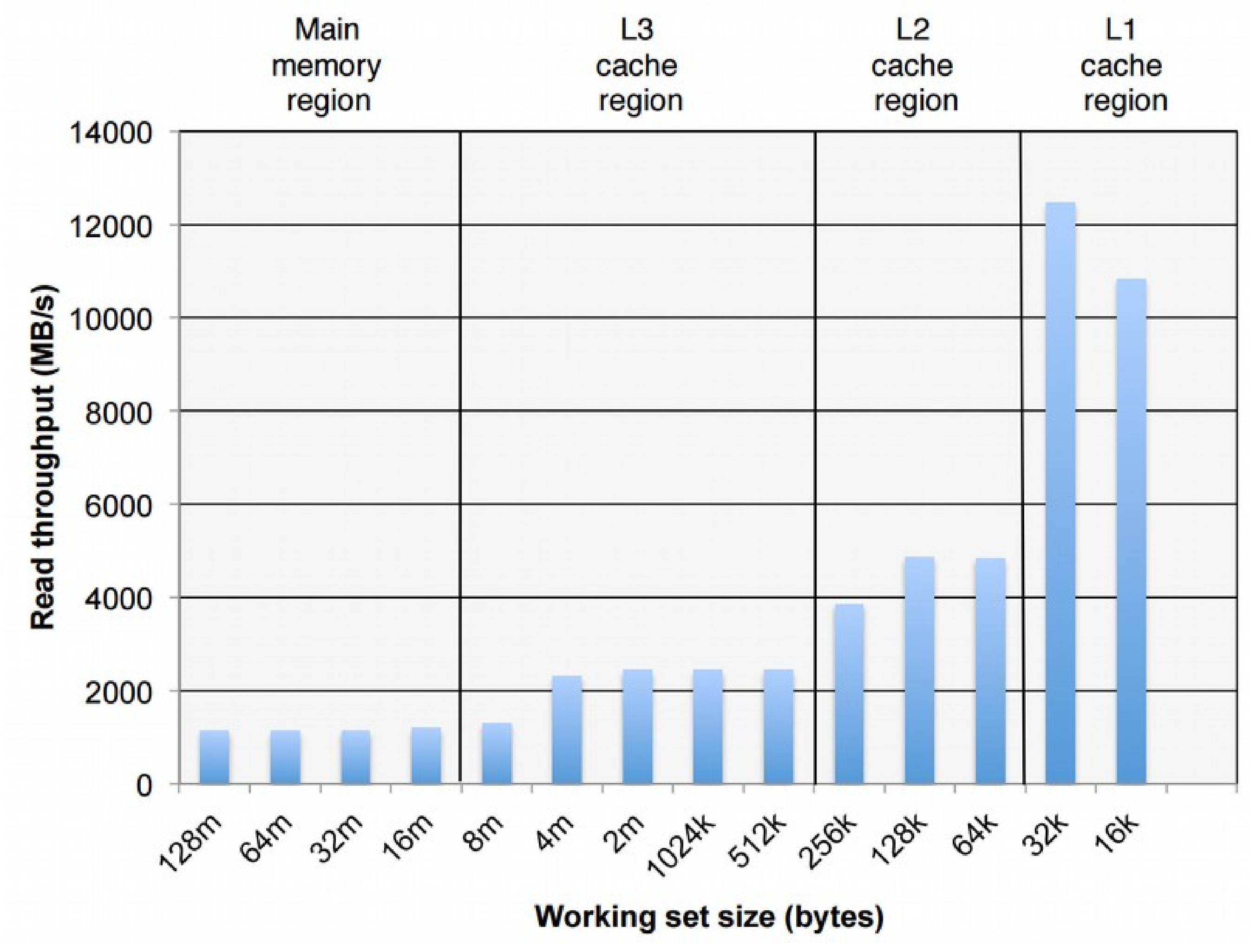
- How should we handle writes to a cached value?
  - Write-through: immediately update to lower level
    - Typically used for higher levels of memory hierarchy
  - Write-back: defer update until replacement/eviction
    - Typically used for lower levels of memory hierarchy
- How should we handle write misses?
  - Write-allocate: load then update
    - Typically used for write-back caches
  - No-write-allocate: update without loading
    - Typically used for write-through caches

Performance impact

- Metrics
  - Hit rate/ratio: # hits / # memory accesses (1 – miss rate)
    - Hit time: delay in accessing data for a cache hit
  - Miss rate/ratio: # misses / # memory accesses
    - Miss penalty: delay in loading data for a cache miss
  - Read throughput (or "bandwidth"): the rate that a program reads data from a memory system
- General observations:
  - Larger cache = higher hit rate but higher hit time
  - Lower miss rates = higher read throughput

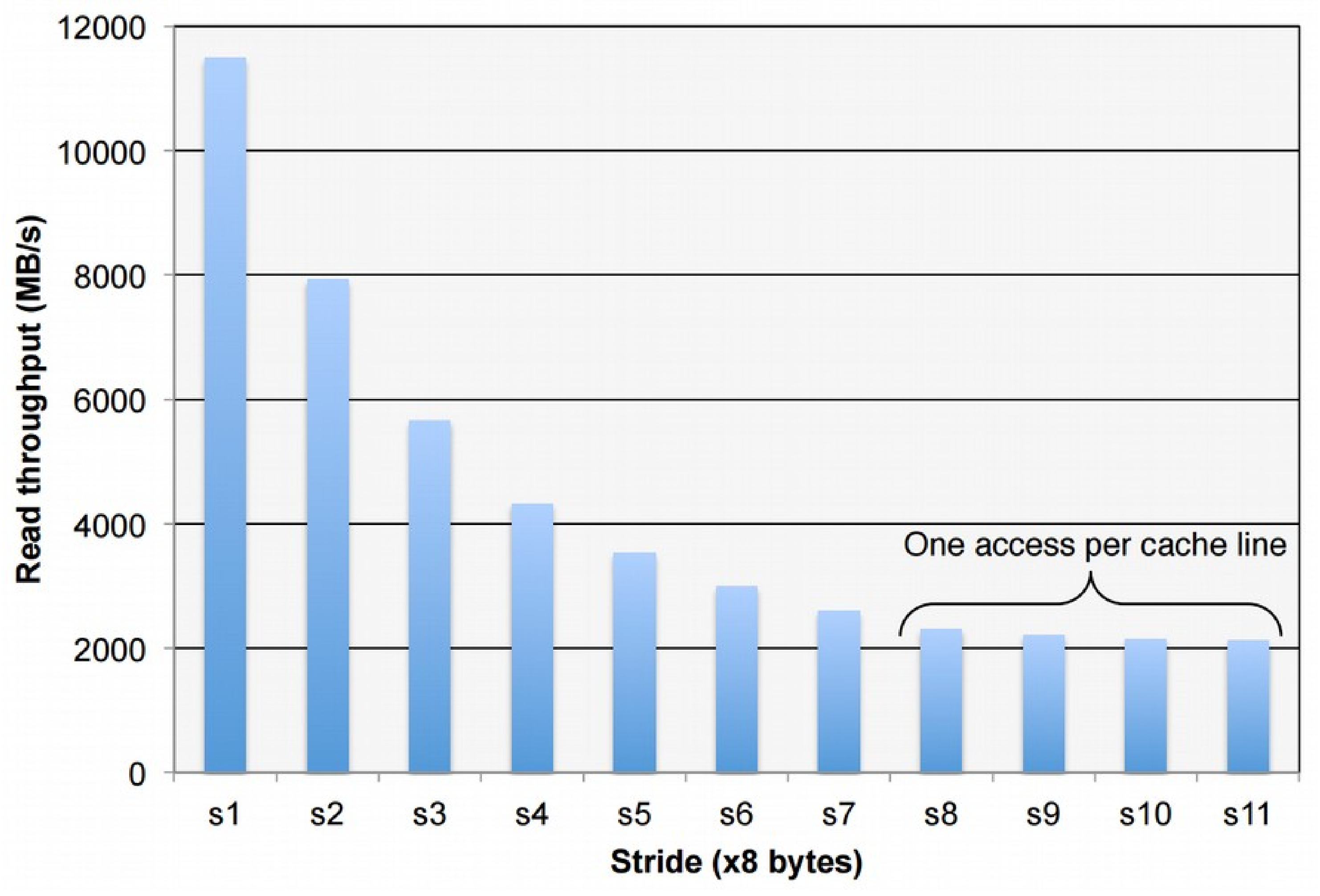
Temporal locality

- Working set size vs. throughput



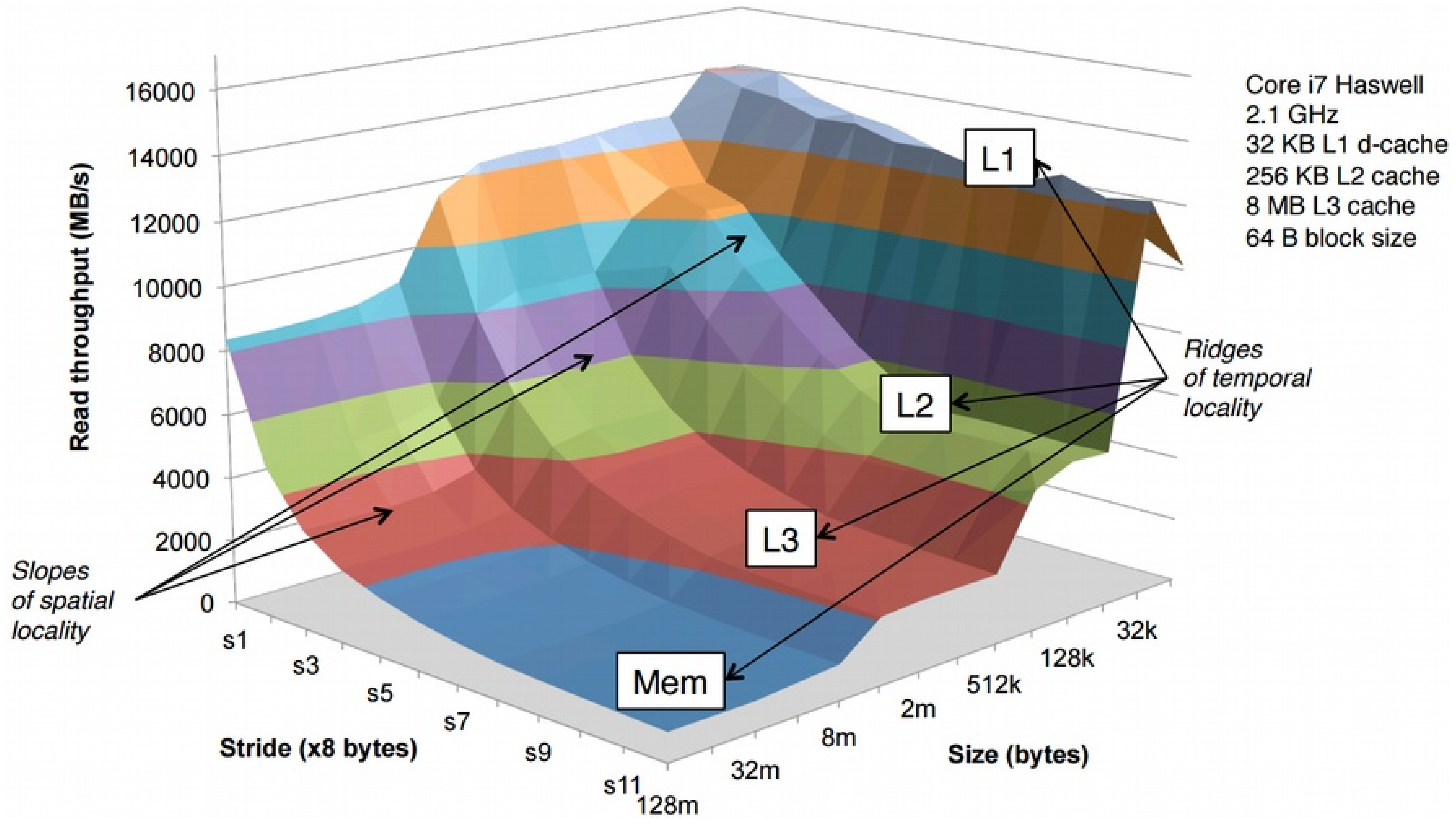
Spatial locality

- Stride vs. throughput

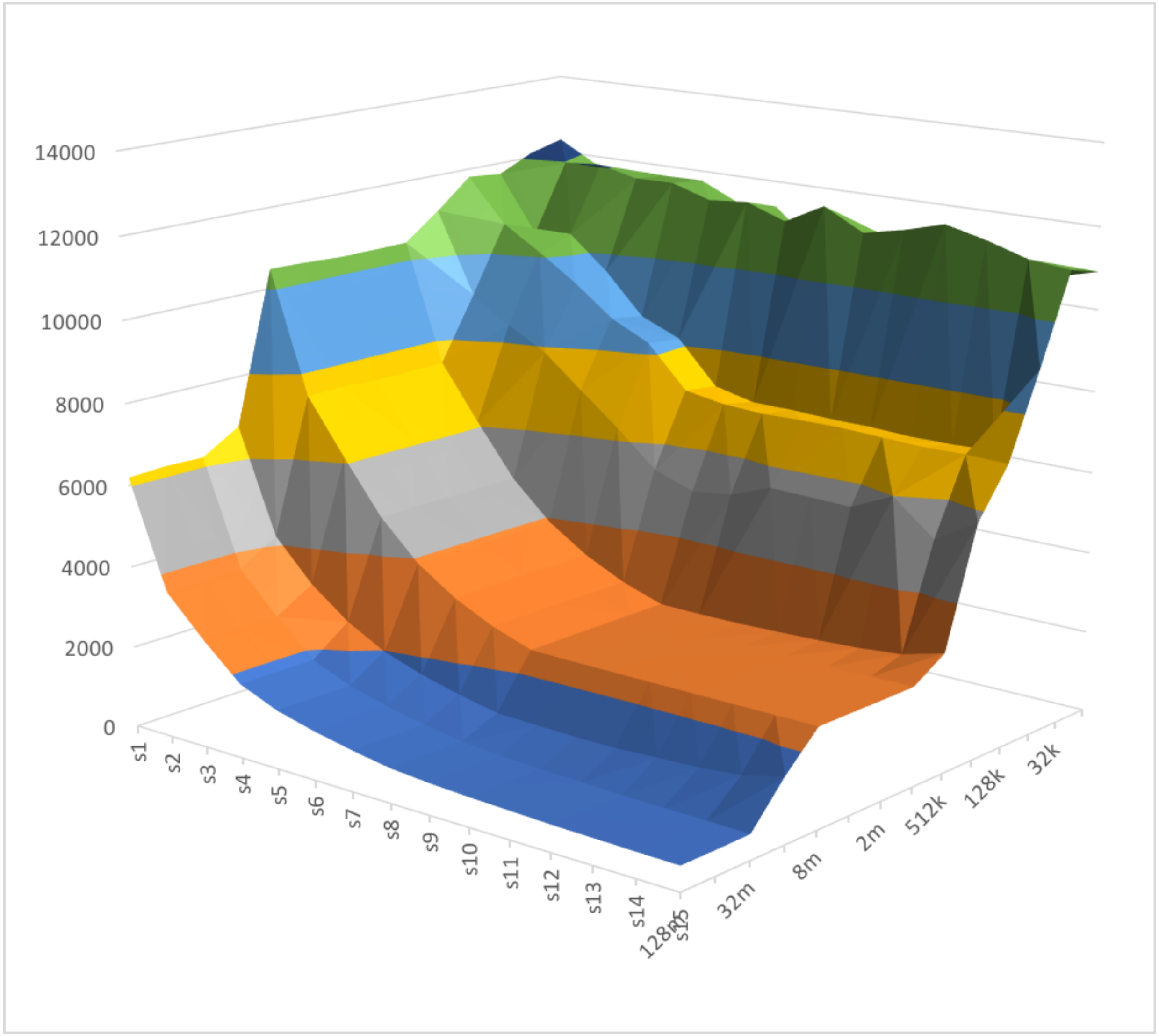


Memory mountain

- Stride and WSS vs. read throughput



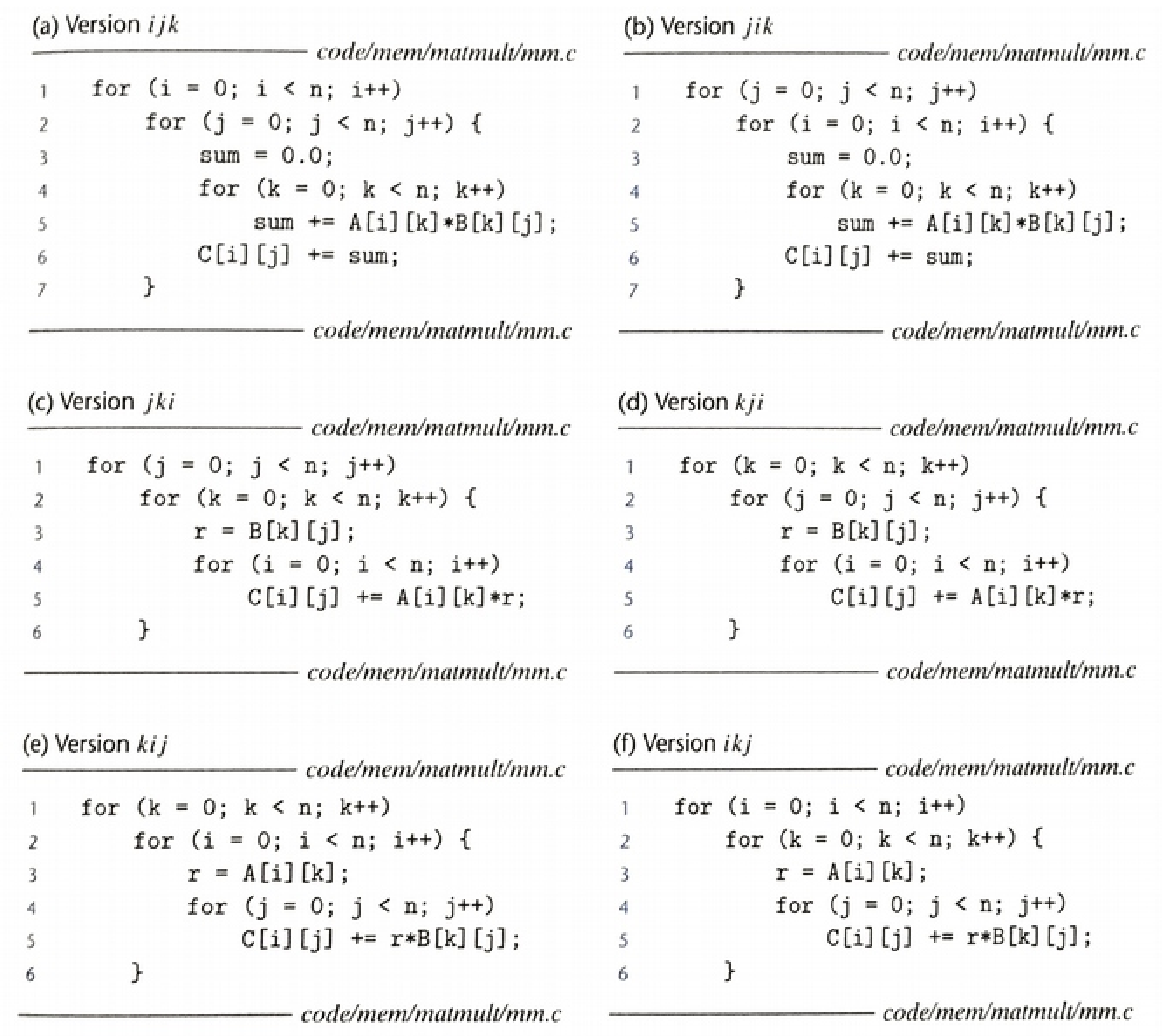
Memory mountain (stu)



Output of **lscpu**:

```
Architecture:          x86_64
Byte Order:            Little Endian
CPU(s):                24
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
Vendor ID:             Intel
Model name:
Intel(R) Xeon(R) CPU E5-2640
CPU max MHz:           3000.0000
CPU min MHz:           1200.0000
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              15360K
```

Case study: matrix multiply



(a) Version *ijk*

code/mem/matmult/mm.c

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += A[i][k]*B[k][j];
        C[i][j] += sum;
    }
```

code/mem/matmult/mm.c

(b) Version *jik*

code/mem/matmult/mm.c

```
for (j = 0; j < n; j++)
    for (i = 0; i < n; i++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += A[i][k]*B[k][j];
        C[i][j] += sum;
    }
```

code/mem/matmult/mm.c

(c) Version *jki*

code/mem/matmult/mm.c

```
for (j = 0; j < n; j++)
    for (k = 0; k < n; k++) {
        r = B[k][j];
        for (i = 0; i < n; i++)
            C[i][j] += A[i][k]*r;
    }
```

code/mem/matmult/mm.c

(d) Version *kji*

code/mem/matmult/mm.c

```
for (k = 0; k < n; k++)
    for (j = 0; j < n; j++) {
        r = B[k][j];
        for (i = 0; i < n; i++)
            C[i][j] += A[i][k]*r;
    }
```

code/mem/matmult/mm.c

(e) Version *kij*

code/mem/matmult/mm.c

```
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++) {
        r = A[i][k];
        for (j = 0; j < n; j++)
            C[i][j] += r*B[k][j];
    }
```

code/mem/matmult/mm.c

(f) Version *ikj*

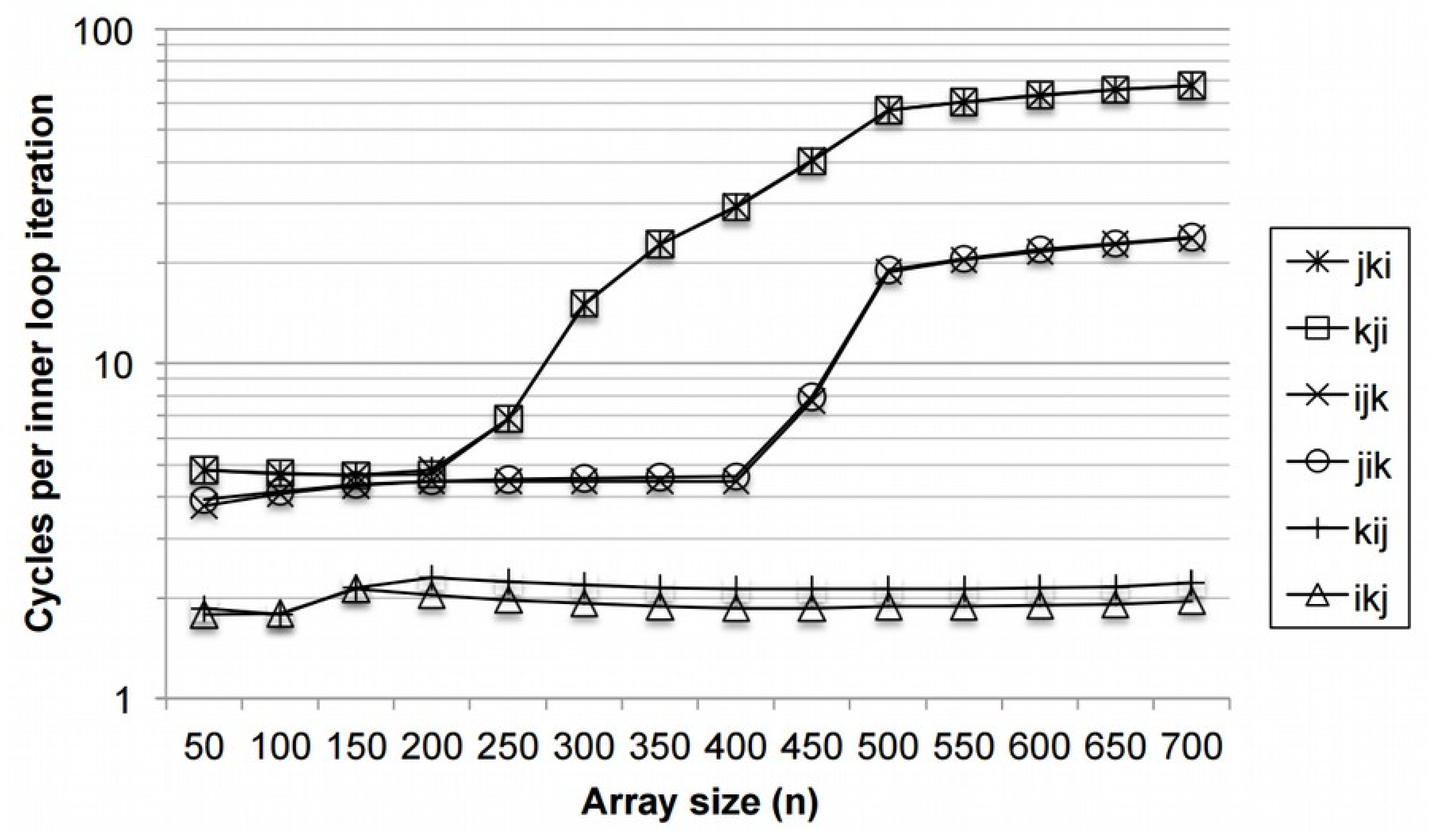
code/mem/matmult/mm.c

```
for (i = 0; i < n; i++)
    for (k = 0; k < n; k++) {
        r = A[i][k];
        for (j = 0; j < n; j++)
            C[i][j] += r*B[k][j];
    }
```

code/mem/matmult/mm.c

Figure 6.44 **Six versions of matrix multiply.** Each version is uniquely identified by the ordering of its loops.

Case study: matrix multiply



Optimization strategies

- Focus on the common cases
- Focus on the code regions that dominate runtime
- Focus on inner loops and minimize cache misses
- Favor repeated local accesses (temporal locality)
- Favor stride-1 access patterns (spatial locality)

Next time

- Virtual memory: an OS-level memory cache