CSE2431 – Lecture Topic 6
Memory Hierachy (part 2)

The Ohio State University
College of Engineering

Memory Hierarchy (Part 2)

Instructor: Luan Duong, Ph.D.

CSE 2431: Introduction to Operating Systems

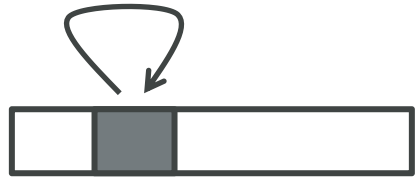
Reading: **Chap. 6** from Computer Systems: A Programmer's Perspective by Randal E. Bryant and David R. O'Halloran, 3rd edition, Pearson/Prentice Hall, 2016

Outline: Memory Hierarchy

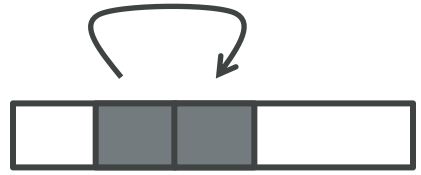
- Storage Technologies
- Locality
- Memory Hierarchy
- Cache Memories
- Writing Cache-friendly Code
- Impact of Caches on Program Performance

Locality

- **Principle of locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently
- **Temporal locality:** Recently referenced items are likely to be referenced again soon
- **Spatial locality:** Items with nearby addresses tend to be referenced close together in time
- Principle of locality has an enormous impact on the design and performance of hardware and software systems. **In general, programs with good locality run faster than programs with poor locality.**
- This principle is used by all levels of a modern computer system: hardware, OSes, and application programs.



Temporal Locality



Spatial Locality

Locality – Example

- Data references
  - Reference array elements in succession (**stride-1 reference pattern**).
  - Reference variable `sum` each iteration.
- Instruction references
  - Reference instructions in sequence.
  - Cycle through loop repeatedly.

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

Locality of Reference to Program Data (1)

```
int sum_array_rows(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;
}
```

- Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for any professional programmer.
- Question: Does this function have good locality with respect to array a?

Locality of Reference to Program Data (2)

```
int sum_array_cols(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;
}
```

- Question: Does this function have good locality with respect to array a?

Locality of Instruction Fetches

```
int sum_array_rows(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;
}
```

- Question: Does this function have good locality with respect to instructions?

Locality - Summary

- Simple rules for evaluating the locality in a program
- Programs that repeatedly reference the same variables enjoy good temporal locality.
- For programs with stride-k reference patterns, smaller strides yield better spatial locality. Programs with stride-1 reference patterns have good spatial locality. Programs that hop around memory with large strides have poor spatial locality.
- Loops have good temporal and spatial locality with respect to instruction fetches. The smaller the loop body and the greater the number of loop iterations, the better the locality.

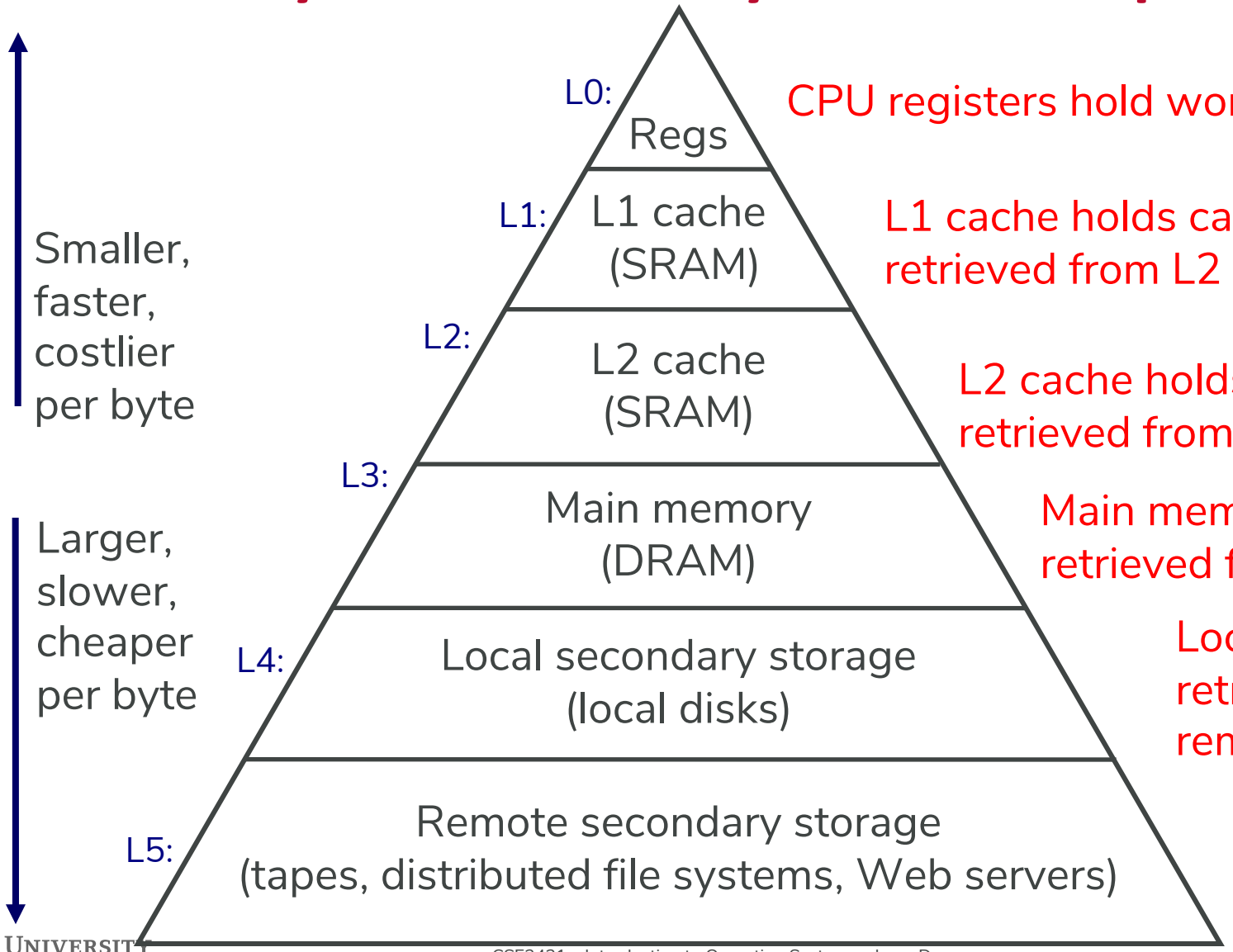
Outline: Memory Hierarchy

- Storage Technologies
- Locality
- **Memory Hierarchy**
- Cache Memories
- Writing Cache-friendly Code
- Impact of Caches on Program Performance

The Memory Hierarchy

- Fundamental properties of storage technology and computer software:
  - Storage technology: Different storage technologies have widely different access times. Faster technologies cost more per byte than slower ones and have less capacity. The gap between CPU and main memory speed is widening.
  - Computer software: Well-written programs tend to exhibit good locality.
- The complementary nature of these properties suggest an approach for organizing memory systems, knows as a memory hierarchy.

The Memory Hierarchy – Example



CPU registers hold words retrieved from L1 cache

L1 cache holds cache lines retrieved from L2 cache

L2 cache holds cache lines retrieved from main memory

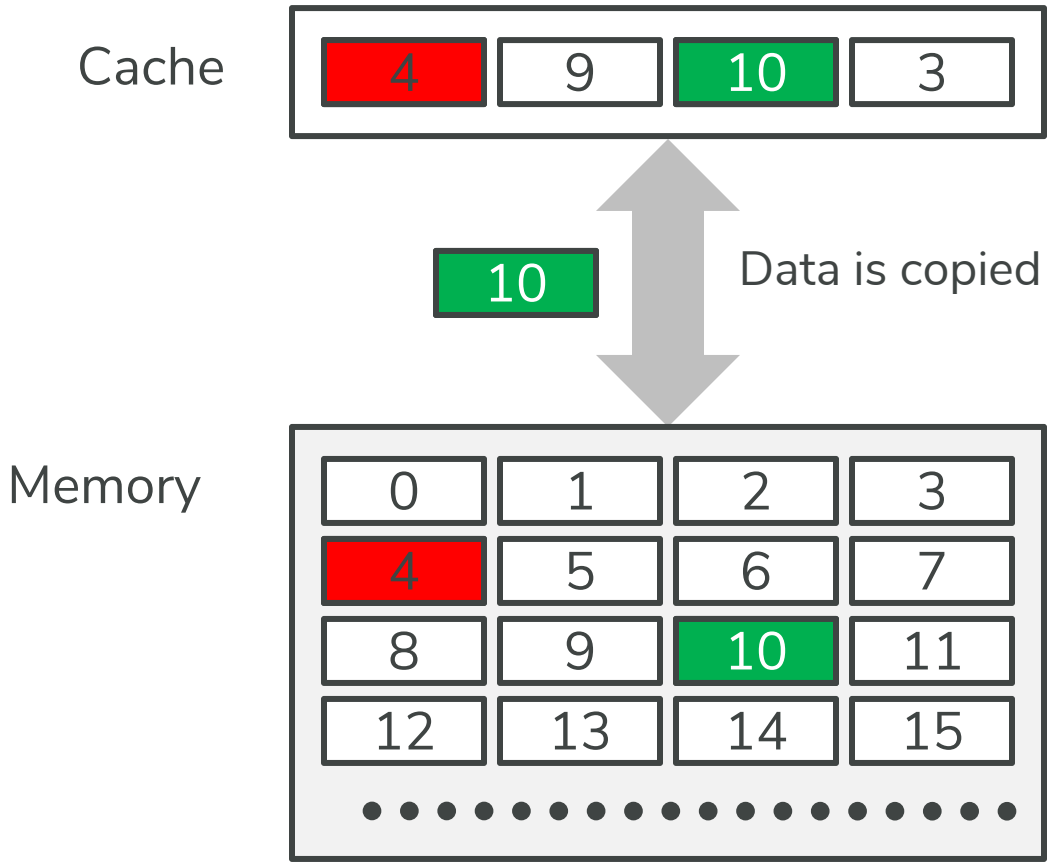
Main memory holds disk blocks retrieved from local disks

Local disks hold files retrieved from disks on remote network servers

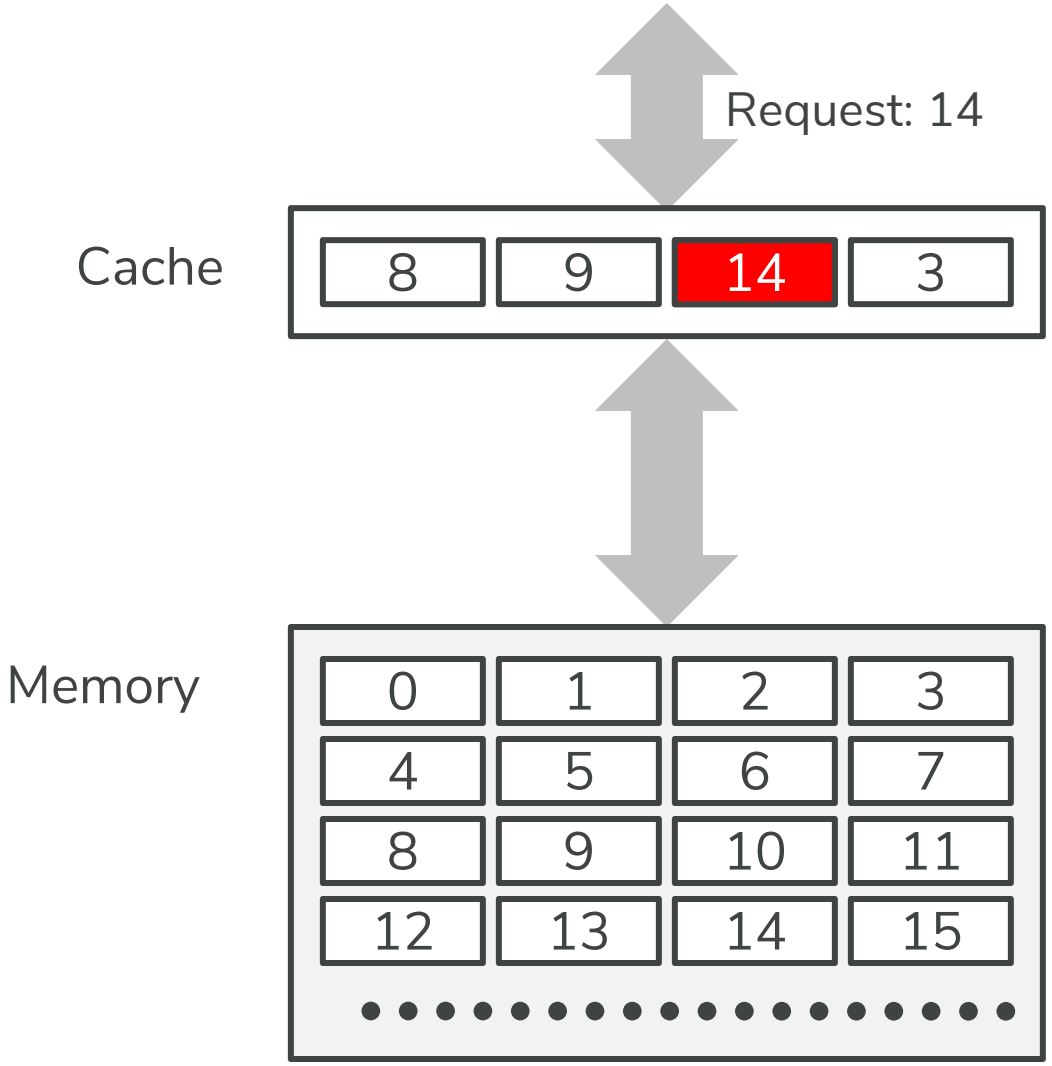
Memory Hierarchy – Caching

- A cache is a small, fast storage device that acts as a staging area for the data objects stored in a larger, slower device.
- The central idea is that for each level k in the memory hierarchy, the faster and smaller storage device serves as a cache for the larger and slower storage devices at level ***k + 1***.
- If a program finds a needed data object from level ***k + 1*** in level ***k*** then we have a **cache hit**. Otherwise, we have a **cache miss**, and the data must be brought to level k from level ***k + 1***.

General Cache Concepts



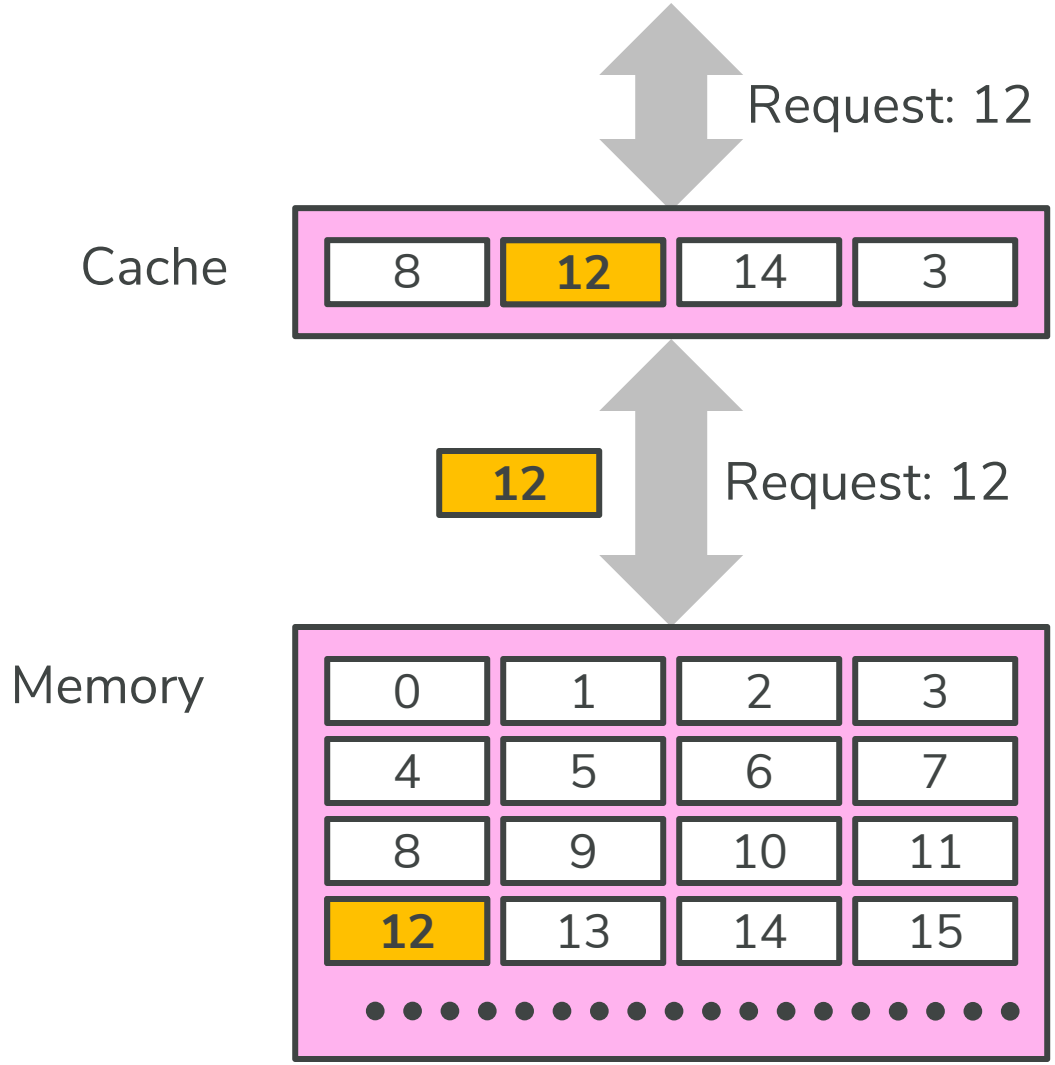
General Cache Concepts: **Cache Hit**



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: **Cache Miss**



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache
- **Placement policy:** determines where b goes
- **Replacement policy:** determines which block gets evicted (victim)

Does Cache Miss have type? Yes!

- **Cold miss:**
  - Cache at level k is empty. Temporary situation that resolves itself when repeated accesses cause the cache to ‘warm up’
- **Conflict miss:**
  - Most caches limit the blocks at ***k + 1*** to a small subset (possibly only one) position at level ***k*** (e.g., block *i* restricted to (***i* mod 4**))
  - Cache at level ***k*** is large enough but needed blocks map to the same position, for instance, blocks 0, 4, 8, 12, 16, … mapping to 0 using (i mod 4)
- **Capacity miss:**
  - Set of active blocks at ***k + 1*** larger than cache.

Examples of Caching in the Hierarchy

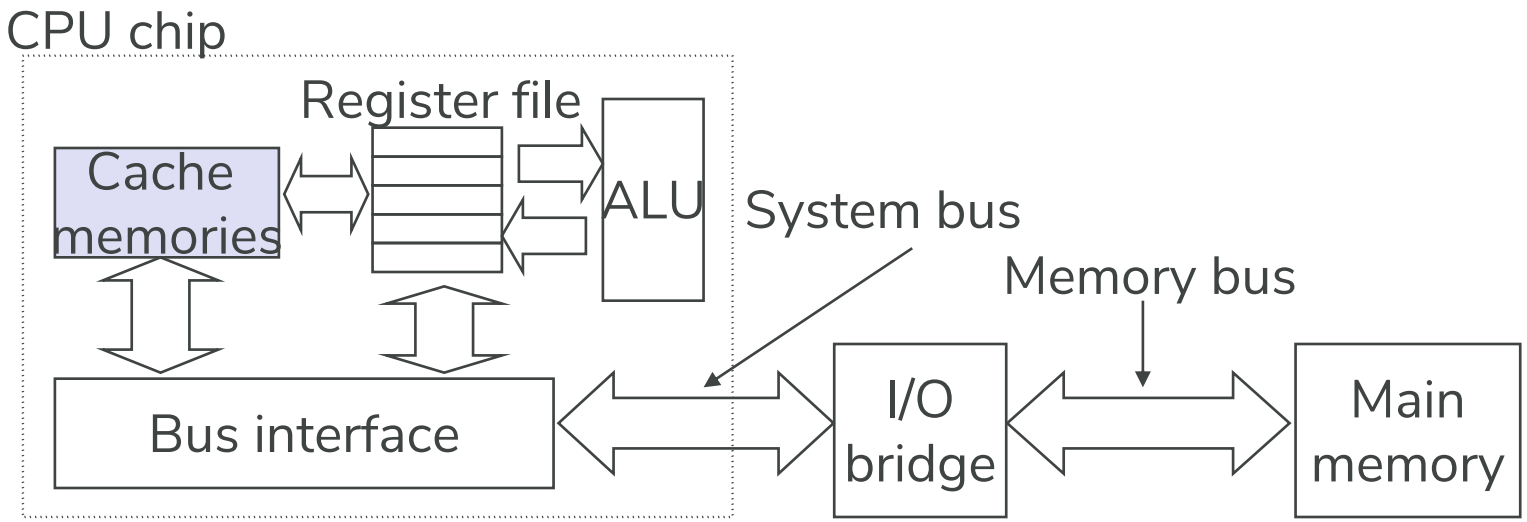
Cache Type	What is Cached?	Where's it Cached?	Latency (cycles)	Managed By
Registers	4–8-byte words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-byte blocks	On-Chip L1	1	Hardware
L2 cache	64-byte blocks	On/Off-Chip L2	10	Hardware
Virtual memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Outline: Memory Hierarchy

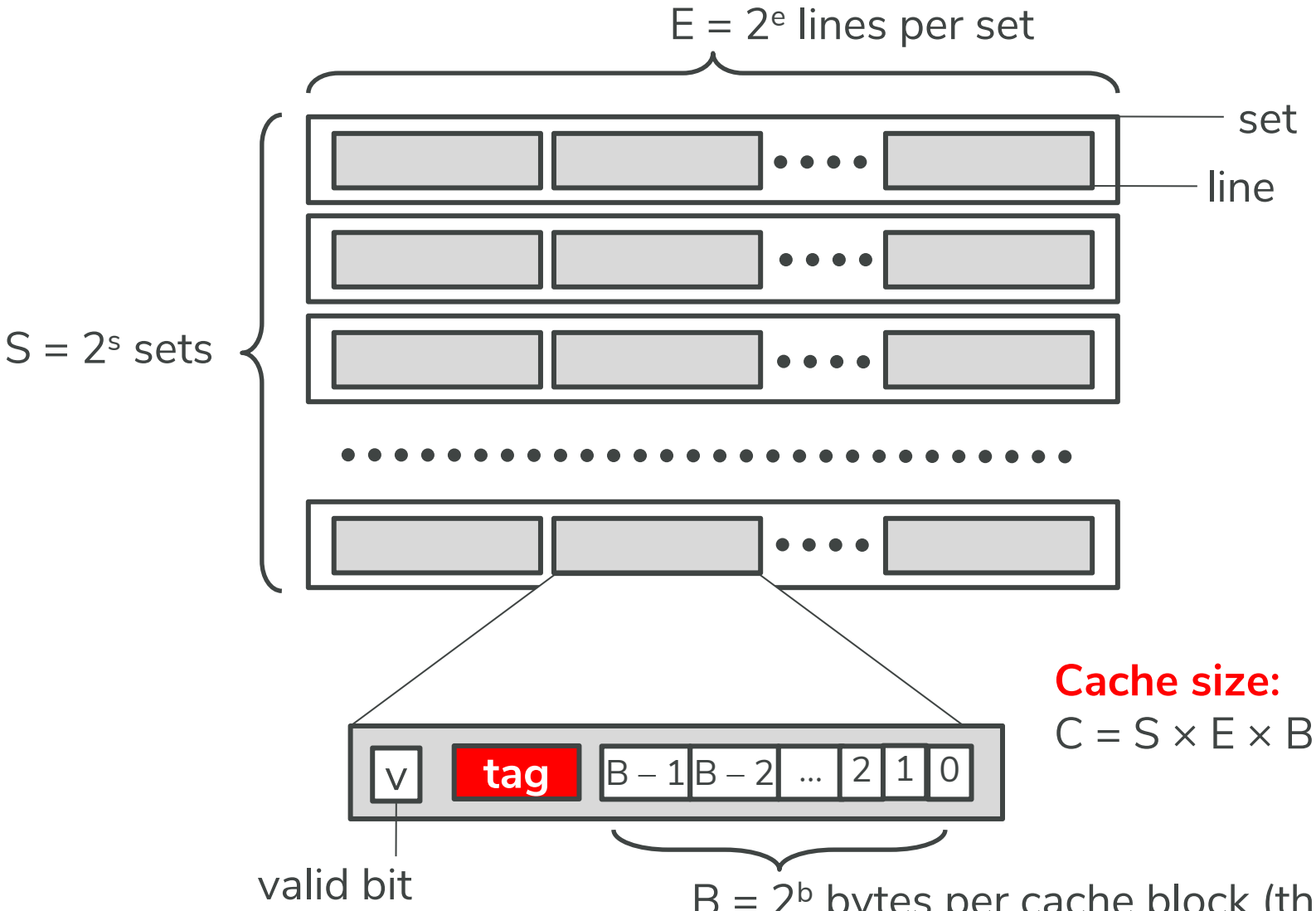
- Storage Technologies
- Locality
- Memory Hierarchy
- **Cache Memories**
- Writing Cache-friendly Code
- Impact of Caches on Program Performance

Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:



General Cache Organization



Note: Data blocks are numbered from the **most significant bit (MSB)** to the **least significant bit (LSB).** The block offset starts at the beginning of the block and moves over the specified amount.

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

Steps of a Cache Request

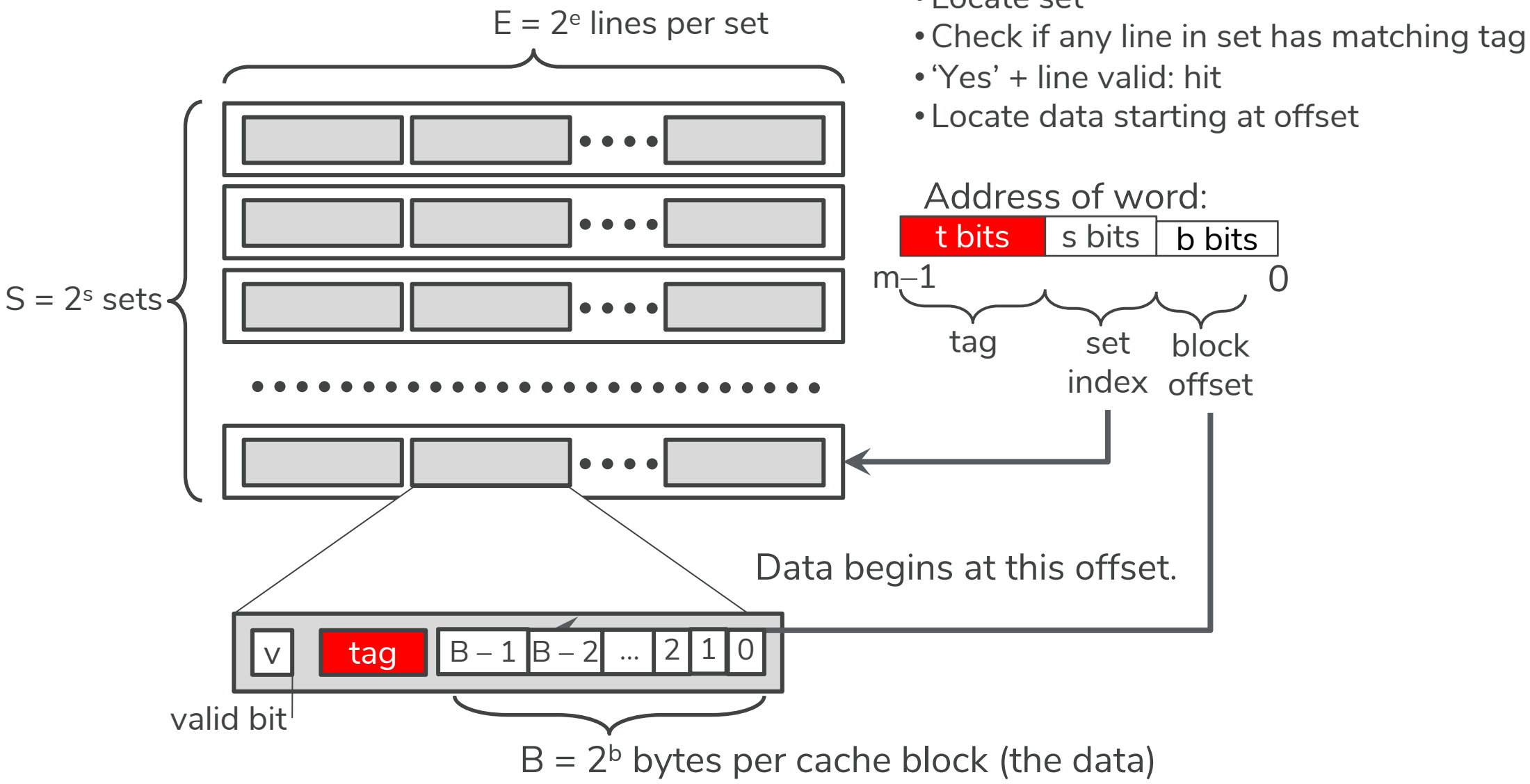
- Given a request for the word w, the address for w is used to determine the following;
  - 1. Set Selection: Determine set within cache
  - 2. Line Matching: Determine line within specific set
  - 3. Word Extraction: Extract word from cache and return it to CPU

Cache Terminology

- **Block:** fixed size packet of info that moves back and forth between a cache and main memory (or a lower-level cache)
- **Line:** a container in a cache that stores a block as well as other info such as the valid bit and the tag bits
- **Set:** collection of one of more lines. Sets in direct-mapped caches consist of a single line. Sets in set associative and fully associative caches consist of multiple lines.

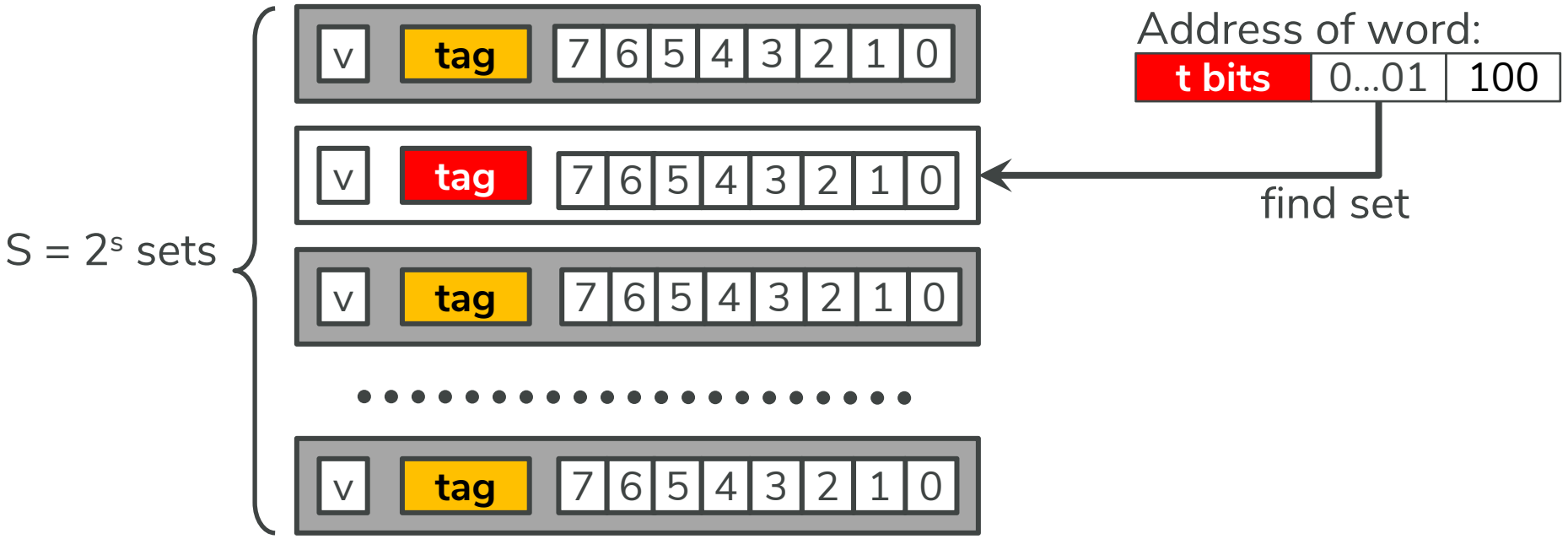
Cache Read

- Locate set
- Check if any line in set has matching tag
- 'Yes' + line valid: hit
- Locate data starting at offset



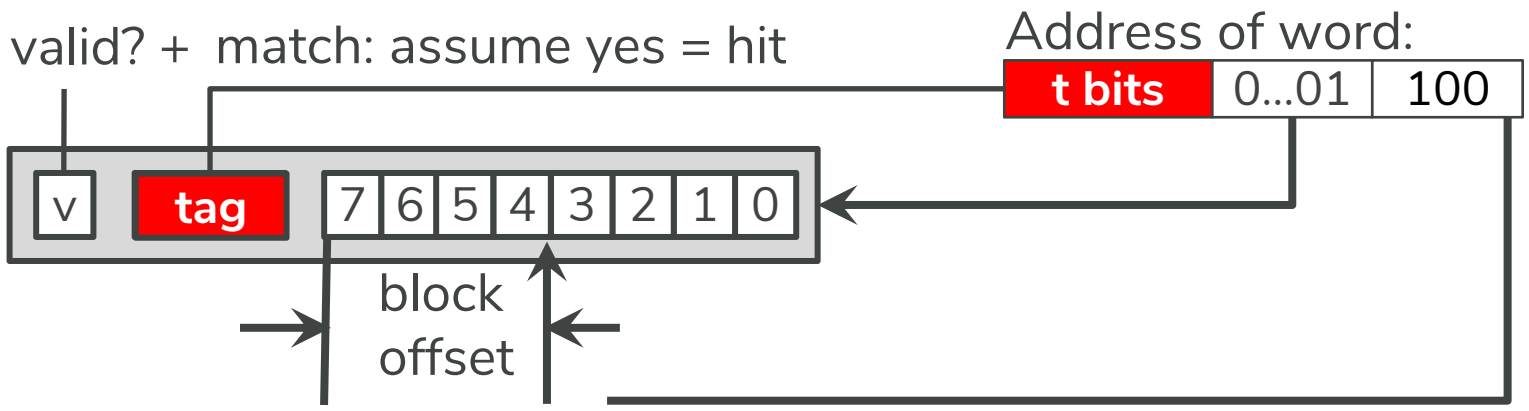
Example: Direct-Mapped Cache (E=1) (1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



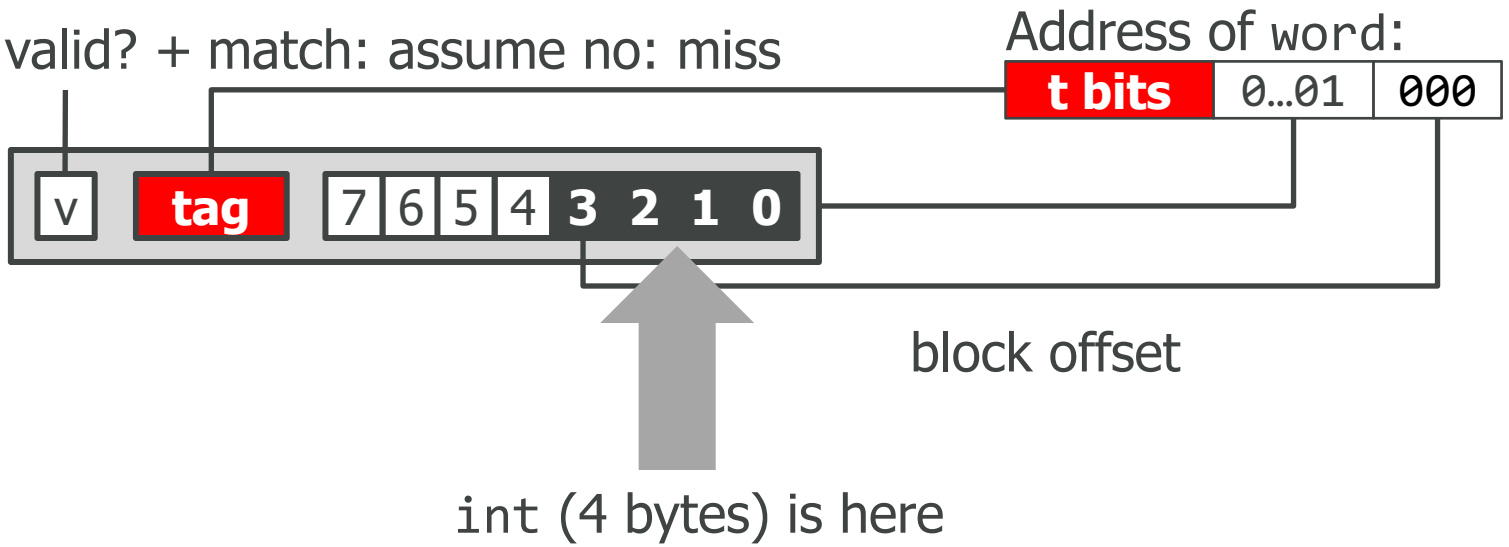
Example: Direct-Mapped Cache (E=1) (2)

Direct mapped: One line per set
Assume: cache block size 8 bytes



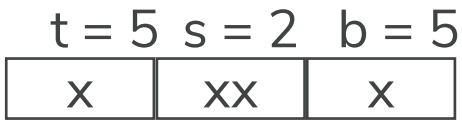
Example: Direct-Mapped Cache (E=1) (3)

Direct mapped: One line per set
Assume: cache block size 8 bytes



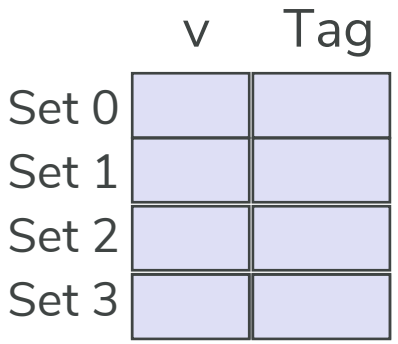
No match: old line is evicted and replaced

Direct-Mapped Cache: An Example

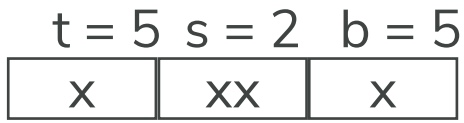


128B Cache, 12-bit address
Direct-mapped, 32B block

offset bits = $\log_2 32 = 5$
sets = 128B / 32B = 4 sets (so #set bits or #index bits = $\log_2 4 = 2$
tag bits = 12-5-2 = 5
Example: 0x060 → Binary: **0000 0**|**11**|**0 0000**
tag. **set/index offset**

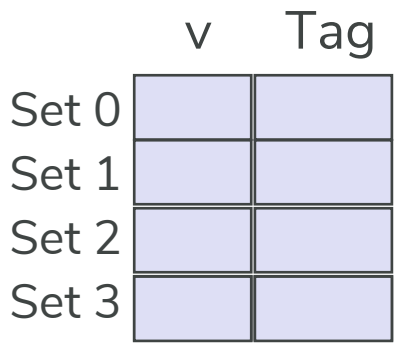


Direct-Mapped Cache: An Example



128B Cache, 12-bit address
Direct-mapped, 32B block

Addr	Tag	Set	Offset
0x070	00000	11	10000
0x080	00001	00	00000
0x068	00000	11	01000
0x190	00011	00	10000
0x084	00001	00	00100
0x178	00010	11	11000
0x08C	00001	00	01100
0xF00	11110	00	00000
0x064	00000	11	00100



Direct-Mapped Cache: Another Example

t = 1	s = 2	b = 1
x	xx	x

M = 16-byte addresses, B = 2 bytes/block,
S = 4 sets, E = 1 block/set

Address trace (reads, one byte per read):

0	[$0\underline{00}0_2$],	miss
1	[$0\underline{00}1_2$],	hit
7	[$0\underline{11}1_2$],	miss
8	[$1\underline{00}0_2$],	miss
0	[$0\underline{00}0_2$]	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

A Higher-Level Example

```
int sum_array_rows(double a[16][16]) {
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16]) {
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

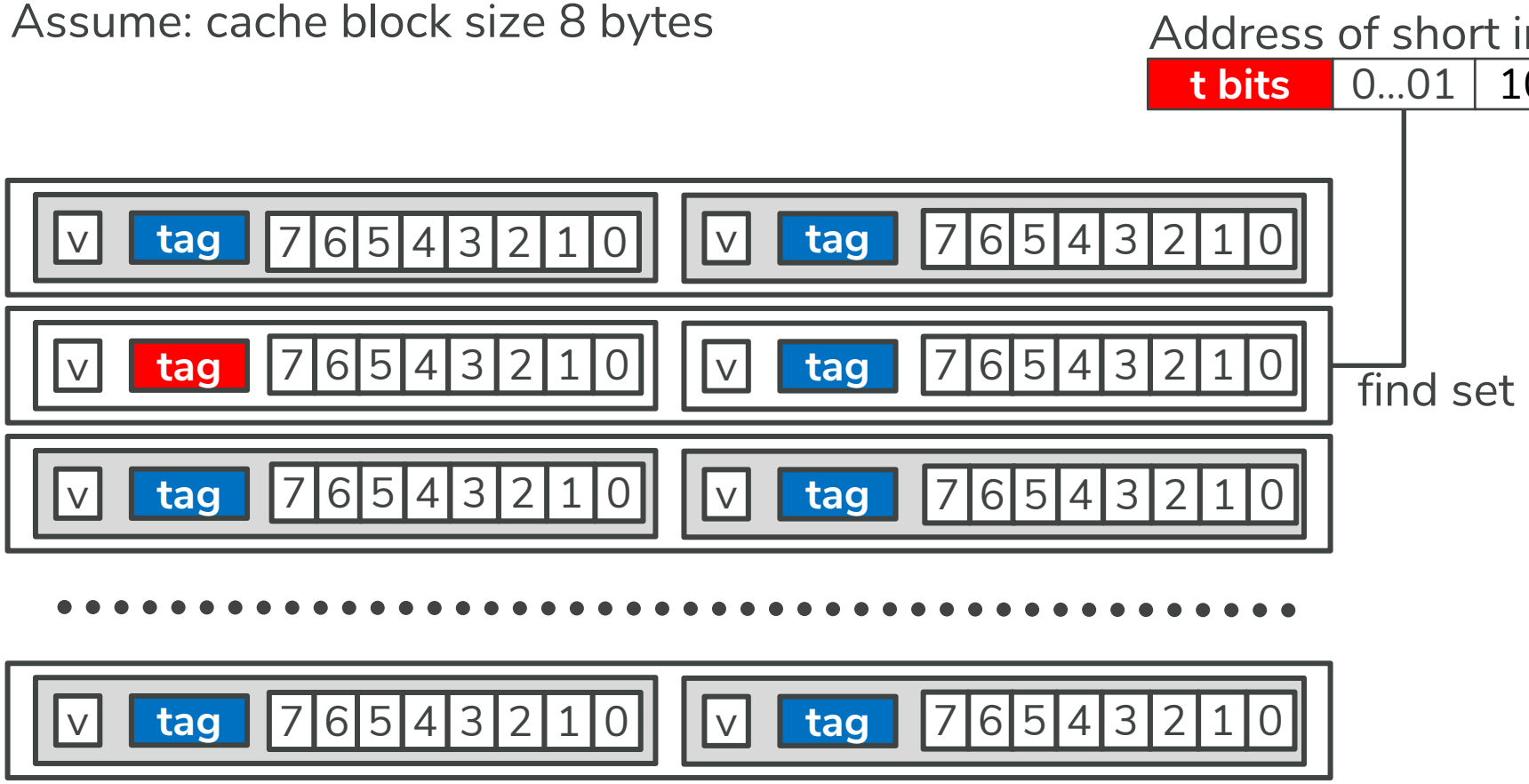
Assume: cold (empty) cache, a[0][0] goes here

blackboard

Example: E-way Set Associative Cache (E=2)

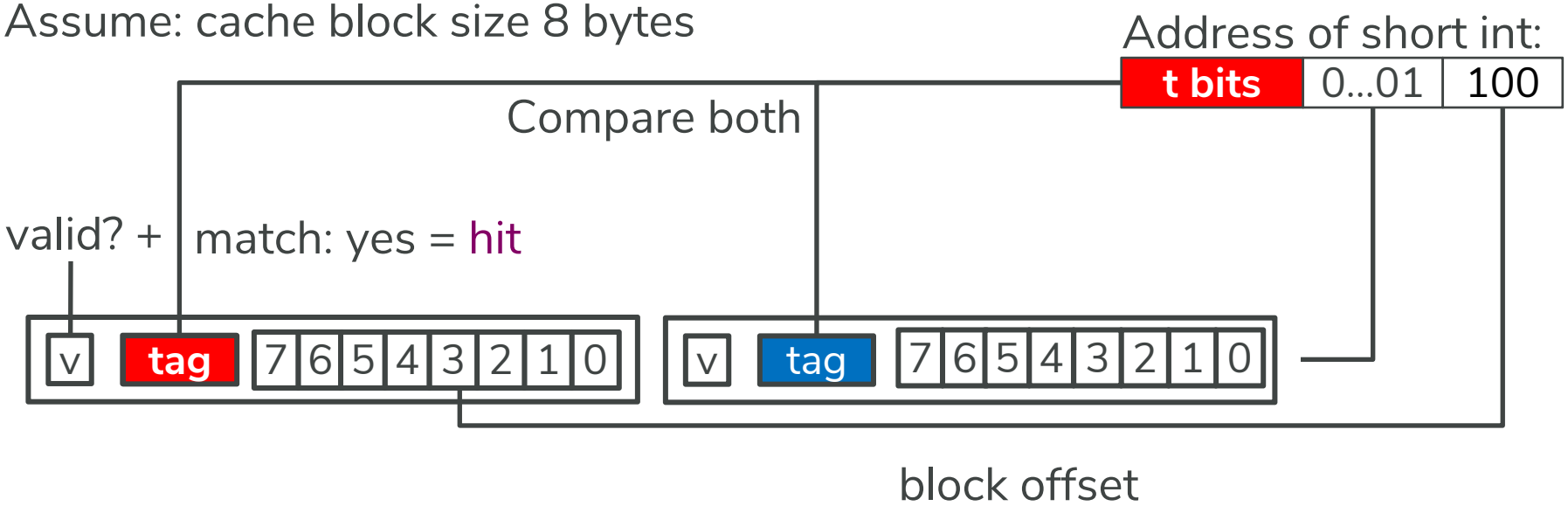
E = 2: two lines per set

Assume: cache block size 8 bytes



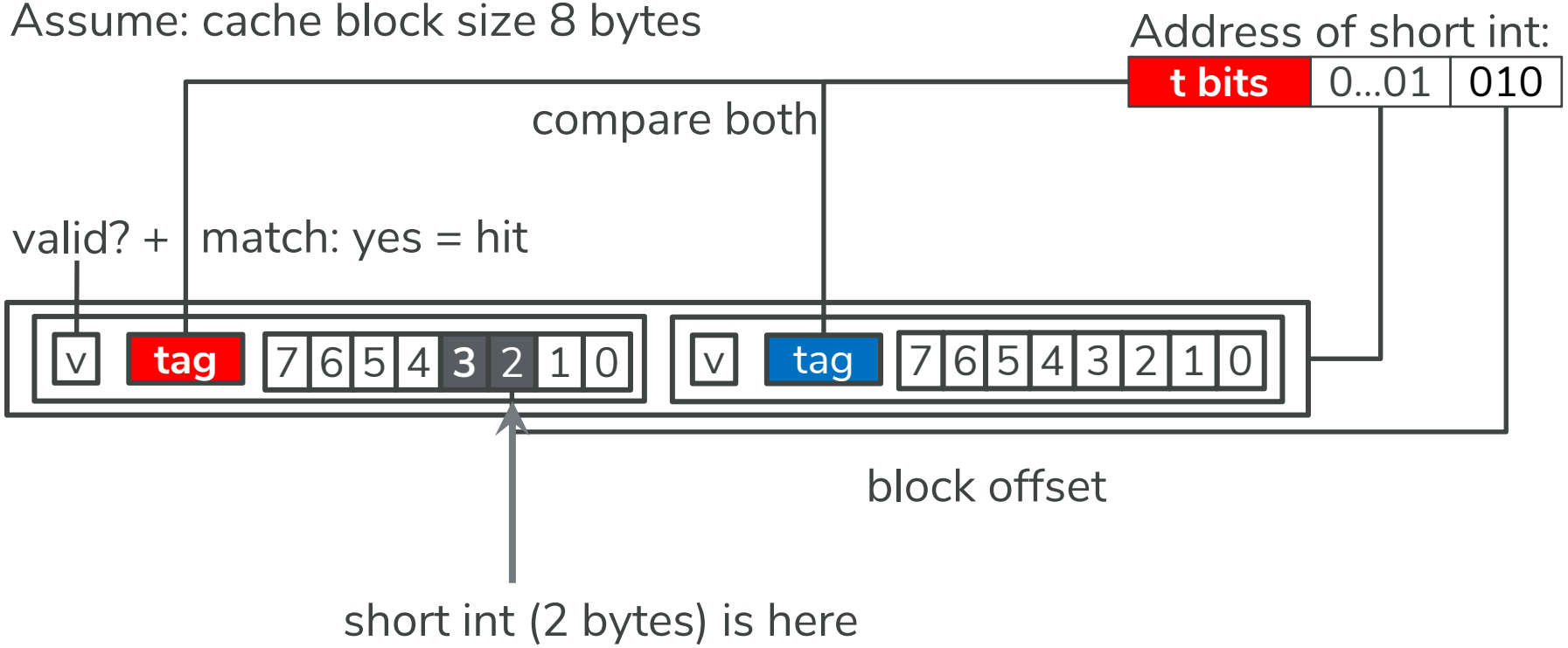
Example: E-way Set Associative Cache (E=2)

E = 2: two lines per set
Assume: cache block size 8 bytes



Example: E-way Set Associative Cache (E=2)

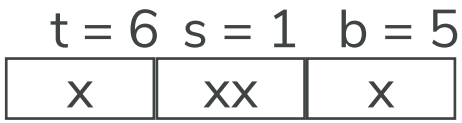
E = 2: two lines per set
Assume: cache block size 8 bytes



No match:

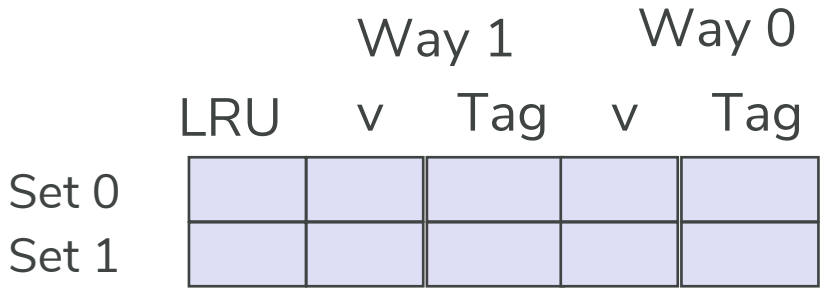
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

E-way Mapped Cache: Example 1 (E=2)



128B Cache, 12-bit address
2-way set-associative, 32B block; Assumption of **LRU for replacement**

offset bits = $\log_2 32 = 5$
sets = 128B / (32B * **2**) = 2 sets (so #set bits or #index bits = $\log_2 2 = 1$)
tag bits = 12-5-1 = 6
Example: 0x060 → Binary: **0000 01|1|0 0000**
tag. set/index offset

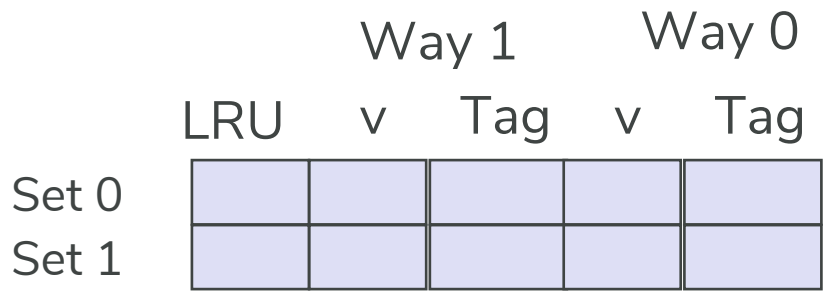


E-way Mapped Cache: Example 1 (E=2)

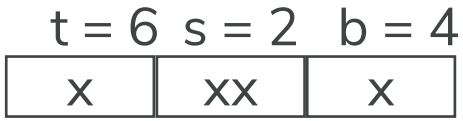
t = 6	s = 1	b = 5
x	xx	x

128B Cache, 12-bit address
2-way set-associative, **32B** block; Assumption of **LRU for replacement**

Addr	Tag	Set	Offset
0x070	000001	1	10000
0x080	000010	0	00000
0x068	000001	1	01000
0x190	000110	0	10000
0x084	000010	0	00100
0x178	000101	1	11000
0x08C	000010	0	01100
0xF00	111100	0	00000
0x064	000001	1	00100

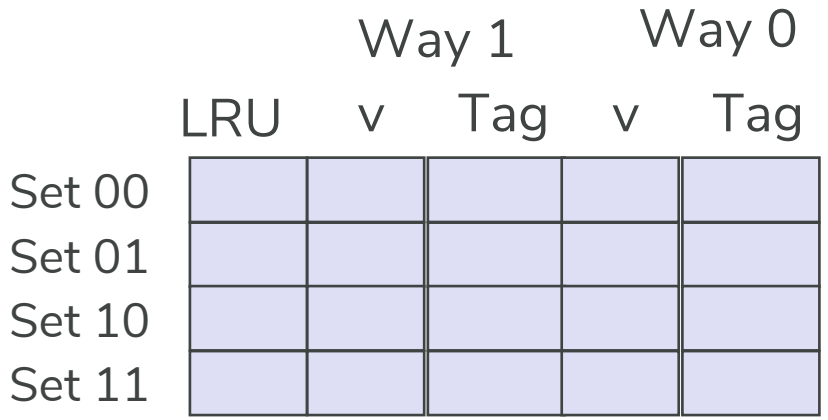


E-way Mapped Cache: Example 2 (E=2)



128B Cache, 12-bit address
2-way set-associative, 16B block; Assumption of **LRU for replacement**

offset bits = $\log_2 16$ = **4** bits
sets = 128B / (16B * **2**) = **4** sets (so #set bits or #index bits = $\log_2 4$ =2)
tag bits = 12-4-2 = 6 tag bits
Example: 0x060 → Binary: **0000 01|10| 0000**
tag. set/index offset

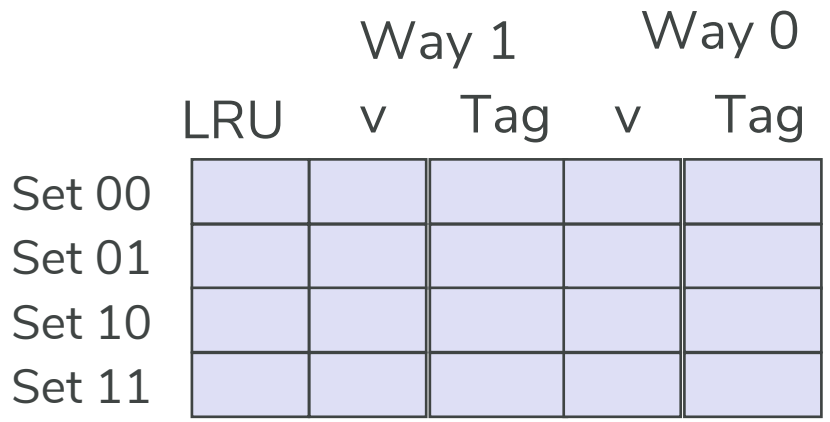


E-way Mapped Cache: Example 2 (E=2)

t = 6	s = 2	b = 4
x	xx	x

128B Cache, 12-bit address
2-way set-associative, **16B** block; Assumption of **LRU for replacement**

Addr	Tag	Set	Offset
0x070	000001	11	0000
0x080	000010	00	0000
0x068	000001	10	1000
0x190	000110	01	0000
0x084	000010	00	0100
0x178	000101	11	1000
0x08C	000010	00	1100
0xF00	111100	00	0000
0x064	000001	10	0100



2-Way Set Associative Cache Simulation

t = 2	s = 1	b = 1
xx	x	x

M = 16-byte addresses, B = 2 bytes/block,
S = 2 sets, E = 2 blocks/set

Address trace (reads, one byte per read):

0	[00$\underline{0}$0$_2$],	miss
1	[00$\underline{0}$1$_2$],	hit
7	[01$\underline{1}$1$_2$],	miss
8	[10$\underline{0}$0$_2$],	miss
0	[00$\underline{0}$0$_2$]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

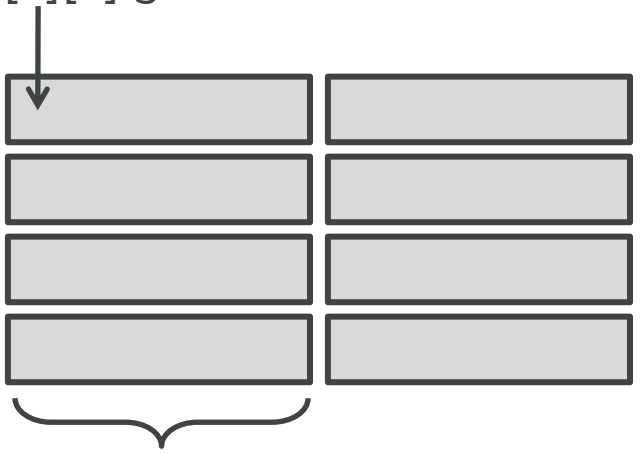
High-Level Example (2)

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache,
a[0][0] goes here

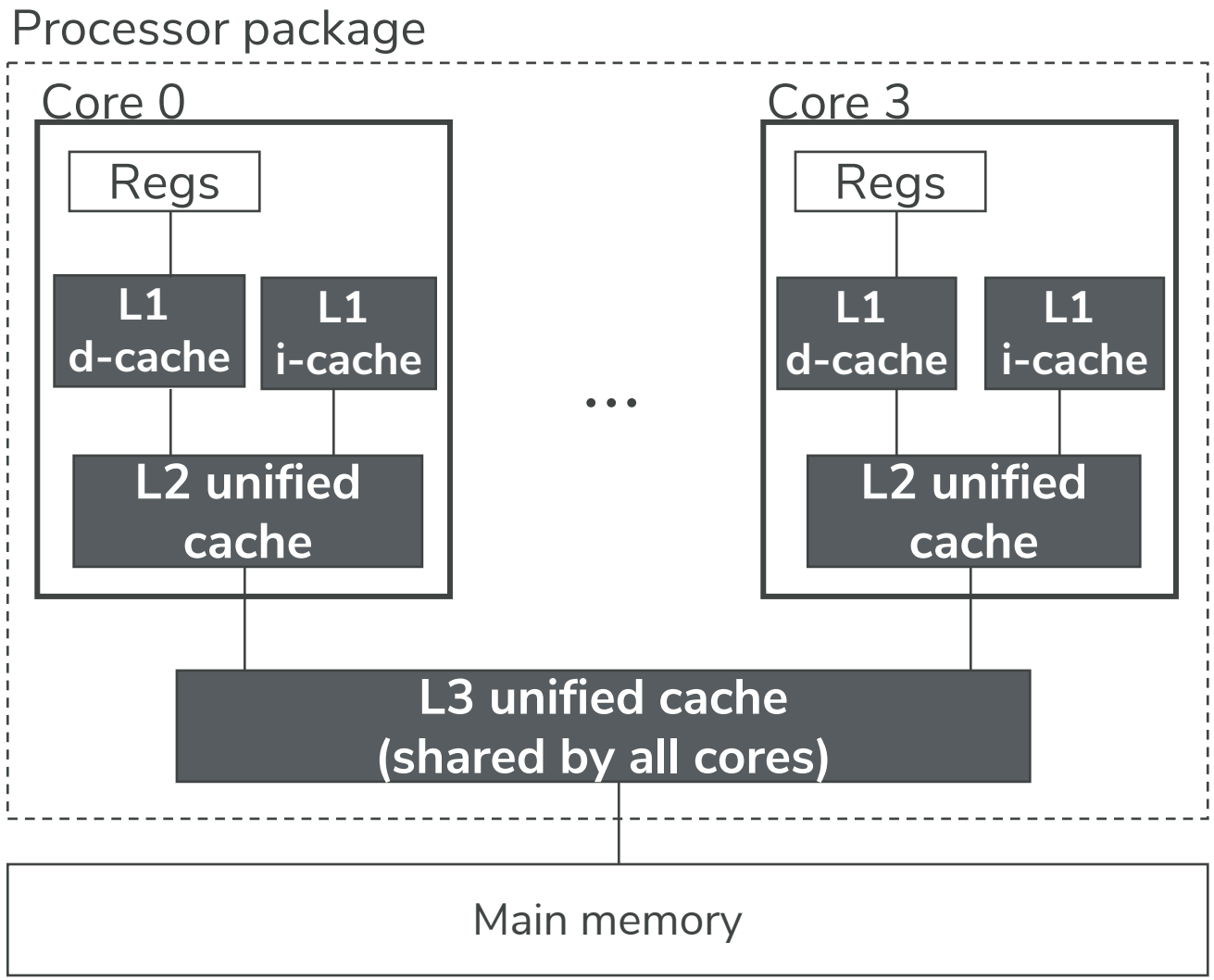


blackboard

How can we handle Writes?

- Hit
  - **Write-through**: write immediately to memory
  - **Write-back**: wait and write to memory when line is replaced (need a dirty bit to see if line is different from memory or not)
    - Nested loop structure
- Miss
  - **Write-allocate**: load into cache, update line in cache (good if more writes to the location follow)
  - **No-write-allocate**: writes immediately to memory
- Typical
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

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: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

Performance Metrics

Cache performance is evaluated with several metrics:

- ***Miss Rate:*** Fraction of memory references during execution of program, or part thereof, that miss (#misses/#references) = 1 – hit rate. Usually 3%–10% for L1, < 1% for L2.
- ***Hit Rate:*** The fraction of memory references that hit. = 1 – miss rate
- ***Hit Time:*** The time to deliver a word in the cache to the CPU, including time for set selection, line identification, and word selection. Several clock cycles for L1, 5–20 cycles for L2.
- ***Miss Penalty:***
  - Any additional time required because of a miss.
  - Penalty for L1 served from L2 is ~10 cycles; from L3, ~40 cycles; and from main memory, 100 cycles.

Some Insights… (1)

- ***99% Hits is Twice as Good as 97%:***
  - Consider: each hit time (1 cycle), miss penalty (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.
- ***Impact of Cache Size:***
  - On one hand, a larger cache will tend to increase the hit rate. On the other hand, it's harder to make larger memories run faster. As a result, larger caches tend to increase hit time.

Some Insights… (2)

- ***Impact of Block Size:***
  - Larger blocks can help increase hit rate by exploiting spatial locality; however, larger blocks imply a smaller number of cache lines, which hurt hit rate with more temporal locality than spatial. Usually blocks are 32–64 bytes

Some Insights… (3)

Impact of Associativity:

- **Higher associativity (larger values of *E*)** decrease the vulnerability of the cache to thrashing due to conflict misses.
- **Higher associativity** expensive to implement and hard to make fast. It requires more tag bits per line, additional LRU state bits per line, and additional control logic. It can increase hit time and increase miss penalty due to increased complexity.
- **Trade-off between hit time and miss penalty**. Intel Core i7 systems: L1 and L2 are 8-way, L3 is 16-way.

Outline: Memory Hierarchy

- Storage Technologies
- Locality
- Memory Hierarchy
- Cache Memories
- **Writing Cache-friendly Code**
- Impact of Caches on Program Performance

Writing Cache-Friendly Code

- Programs with **better locality** tend to have lower miss rates and programs with lower miss rates will tend to run faster than programs with higher miss rates.
- Good programmers should always try to write code that is cache friendly, in the sense that it has good locality.

Writing Cache-Friendly Code (2)

- Make the **common case go fast**. Programs often spend most of their time in a few core functions. These functions often spend most of their time in a few loops. So focus on the inner loops of the core function and ignore the rest.
- Minimize the number of cache misses for each inner loop. Good programmers should always try to write code that is cache friendly, in the sense that it has good locality.

Impact of Caches on Program Performance

- The memory mountain
- Rearranging loops to improve spatial locality
- Using blocking to improve temporal locality

Impact of Caches on Program Performance

Every computer has a unique memory mountain that characterizes the capabilities of its memory.

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

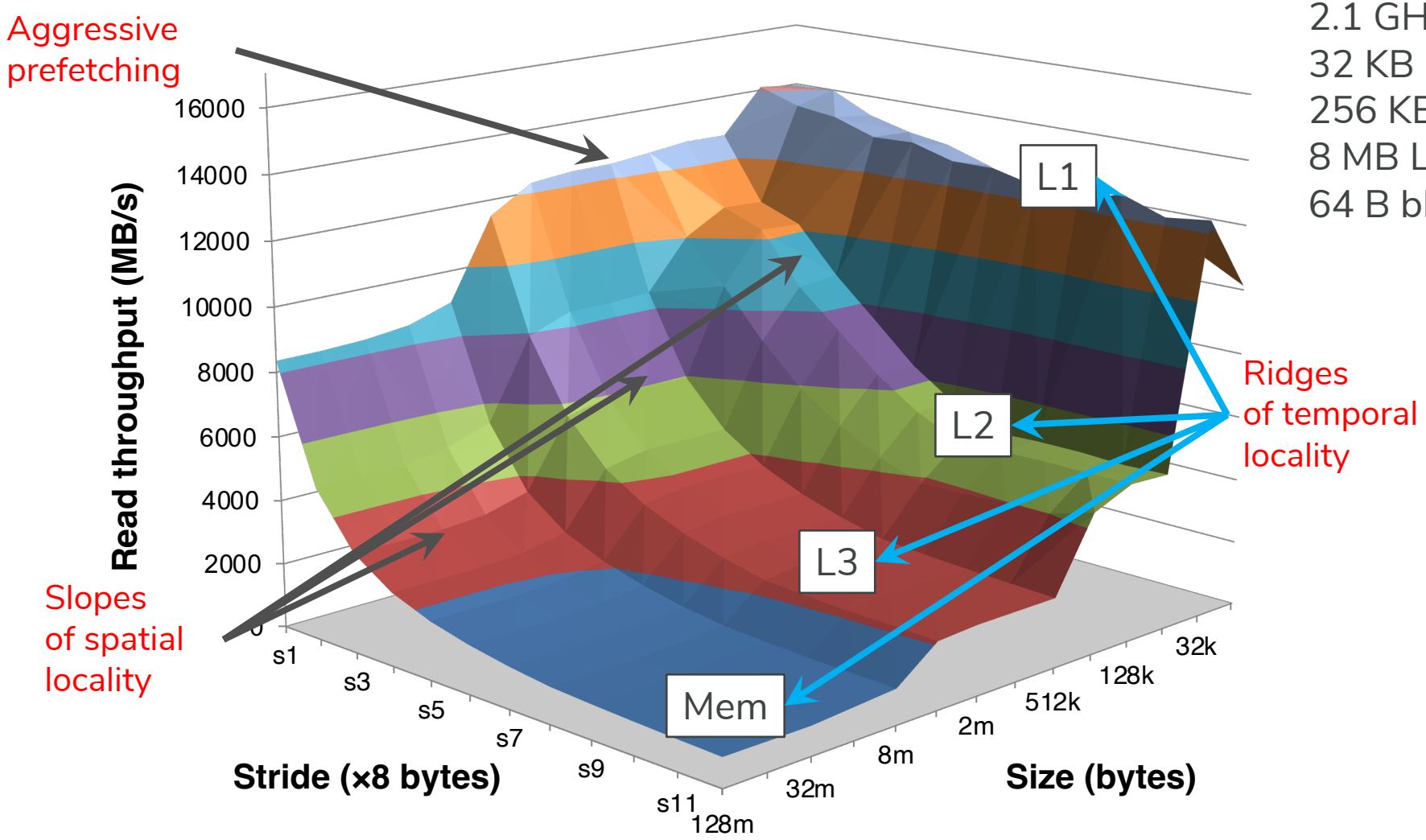
Memory Mountain: Measure read throughput as a function of spatial and temporal locality.

- Compact way to characterize memory system performance.

Memory Mountain Test Function

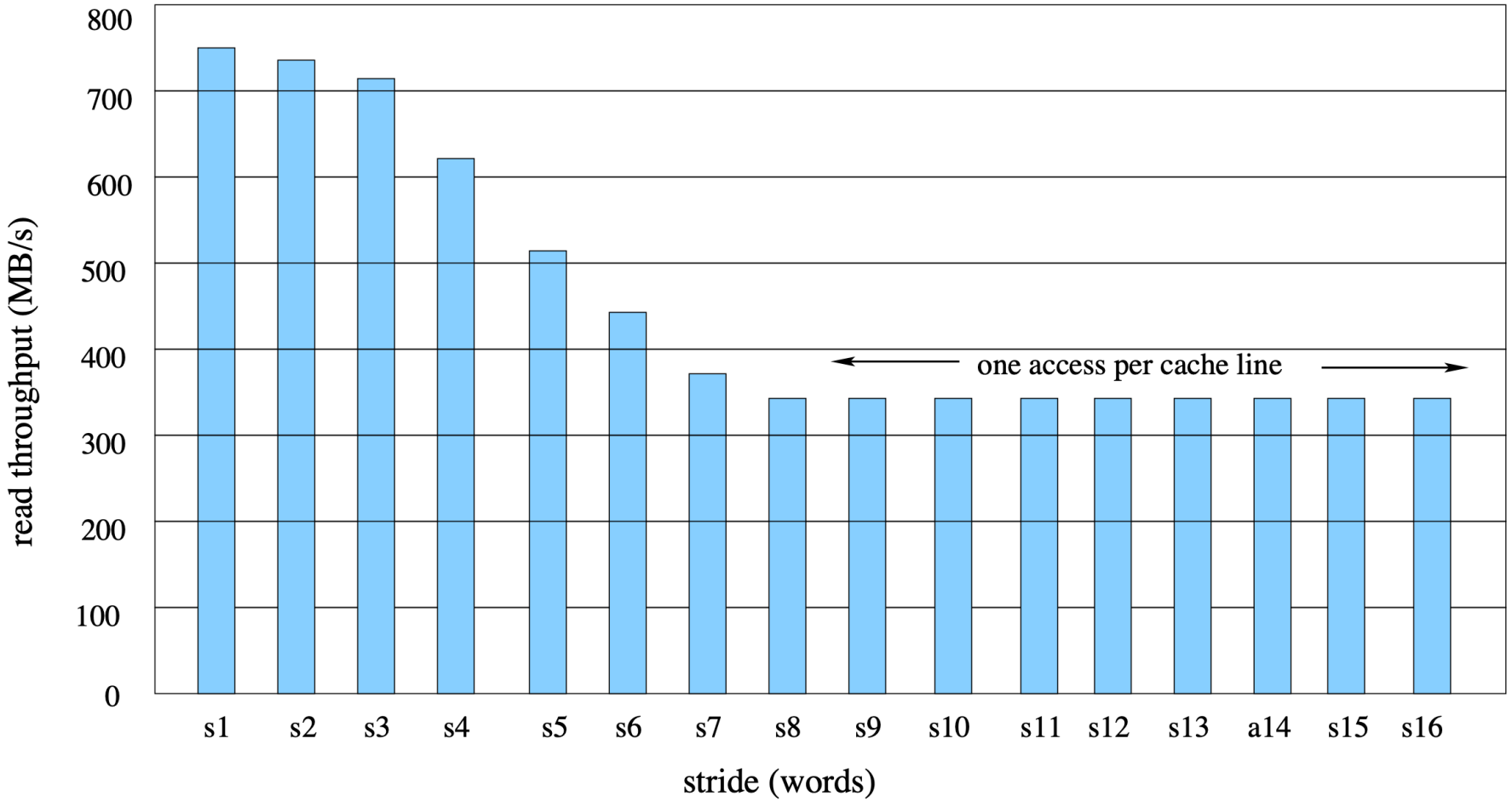
```
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz) {
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);                      /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

Example: Memory Mountain



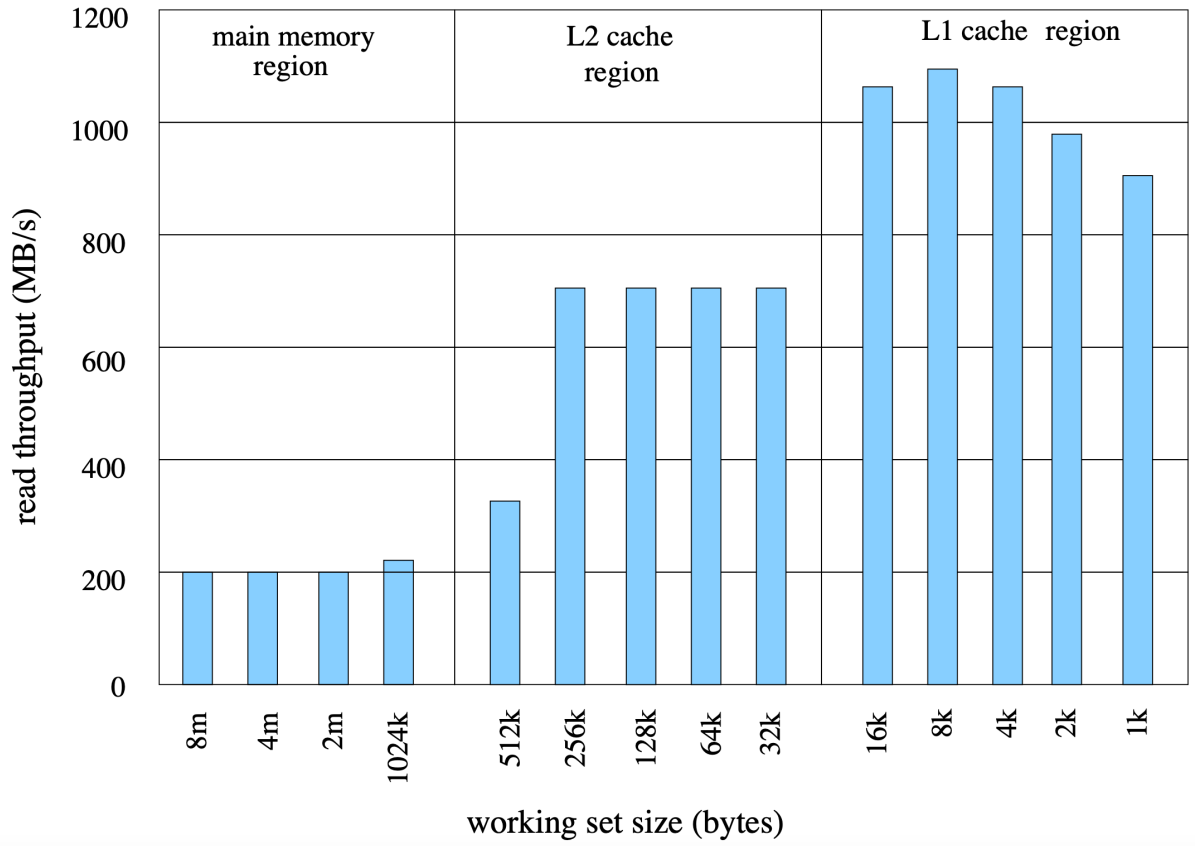
Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Example: Memory Mountain



A slice through memory mountain with size = 256KB
This shows cache block size.

Example: Memory Mountain



Slice through memory mountain with stride = 1

Illustrate the read throughput with **different** caches and memory

Memory Mountain Summary

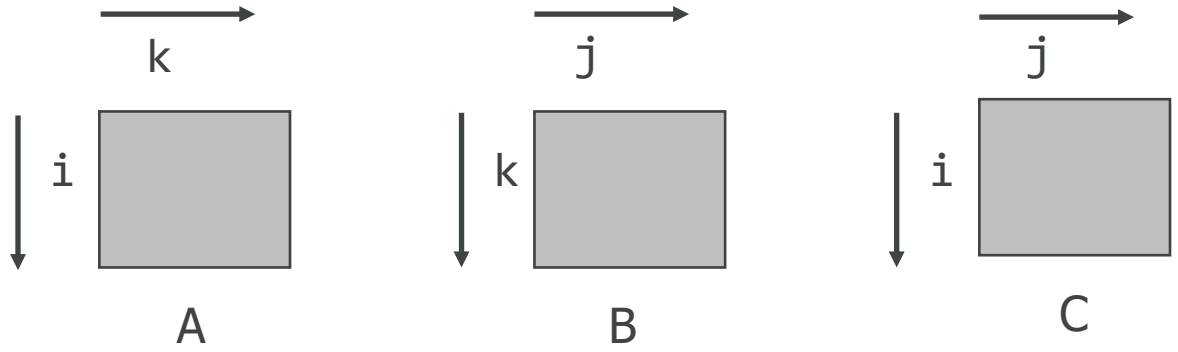
- The performance of the memory mountain is not characterized by a single number. Instead, it is a mountain of temporal and spatial locality whose elevations can vary by over 10 times.
- Wise programmers try to structure their programs so that they run in the peaks instead in the valleys.
- The aim is to exploit temporal locality so that heavily used words are fetched from the L1 cache, and to exploit spatial locality so that as many words as possible are accessed from a single L1 cache line.

Programming Ex: Matrix Multiplication

- Consider the problem of multiplying a pair of $N \times N$ matrices: $\mathbf{C} = \mathbf{AB}$.
- A matrix multiplying function is typically implemented using three nested loops, which are identified with indices i, j, and k.
- If we permute the loops and make some minor code changes, we can create six functionally equivalent versions. Each version is uniquely identified by the ordering of its loops.

Miss Rate Analysis (Matrix Multiplication)

- Assume:
  - Line size = 32 bytes (big enough for four 64-bit words)
  - Matrix dimension (*N*) is very large: approximate 1/*N* as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop



Matrix Multiplication Example (cont)

- Description:
  - Multiply $N \times N$ elements
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination; may be able to hold in register

```
/* ijk */
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;
  }
}
```

Variable sum held in register

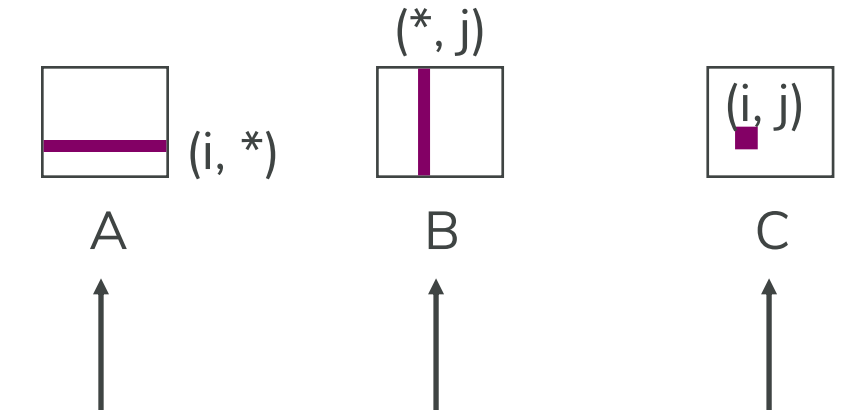
Impact of Caches on Program Performance

- C arrays allocated in row-major order (each row in contiguous memory locations)
- Stepping through columns in one row:
  - **for** (i = 0; i < N; i++)
    sum += a[0][i];
  - Accesses successive elements
  - If block size (*B*) > 4 bytes, exploit spatial locality
    - Compulsory miss rate = 4 bytes / *B*
- Stepping through rows in one column:
  - **for** (i = 0; i < n; i++)
    sum += a[i][0];
  - Accesses distant elements
  - No spatial locality!
    - Compulsory miss rate = 1 (*i.e.*, 100%)

Matrix Multiplication (ijk)

```
/* ijk */
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;
  }
}
```

Inner loop:

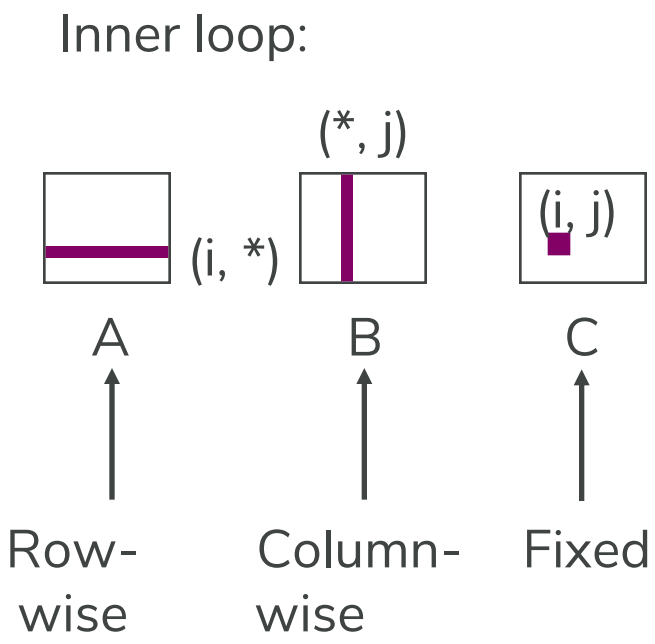


Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
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
  }
}
```

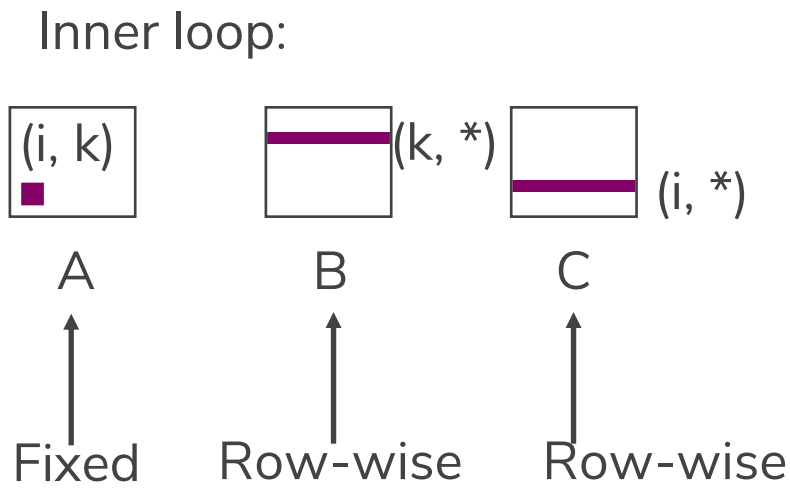


Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
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];
  }
}
```



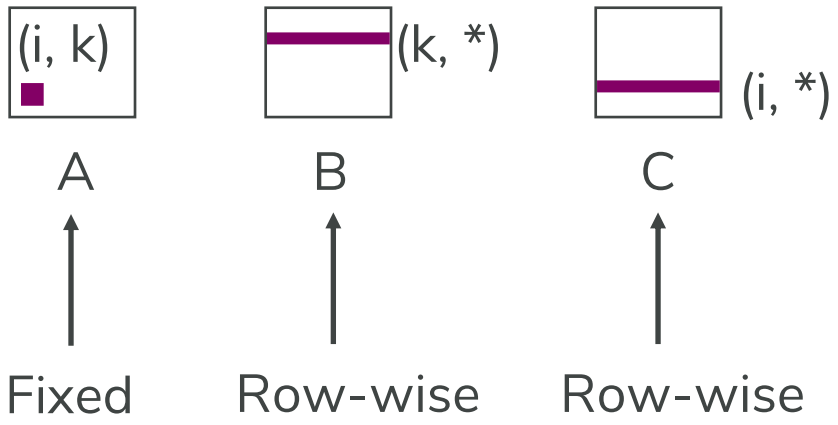
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
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];
  }
}
```

Inner loop:

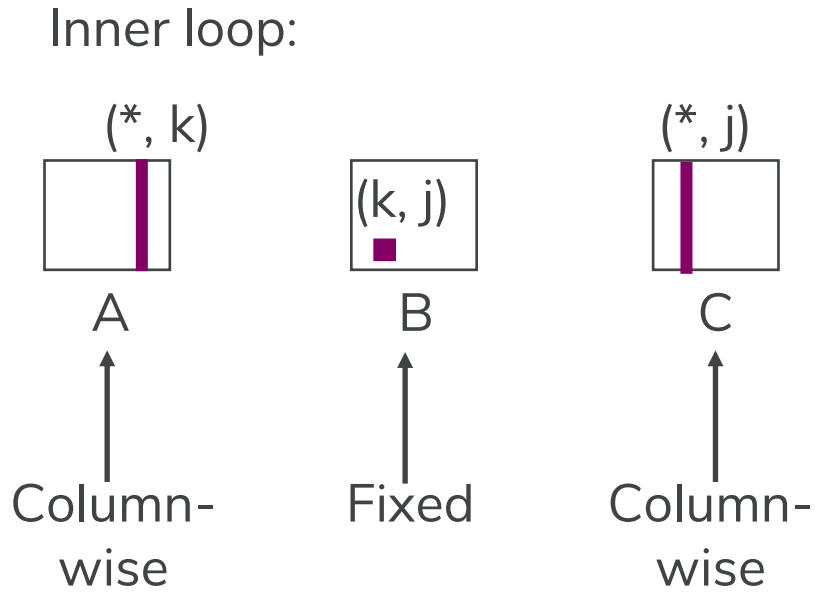


Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
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;
  }
}
```

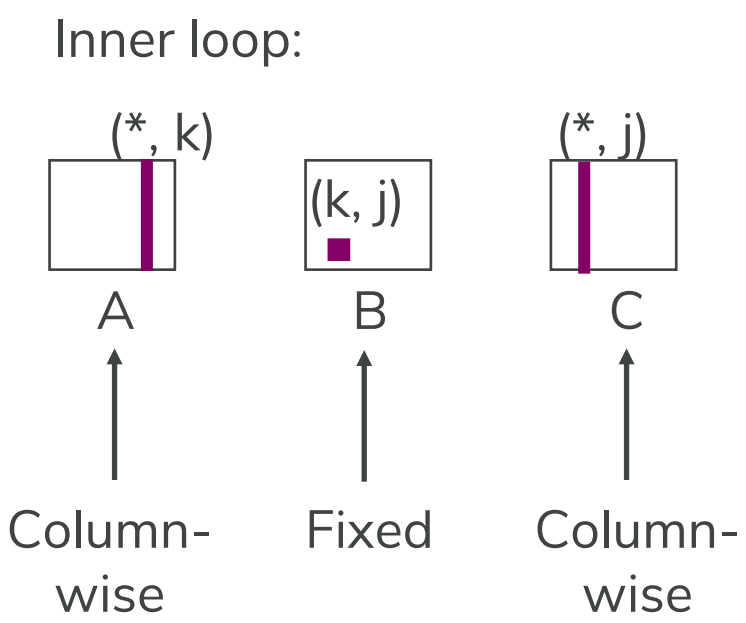


Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
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;
  }
}
```



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (Summary)

```
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;
 }
}
```

ijk (and jik):

- 2 loads, 0 stores
- misses/iteration = 1.25

```
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];
 }
}
```

kij (and ikj):

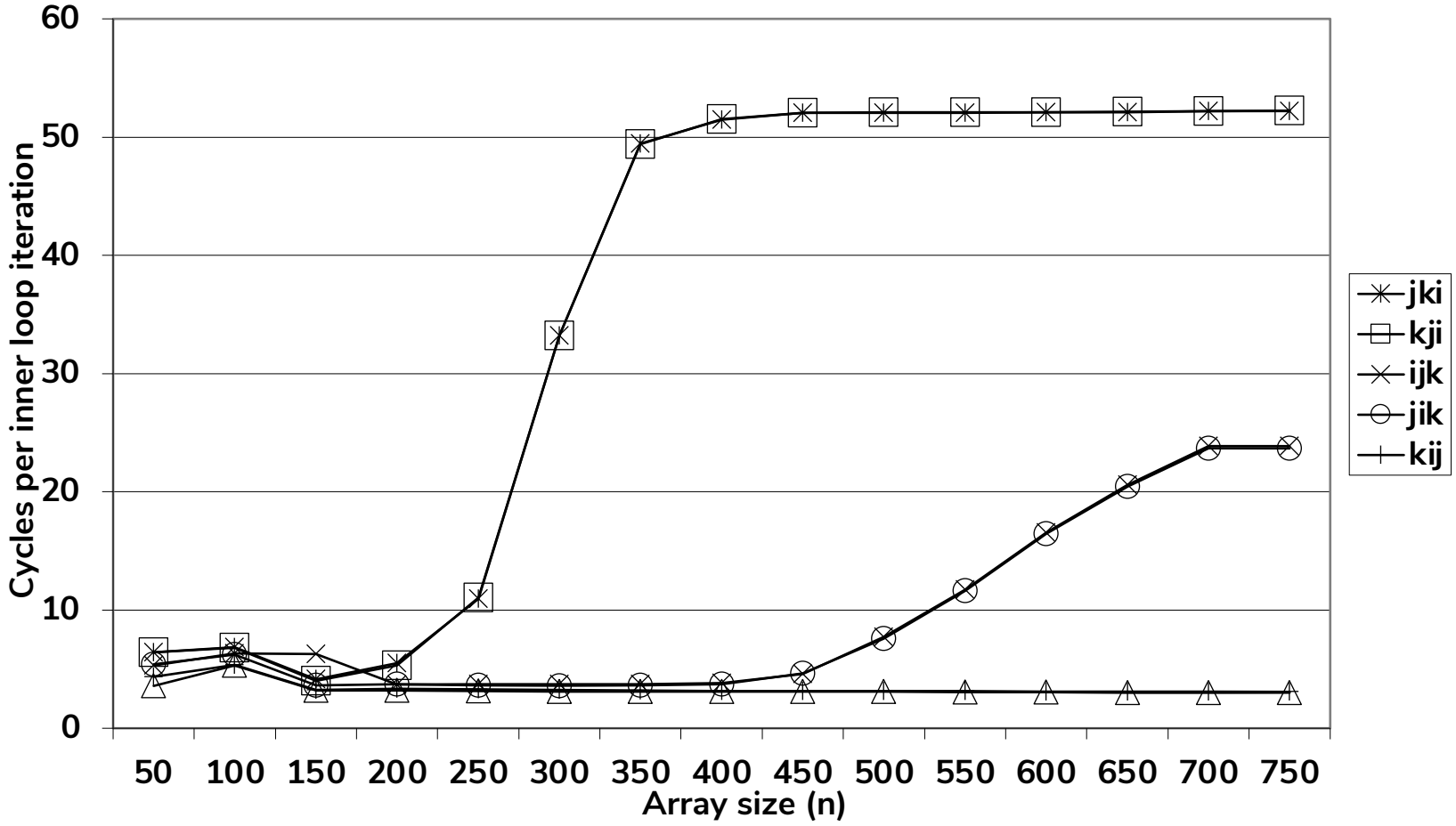
- 2 loads, 1 store
- misses/iteration = 0.5

```
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;
 }
}
```

jki (and kji):

- 2 loads, 1 store
- misses/iteration = 2.0

Core i7 Matrix Multiply Performance



Concluding Observations

- **Programmers can optimize for cache performance**
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- **All systems favor "cache-friendly code"**
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)