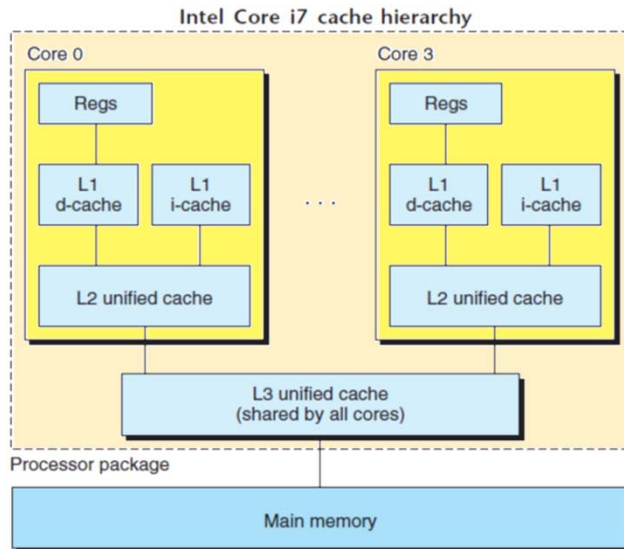
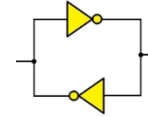


CSE320 System Fundamentals II

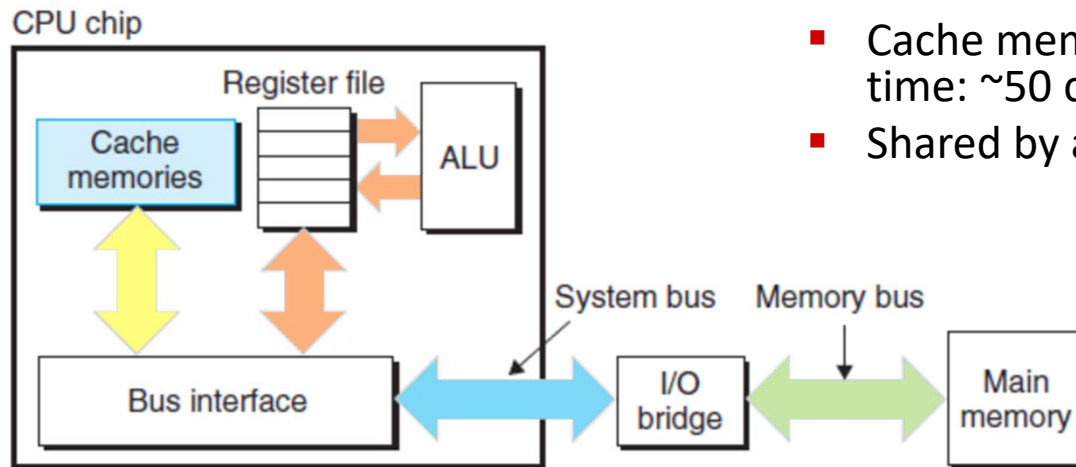
Cache Memories

YoungMin Kwon

Cache Memories

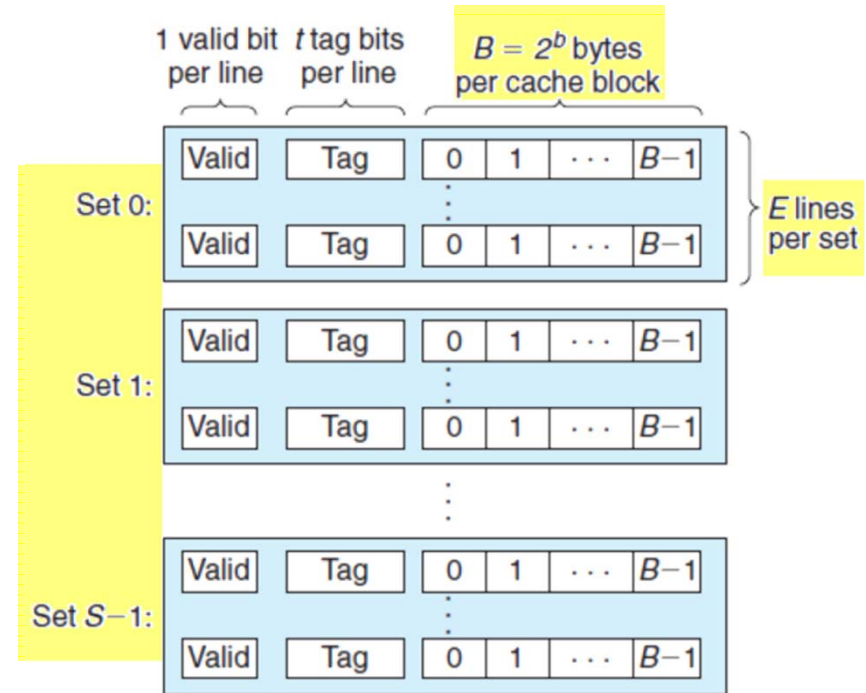


- L1 cache
 - Cache memory below registers, access time: ~ 4 cycles
 - Separate d-cache and i-cache
- L2 cache
 - Cache memory below L1 cache, access time: ~ 10 cycles
 - Unified cache: data and instructions
- L3 cache
 - Cache memory below L2 cache, access time: ~ 50 cycles
 - Shared by all cores



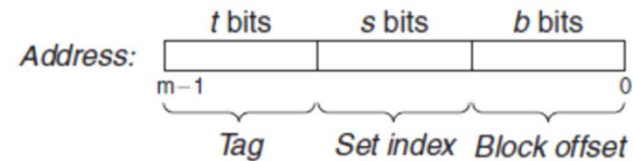
Generic Cache Memory Organization

- **S**: ($=2^s$), # of cache sets
- **E**: # of lines in a cache set
- **B**: ($=2^b$), # of bytes in a line
- **m**: memory address bits
- **Valid bit**: whether the line contains a valid data
- **t**: ($= m - (b+s)$), # of bits in a tag
- **C**: ($= B \times E \times S$), cache size



Cache size: $C = B \times E \times S$ data bytes

(a)

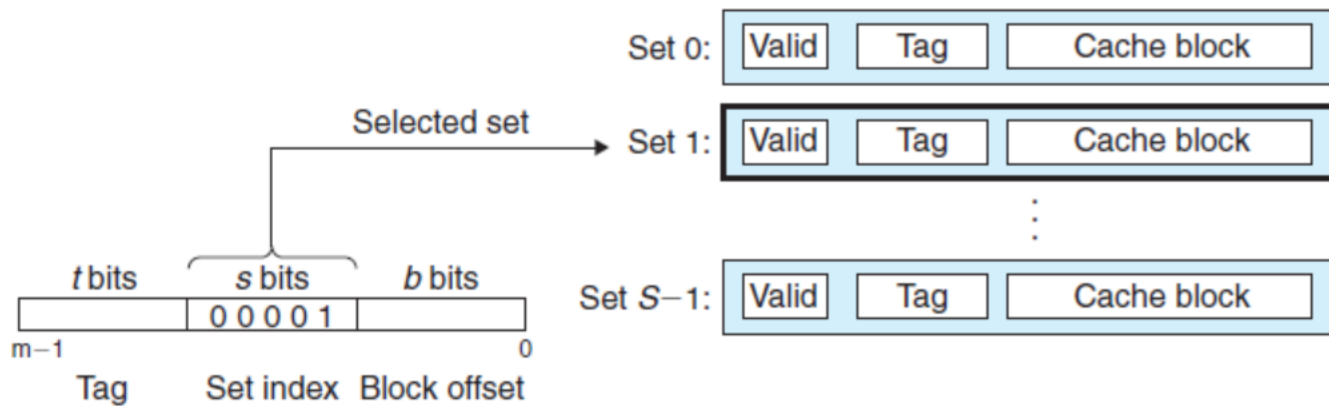


(b)

Caches

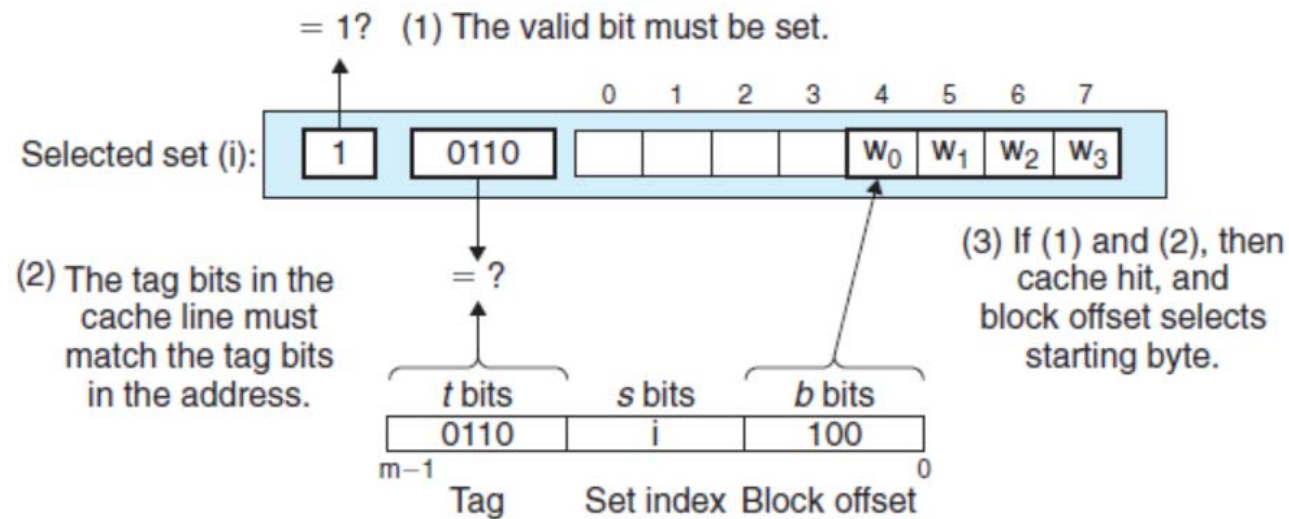
- Classes of caches by E (# of lines per set)
 - $E = 1$: **direct-mapped** cache (1 line per set)
 - $1 < E < C/B$: **set-associative** cache
 - $E = C/B$: **fully-associative** cache (1 set)
- Accessing the requested word from cache
 - Set selection
 - Line matching
 - Word extraction

Direct-Mapped Cache



- Set selection
 - Select the set using the **set index** as an index

Direct-Mapped Cache



- Line Matching
 - A word is contained in the line iff the **valid bit** is set and the **tag** of the line matches the tag of the address
- Word extraction
 - Find the word in the line indexed by the **block offset**

Direct-Mapped Cache (action)

$(S, E, B, m) = (4, 1, 2, 4)$

Address (decimal)	Address bits			Block number (decimal)
	Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Direct-Mapped Cache (action)

- Read from address 0: cache miss

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

- Read from address 1: cache hit
- Read from address 13: cache miss

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Direct-Mapped Cache (action)

- Read from address 8: cache miss

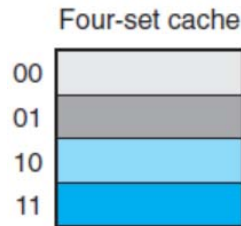
Set	Valid	Tag	block[0]	block[1]
0	1	1	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

- Read from address 0: cache miss

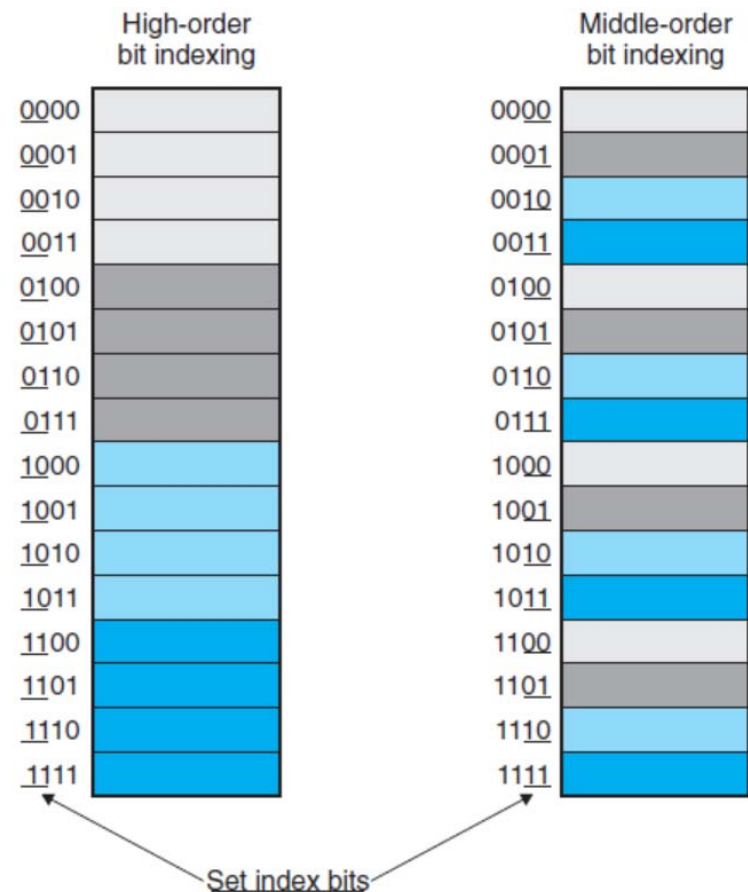
Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Why set-index with the middle bits

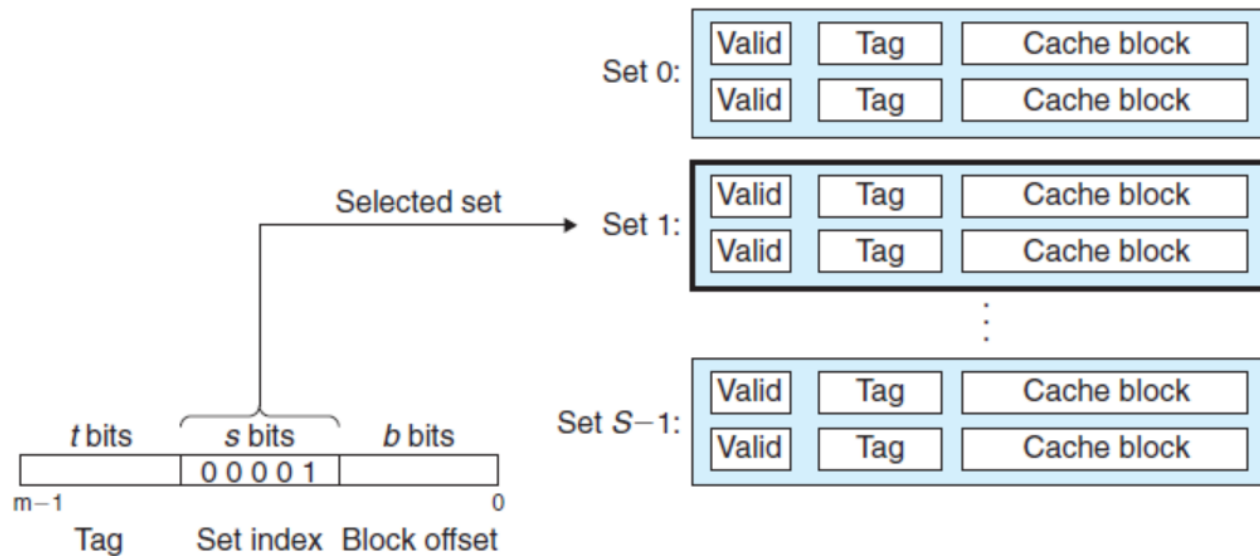
- If the **high-order bits** are used as a **set-index**, then some contiguous memory blocks will be mapped to the same cache set.



```
//example: poor spatial locality
for(i = 0; i < n; i++)
    a[i] = a[i+1];
//a[0] = a[1]
//a[1] = a[2]
//a[2] = a[3]
...
```

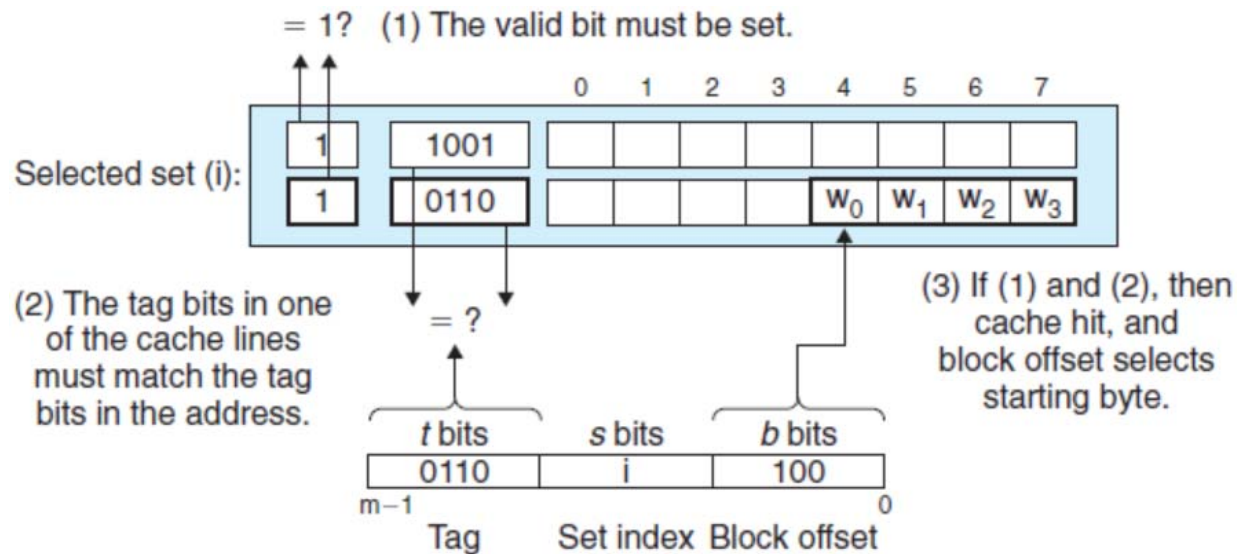


Set Associative Cache



- Set selection
 - Use the set index to select the set

Set Associative Cache



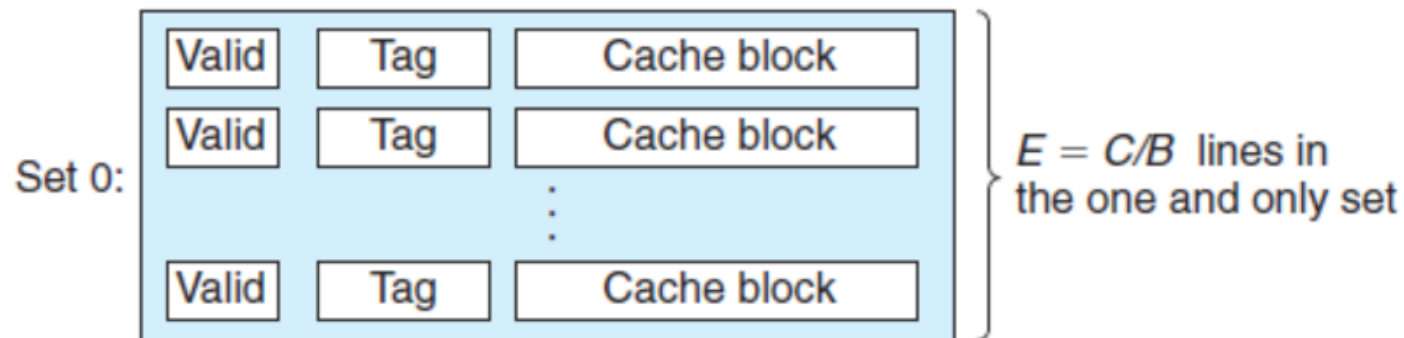
- Line Selection
 - Within the set, find the **valid line** with the **matching tag**
- Word Selection
 - Find the word in the line indexed by the **block offset**

Set Associative Cache

Line replacement on cache misses

- When a line is empty
 - Copy the block to the memory
- Otherwise, follow the replacement policy
 - Choose a line at **random**
 - Choose the least frequently used (**LFU**) line
 - Choose the least recently used (**LRU**) line

Fully Associative Cache



- No need to select a set: there is only 1 set
- **Line matching** and **word selection** work the same way as the set associative cache

Issues with **Writes**: After a **cache HIT**

- Write-through
 - **Immediately write** the word's cache block to the next lower level
 - Causing a **bus traffic** for every write
- Write-back
 - **Defers the update** as long as possible: updates the lower level only when the data is evicted
 - Needs a **dirty bit**
 - Bus traffic is reduced at the cost of additional complexities



Issues with **Writes**: After a **cache MISS**

- Write-allocate
 - Loads the block from the lower level and **updates the cache**
 - Exploits the **spatial locality**
 - Every miss results in a block transfer from the lower level
- No-write-allocate
 - Bypass the cache and write directly to the lower level

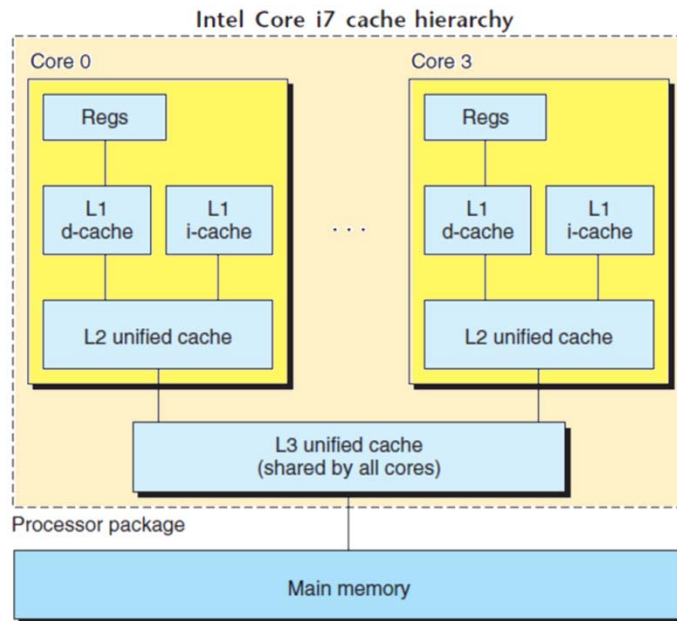


Issues with Writes

- Why write-back
 - Because of the **larger transfer time**, caches at lower level of the memory hierarchy **use write-back**
 - It exploits the **locality**
 - As the logic density increases, the complexity of write-back becomes less of an impediment
 - Write-back/write-allocate is symmetric to the way read is handled



Real Cache Hierarchy



- **i-cache**: a cache for instructions
- **d-cache**: a cache for data
- **unified-cache**: a cache for both instructions and data

Real Cache Hierarchy

Characteristics of the Intel Core i7 cache hierarchy

Cache type	Access time (cycles)	Cache size (C)	Assoc. (E)	Block size (B)	Sets (S)
L1 i-cache	4	32 KB	8	64 B	64
L1 d-cache	4	32 KB	8	64 B	64
L2 unified cache	11	256 KB	8	64 B	512
L3 unified cache	30–40	8 MB	16	64 B	8192

- What are the number of bits in a tag?
 - i7 processors have 48 physical address bits and 52 virtual address bits

Cache Performance Metric

- **Miss rate:** # of misses / # of references
- **Hit rate:** $1 - \text{miss rate}$
- **Hit time:**
 - Time to deliver a word in the **cache to the CPU**
 - Includes the times for set identification, line identification, and word selection
- **Miss penalty:**
 - Any additional time required because of a miss

Performance Impact of Cache Parameters

- Impact of **cache size**: Large cache size
 - Increases hit rate
 - Increases hit time because of H/W complexity
- Impact of **block size**: Large block size
 - Increases **spatial locality**
 - Reduces # of lines \Rightarrow decreases **temporal locality**
 - Think about two or more variables at different scopes
 - Loading large blocks \Rightarrow increases the miss penalty

Performance Impact of Cache Parameters

- Impact of **Associativity**: Increasing E
 - Decrease the **conflict misses**
 - Increases the cost and complexity \Rightarrow increased hit time
 - Complexity in choosing a victim line \Rightarrow increased miss penalty
- Impact of **Write Strategy**
 - Write-through: simpler to implement
 - Use write buffer \Rightarrow read misses are less expensive
 - Write-back: fewer transfers

Cache-Friendly Code

- Average miss count:
 - **Stride-k** reference pattern (in terms of words)
 - Block size is B
 - $\min(1, (\text{wordsize} \cdot k) / B)$ misses per loop

- Example

- Words are 4 bytes,
- Cache blocks are 4 words

```
int sumvec(int v[N])
{
    int i, sum = 0;

    for (i = 0; i < N; i++)
        sum += v[i];

    return sum;
}
```

v[i]	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7
Access order, [h]it or [m]iss	1 [m]	2 [h]	3 [h]	4 [h]	5 [m]	6 [h]	7 [h]	8 [h]

Cache-Friendly Code

- Repeated reference to **local variables** are good
 - Compiler can cache them in the **register file**
 - **Temporal locality**
- **Stride-1** reference pattern is good
 - Caches at all levels of the memory hierarchy store data as contiguous blocks
 - **Spatial locality**

Cache-Friendly Code

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

a[i][j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 [m]	2 [h]	3 [h]	4 [h]	5 [m]	6 [h]	7 [h]	8 [h]
i = 1	9 [m]	10 [h]	11 [h]	12 [h]	13 [m]	14 [h]	15 [h]	16 [h]
i = 2	17 [m]	18 [h]	19 [h]	20 [h]	21 [m]	22 [h]	23 [h]	24 [h]
i = 3	25 [m]	26 [h]	27 [h]	28 [h]	29 [m]	30 [h]	31 [h]	32 [h]

Cache-Friendly Code

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

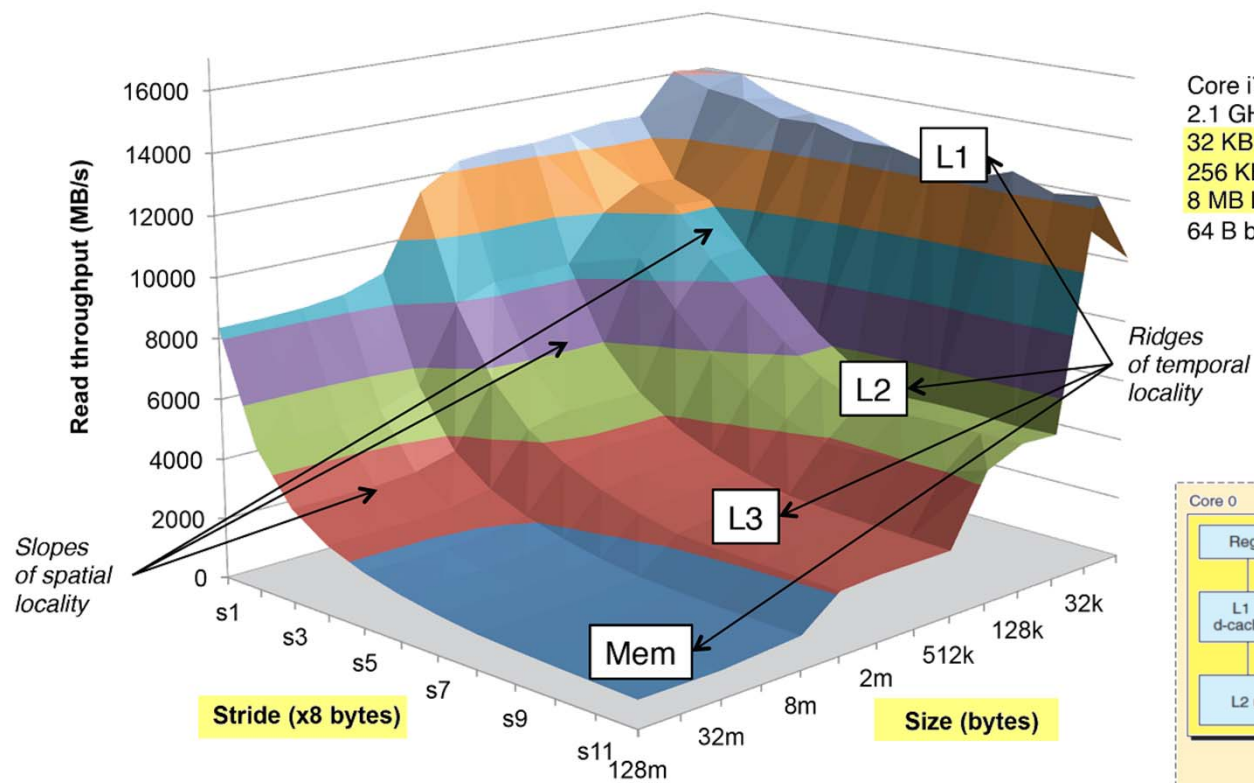
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

a[i][j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 [m]	5 [m]	9 [m]	13 [m]	17 [m]	21 [m]	25 [m]	29 [m]
i = 1	2 [m]	6 [m]	10 [m]	14 [m]	18 [m]	22 [m]	26 [m]	30 [m]
i = 2	3 [m]	7 [m]	11 [m]	15 [m]	19 [m]	23 [m]	27 [m]	31 [m]
i = 3	4 [m]	8 [m]	12 [m]	16 [m]	20 [m]	24 [m]	28 [m]	32 [m]

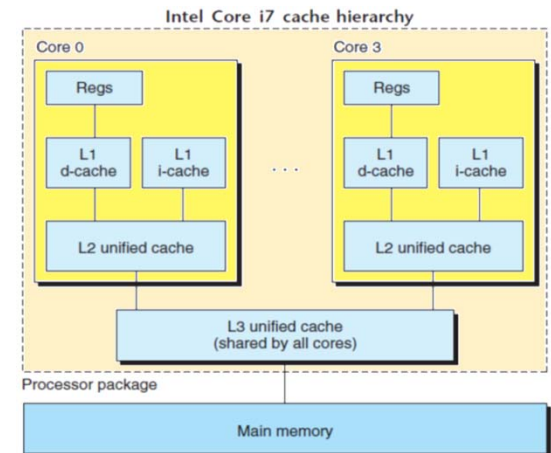
The Memory Mountain

- Read throughput (read bandwidth)
 - The rate that a program reads data from the memory system
 - Reads n bytes over a period of s seconds $\Rightarrow n/s$
- Smaller **size of data set**
 - Results in a smaller working set
 - Better **temporal locality**
- Smaller **stride**
 - Results in better **spatial locality**

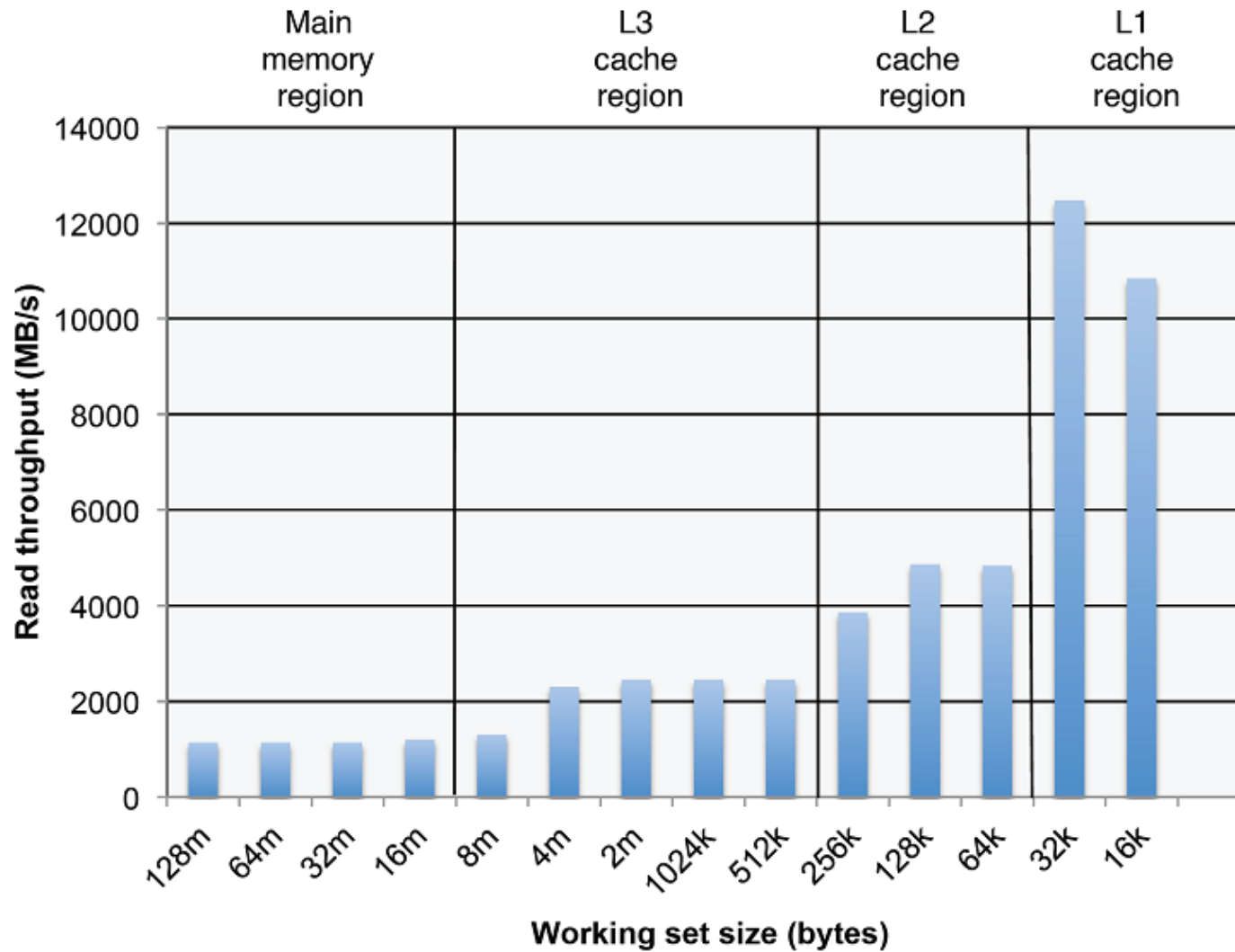
The Memory Mountain



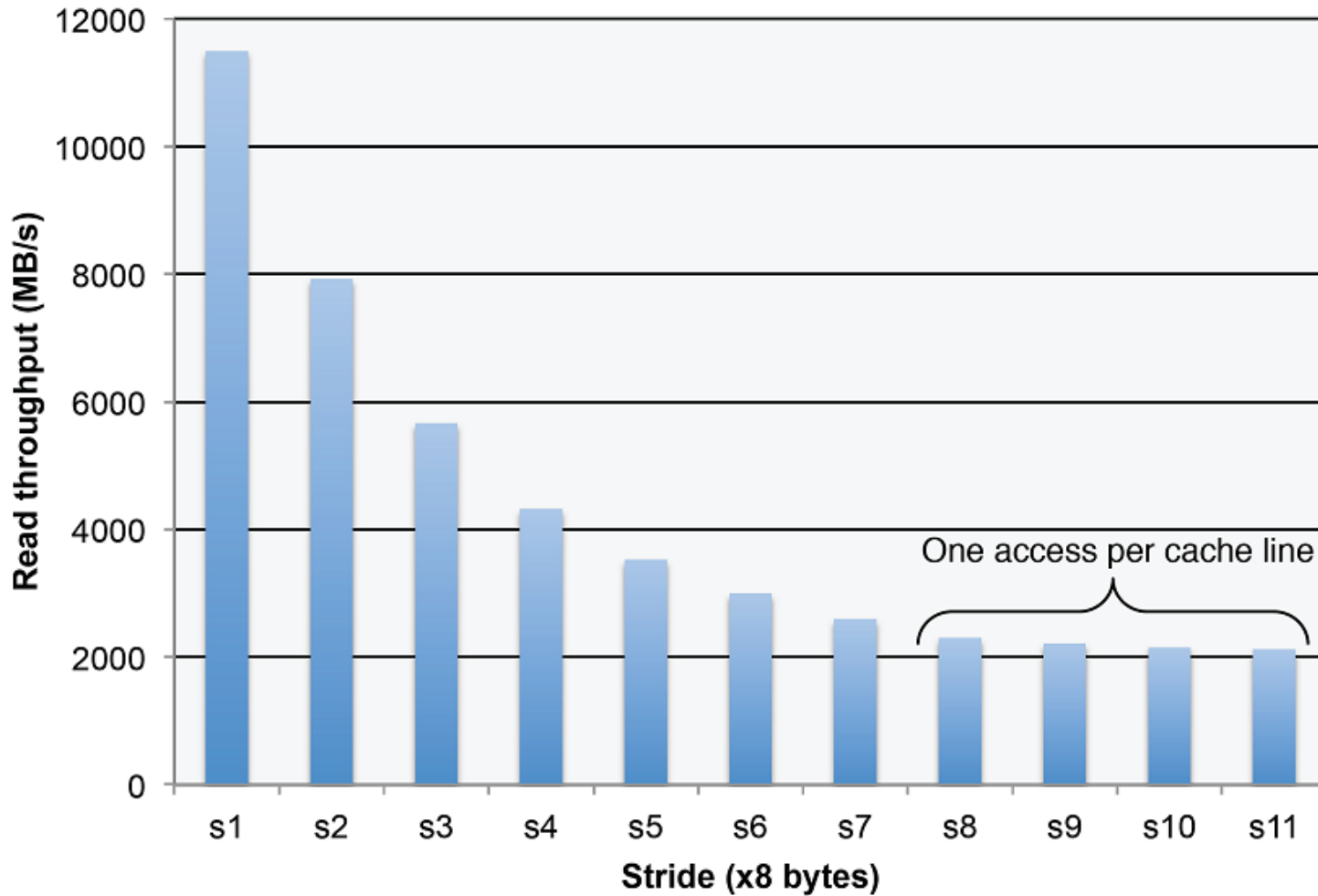
Core i7 Haswell
 2.1 GHz
 32 KB L1 d-cache
 256 KB L2 cache
 8 MB L3 cache
 64 B block size



Read throughput vs working set size



Read throughput vs stride



Exploiting Locality

- Focus on the **inner loop**
- Try to maximize the **spatial locality**
 - Reading data objects sequentially with **stride 1**
- Try to maximize the **temporal locality**
 - **Use a data object as often as possible** once it has been read from memory