



Cache Memories

18-213/18-613: Introduction to Computer Systems
10th Lecture, September 26th, 2024

Reminder: AIV Policy

<http://www.cs.cmu.edu/~18213/academicintegrity.html>

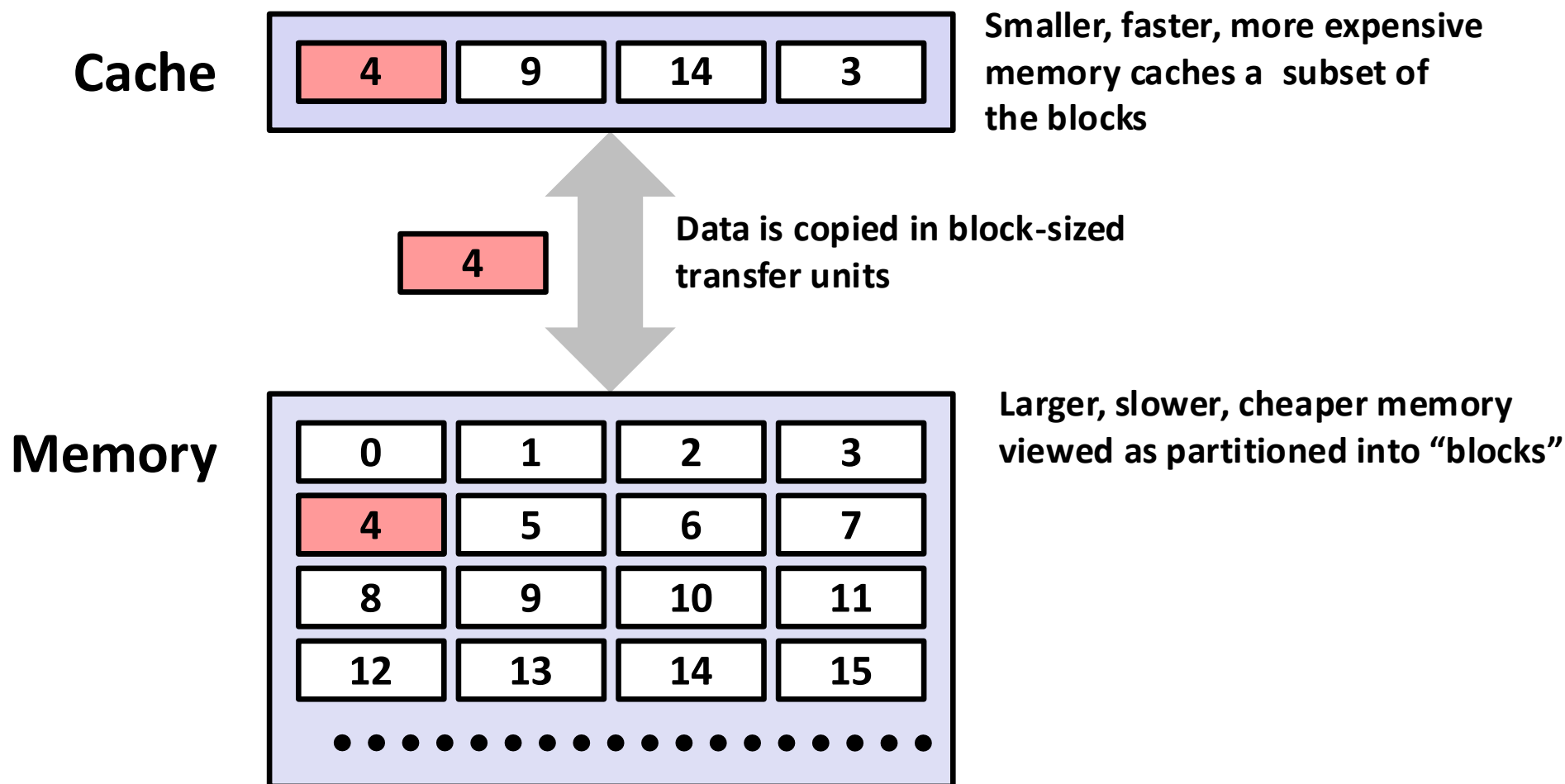
- **No unauthorized use of information**
 - Borrowing code: by copying, retyping, **looking at** a file
 - Describing: verbal description of code from one person to another
 - Searching the Web for solutions
 - Copying code from a previous course or online solution
 - Reusing your code from a previous semester (here or elsewhere)
- **No unauthorized supplying of information**
 - Providing copy: Giving a copy of a file to someone
 - Providing access:
 - Putting material in unprotected directory
 - Putting material in unprotected code repository (e.g., Github)
- **No collaborations beyond high-level, strategic advice**
 - Anything more than block diagram or a few words

**Start early. Make frequent github commits. Plan for stumbling blocks. Use available help.
Don't panic: Far better to turn in 25% correct solution than get an AIV.**

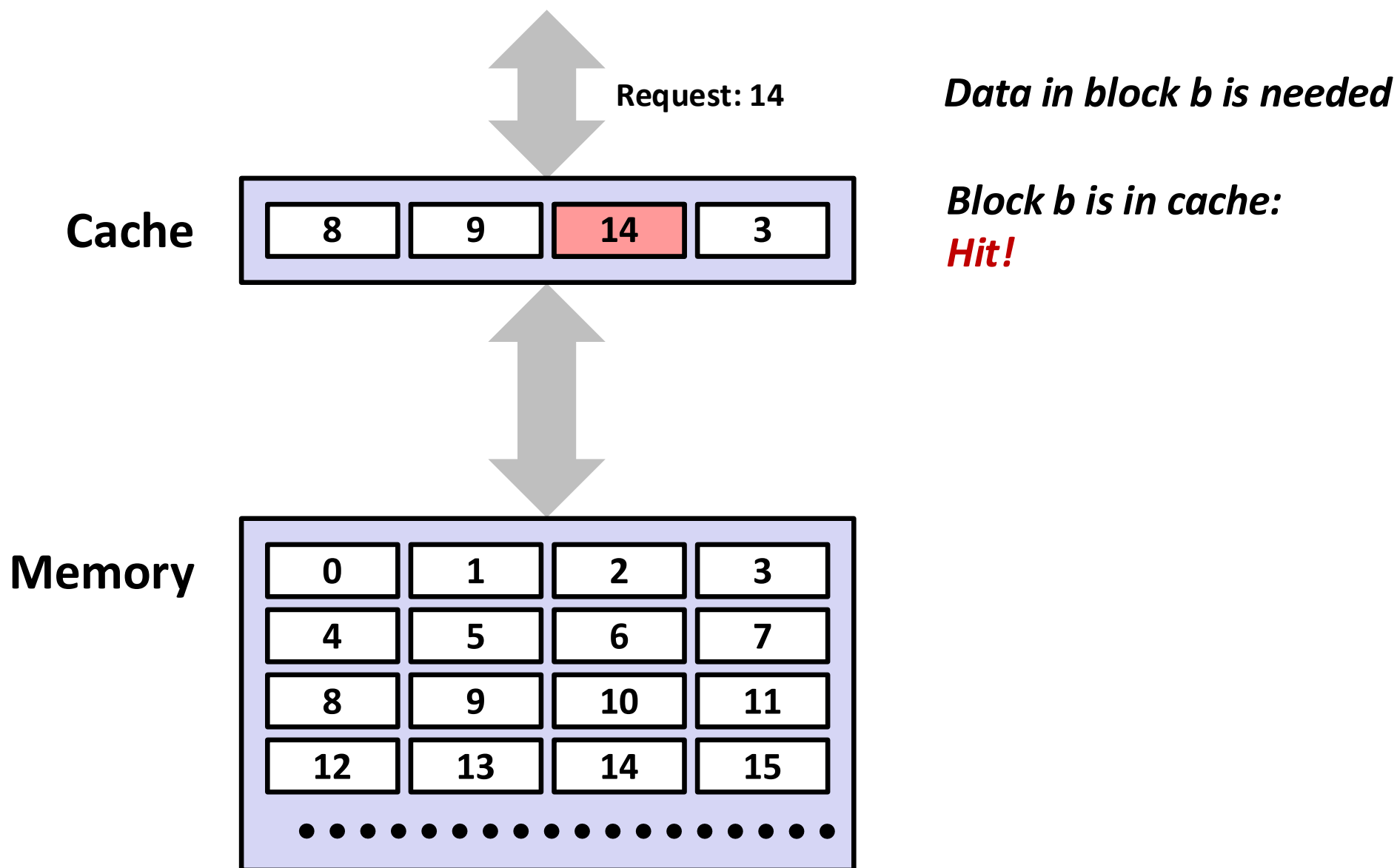
Today

- **Cache memory organization and operation** CSAPP 6.4-6.5
- **Performance impact of caches**
 - The memory mountain CSAPP 6.6.1
 - Rearranging loops to improve spatial locality CSAPP 6.6.2
 - Using blocking to improve temporal locality CSAPP 6.6.3

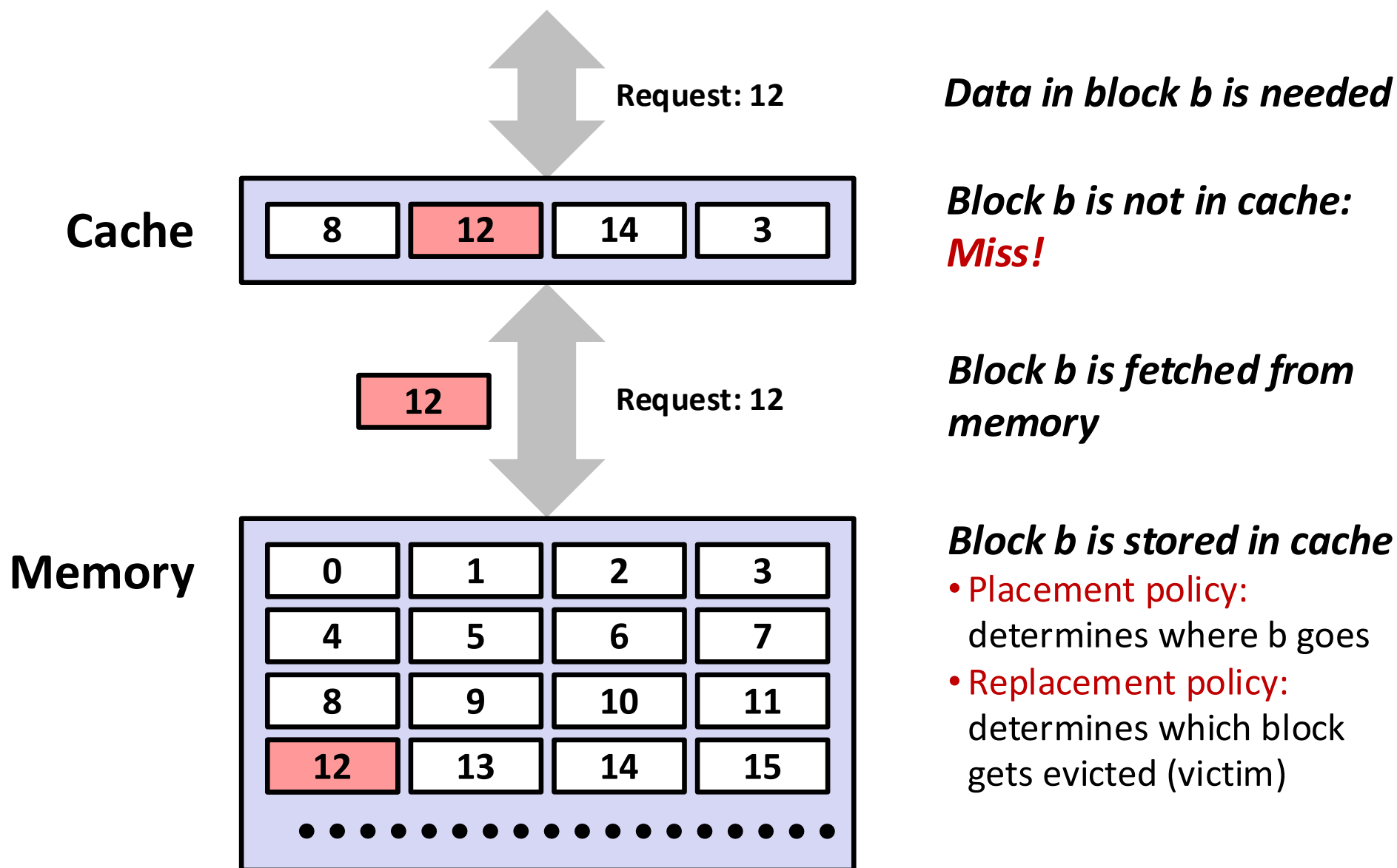
Recall: General Cache Concepts



General Cache Concepts: Hit



General Cache Concepts: Miss



Working Set, Locality, and Caches

- **Working Set:** The set of data a program is currently “working on”
 - Definition of “currently” depends on context, e.g., in this loop
 - Includes accesses to data and instructions
- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently
 - Nearby addresses: **Spatial Locality**
 - Equal addresses: **Temporal locality**
- **Caches** take advantage of temporal locality by storing recently used data, and spatial locality by copying data in block-sized transfer units
 - Locality reduces working set sizes
 - Caches are most effective when the working set fits in the cache

Recall: 3 Types of Cache Misses

■ Cold (compulsory) miss

- Cold misses occur because the cache starts empty and this is the first reference to the block.

■ Capacity miss

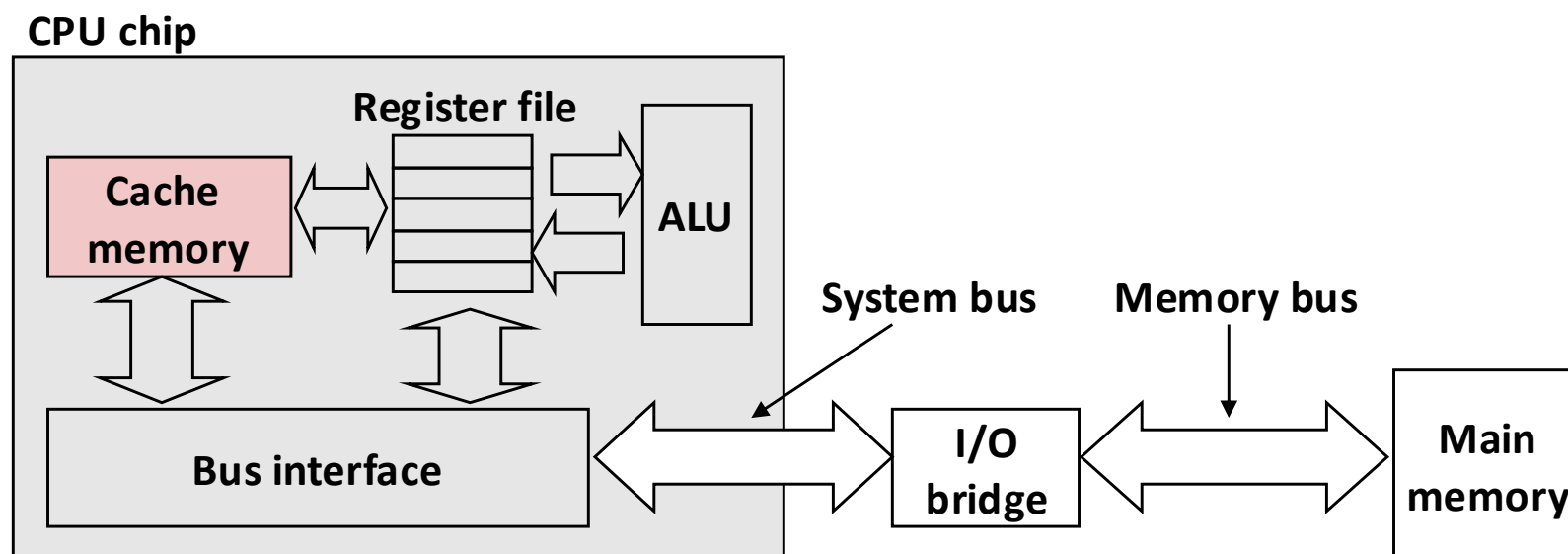
- Occurs when the set of active cache blocks (**working set**) is larger than the cache.

■ Conflict miss

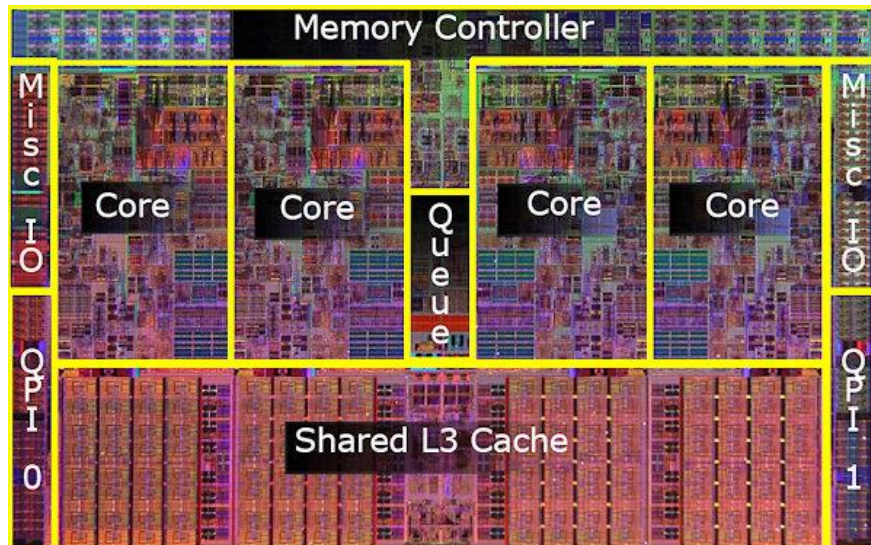
- Occurs when the cache is large enough, but too many data objects all map (by the **placement policy**) to the same limited set of blocks
 - E.g., if the placement policy maps both 0 and 8 to the same block, then referencing 0, 8, 0, 8, 0, 8, ... would miss every time.

CPU 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 cache
- Typical system structure:

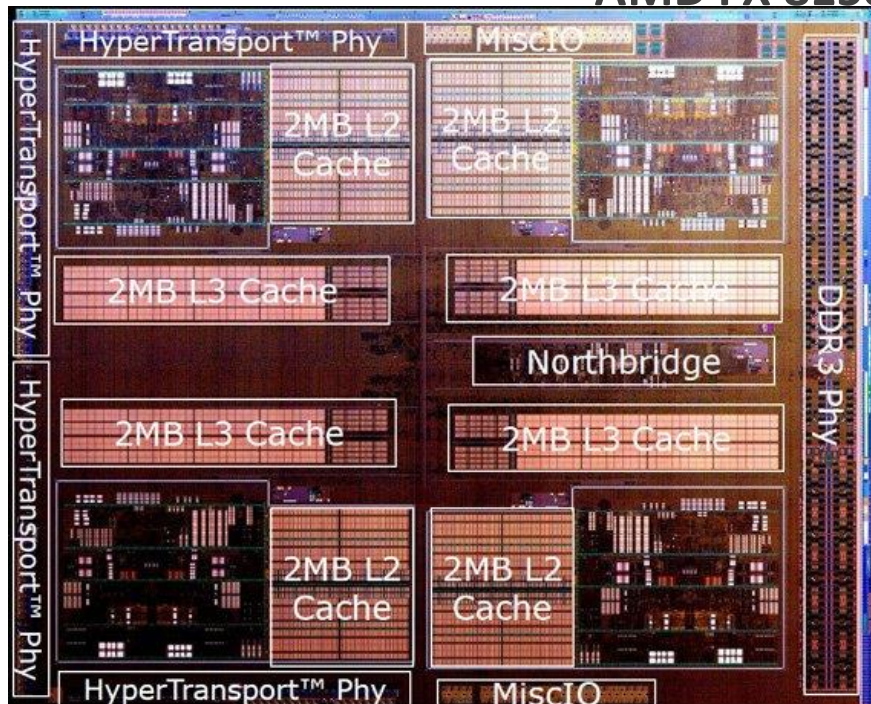


What it Really Looks Like

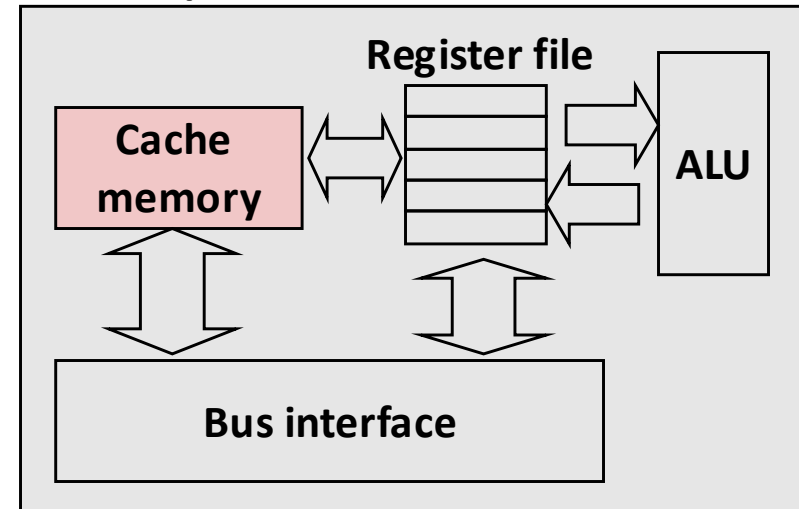


Nehalem

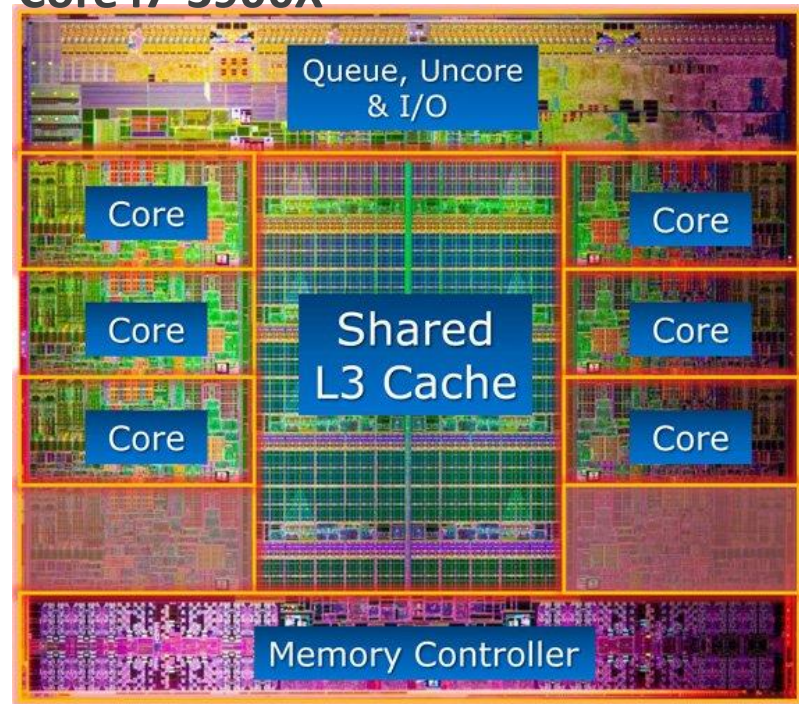
AMD FX 8150



CPU chip



Core i7-3960X



What it Really Looks Like (Cont.)

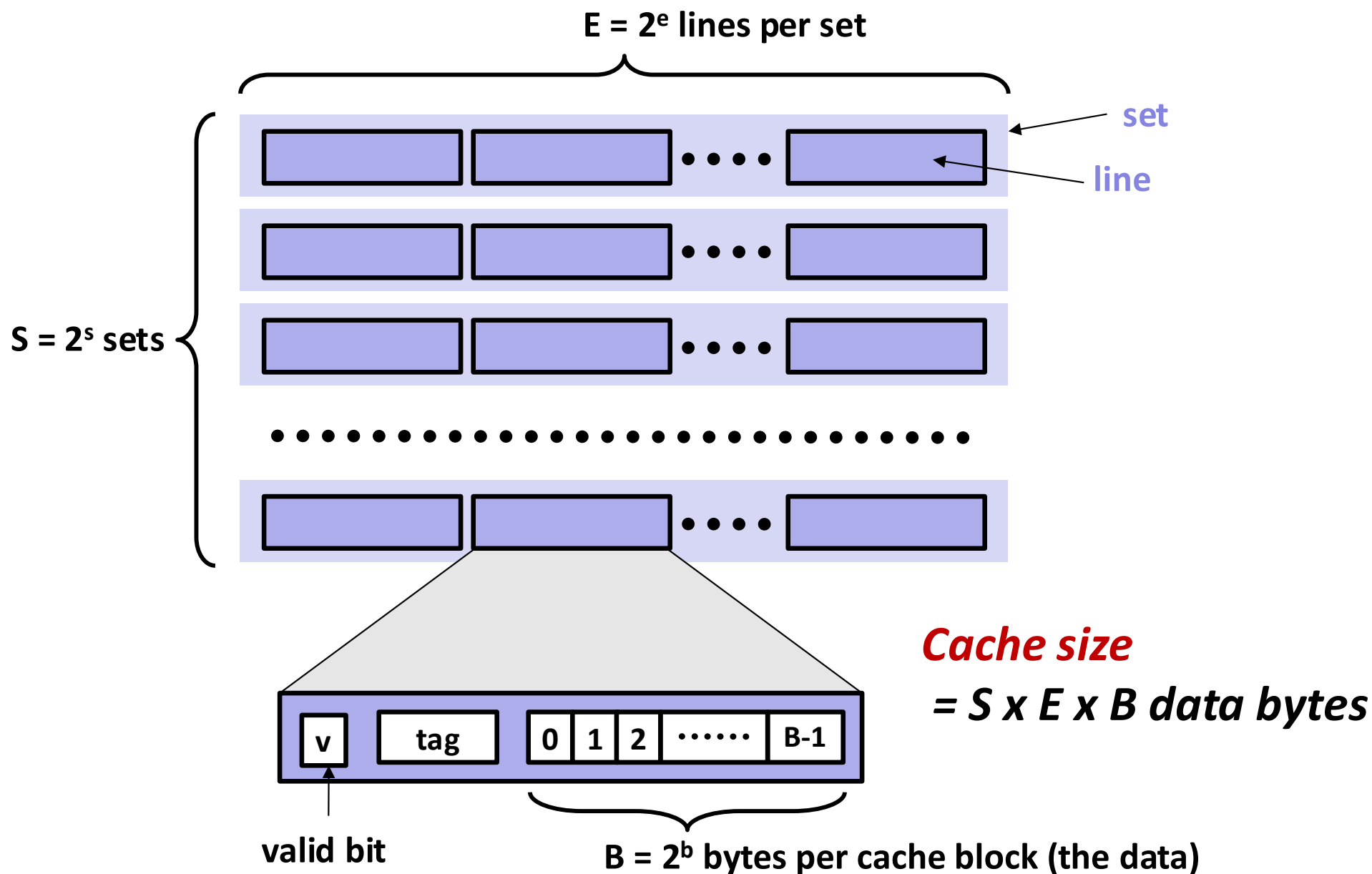


Intel Alder Lake (2021)

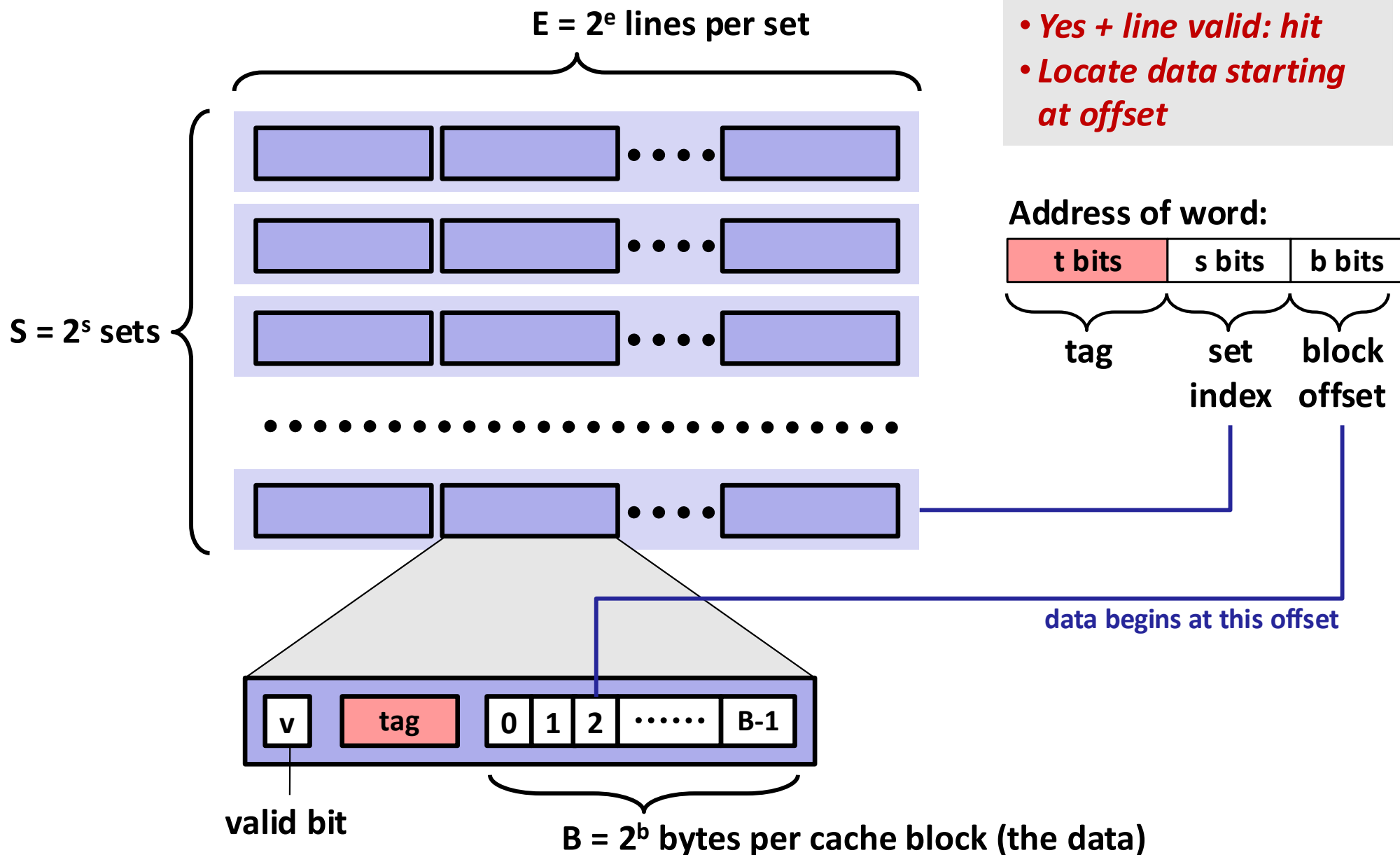
8 P-cores + 8 E-cores

L1 caches per P-core: 32KB Instruction & 48KB Data
 L1 caches per E-core: 64KB Instruction & 32KB Data
 L2 caches: 1.25MB per P-core, 2MB per four E-cores
 L3 cache: 30MB shared among all cores

General Cache Organization (S, E, B)



Cache Read

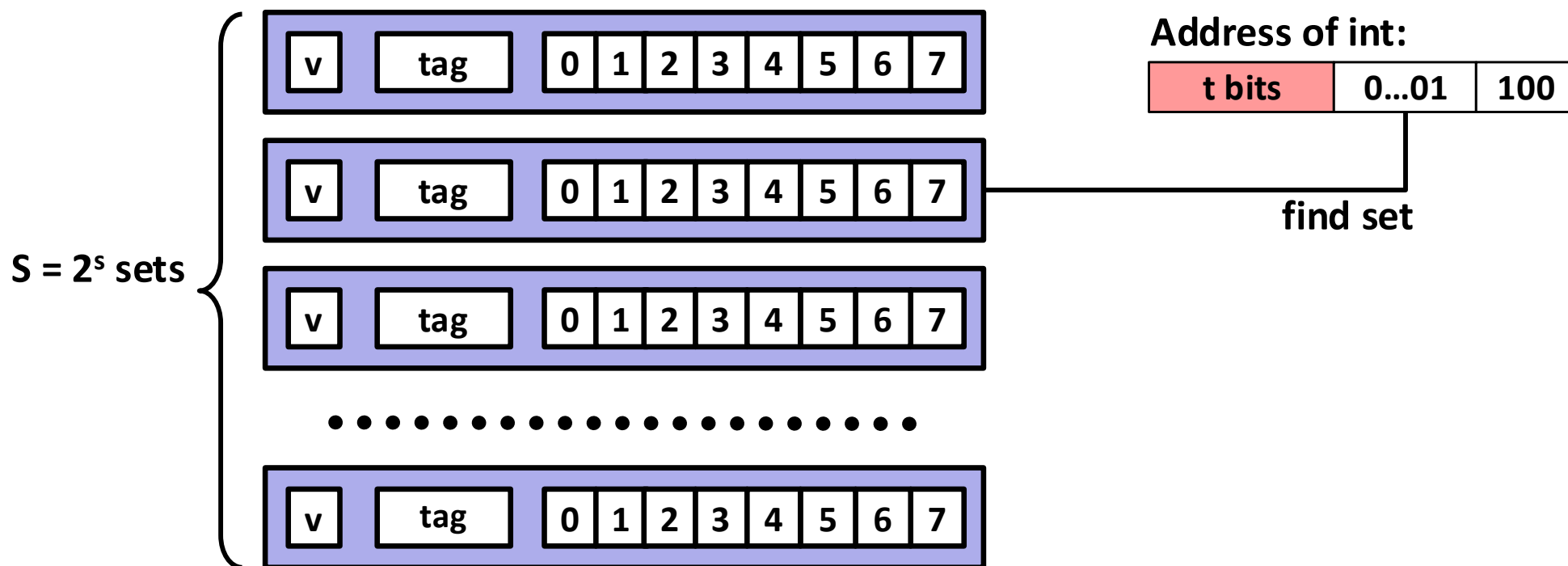


- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

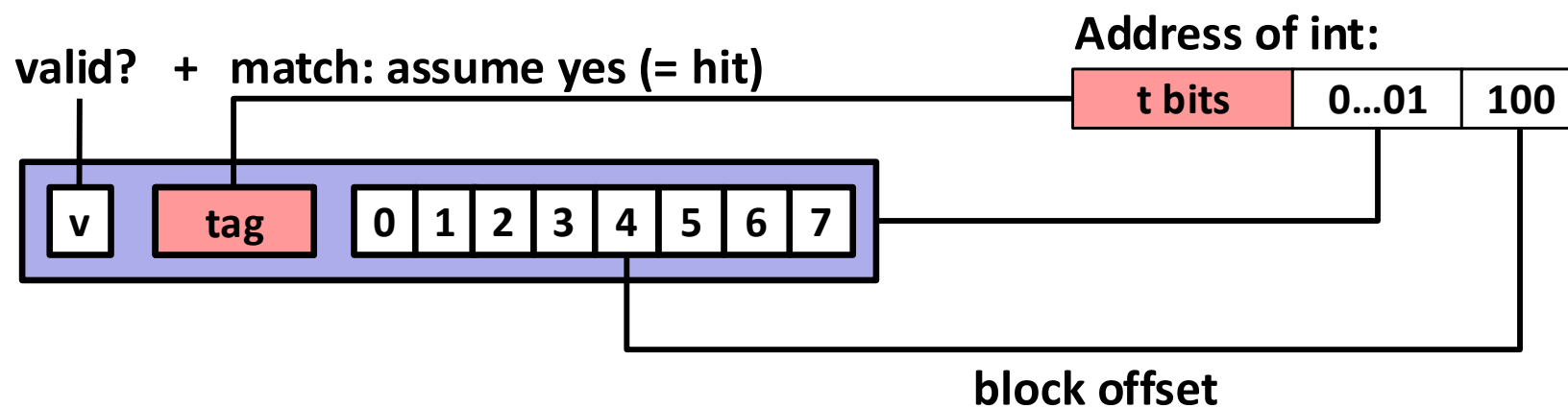
Assume: cache block size B=8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

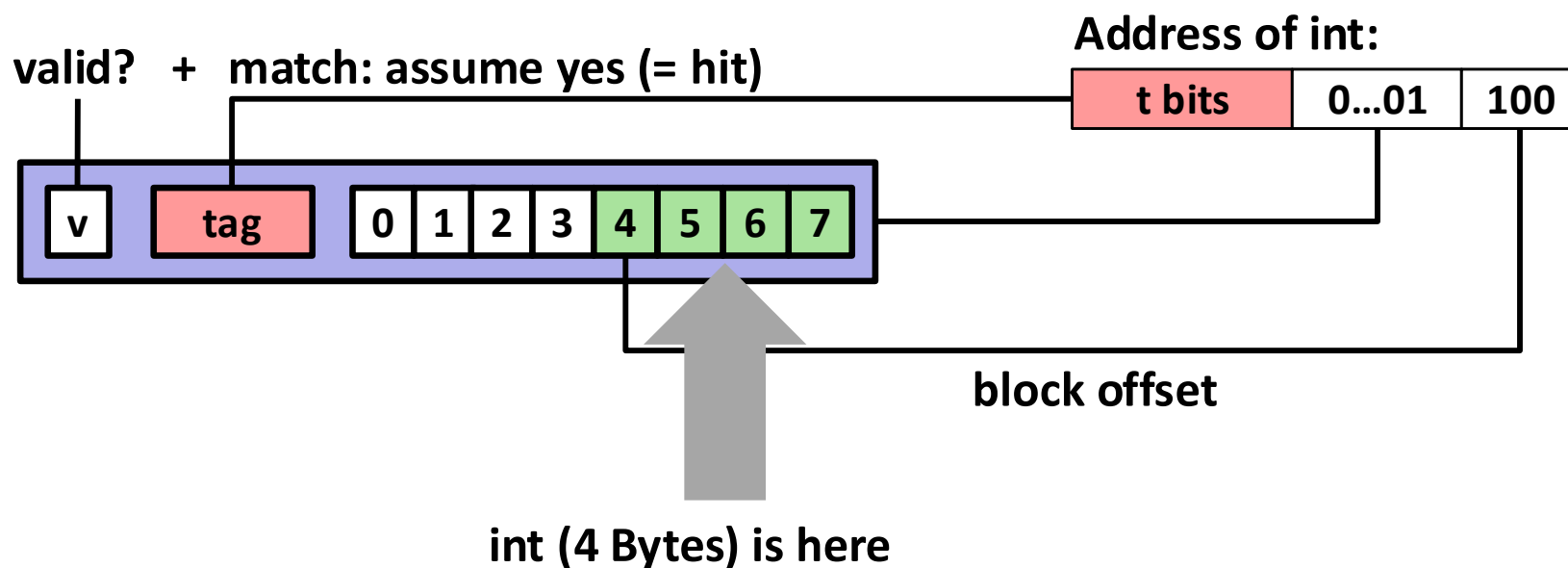
Assume: cache block size B=8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

Assume: cache block size B=8 bytes



If tag doesn't match (= miss): old line is evicted and replaced

Direct-Mapped Cache Simulation

t=1	s=2	b=1
X	XX	X

4-bit addresses (address space size M=16 bytes)
 S=4 sets, E=1 Blocks/set, B=2 bytes/block

Address trace (reads, one byte per read):

0	[<u>000</u> 0] ₂ ,	miss	(cold)
1	[0 <u>00</u> 1] ₂ ,	hit	
7	[0 <u>11</u> 1] ₂ ,	miss	(cold)
8	[<u>100</u> 0] ₂ ,	miss	(cold)
0	[<u>000</u> 0] ₂	miss	(conflict)

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

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

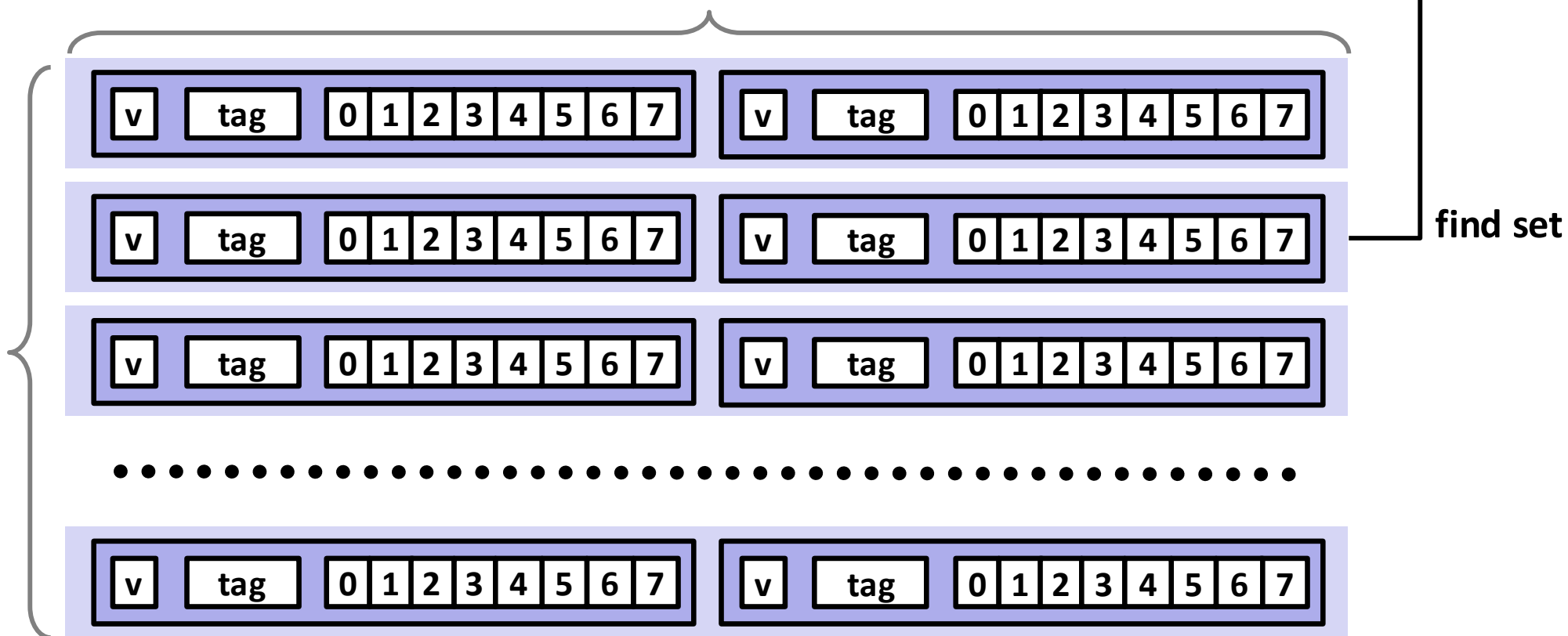
E = 2: Two lines per set

Assume: cache block size B=8 bytes

2 lines per set

Address of short int:

t bits	0...01	100
--------	--------	-----

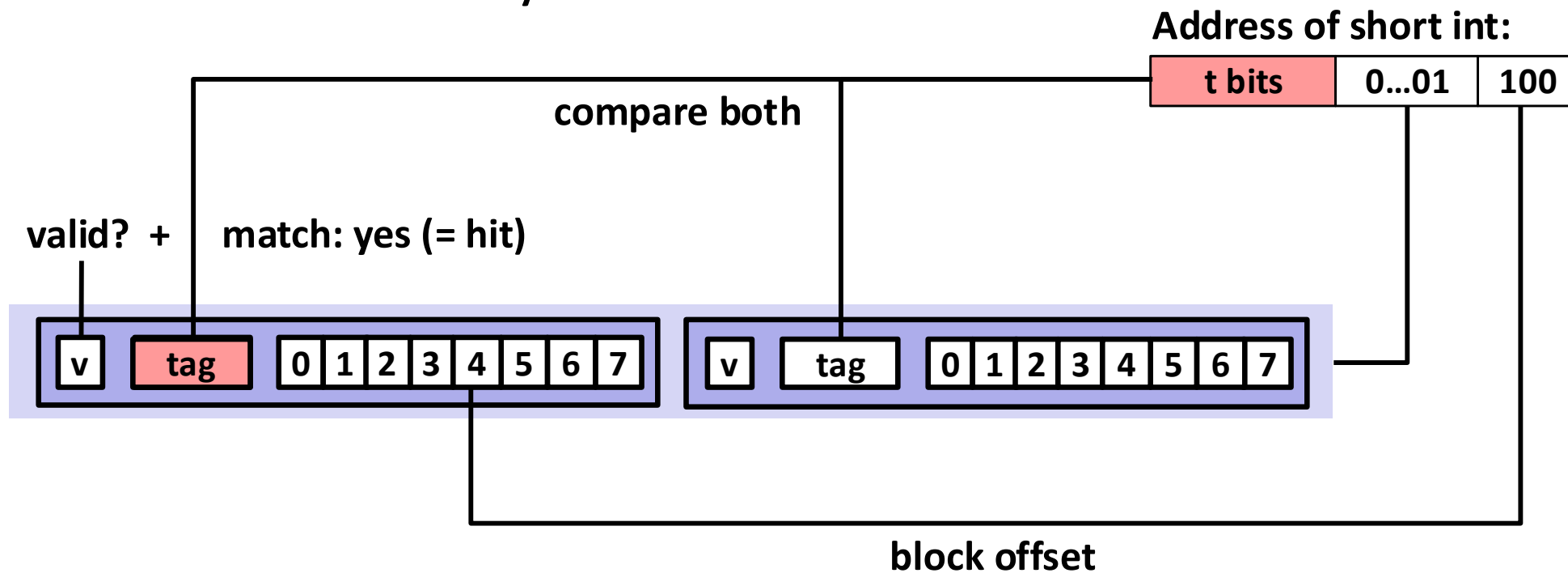


S sets

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

E = 2: Two lines per set

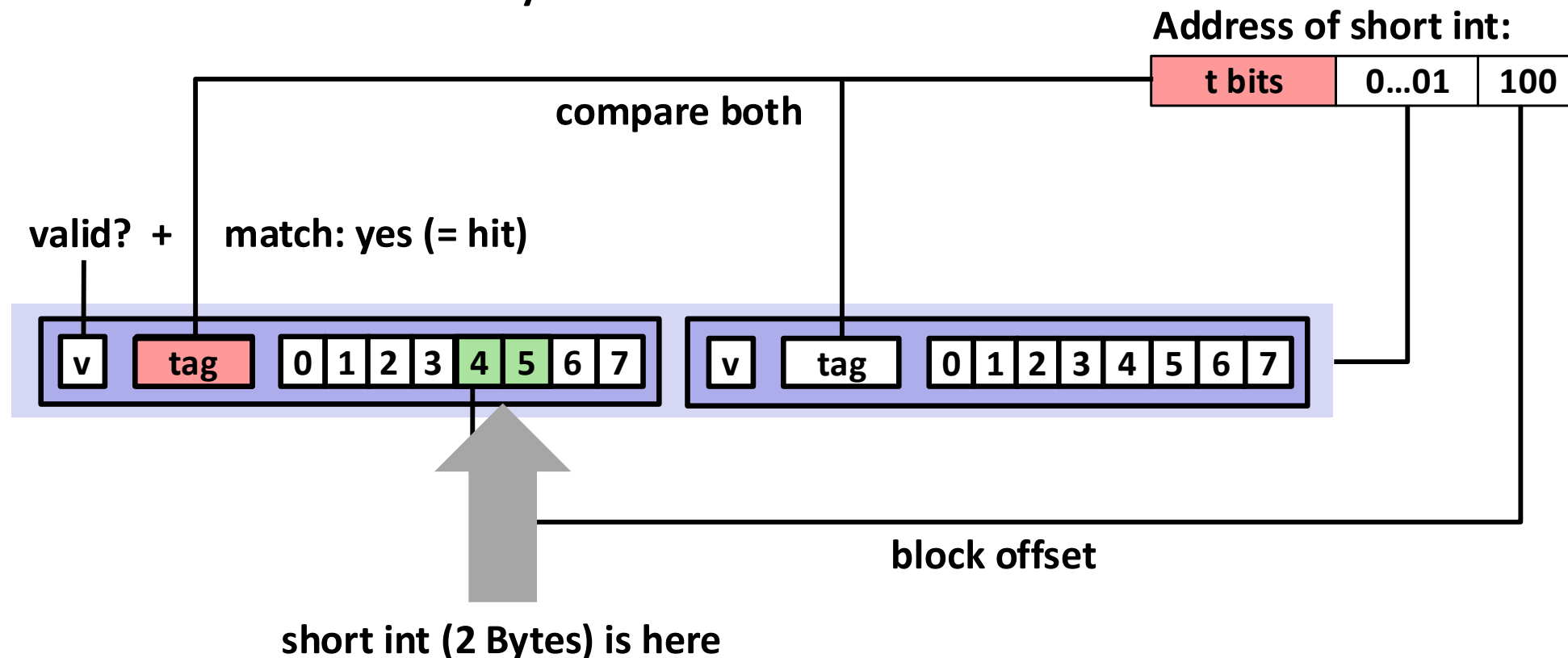
Assume: cache block size B=8 bytes



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

E = 2: Two lines per set

Assume: cache block size B=8 bytes



No match or not valid (= miss):

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

2-Way Set Associative Cache Simulation

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

4-bit addresses (M=16 bytes)

S=2 sets, E=2 blocks/set, B=2 bytes/block

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[0 <u>1</u> 1 ₂],	miss
8	[<u>1</u> 00 ₂],	miss
0	[00 <u>0</u> 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		

What about writes?

■ Multiple copies of data exist:

- L1, L2, L3, Main Memory, Disk

■ What to do on a write-hit?

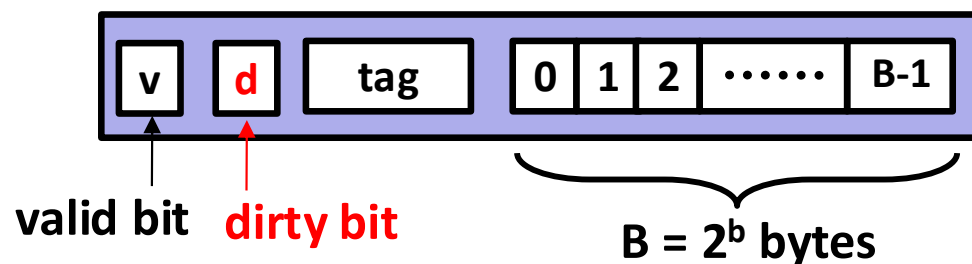
- **Write-through** (write immediately to memory)
- **Write-back** (defer write to memory until replacement of line)
 - Each cache line needs a dirty bit (set if data differs from memory)

■ What to do on a write-miss?

- **Write-allocate** (load into cache, update line in cache)
 - Good if more writes to the location will follow
- **No-write-allocate** (writes straight to memory, does not load into cache)

■ Typical

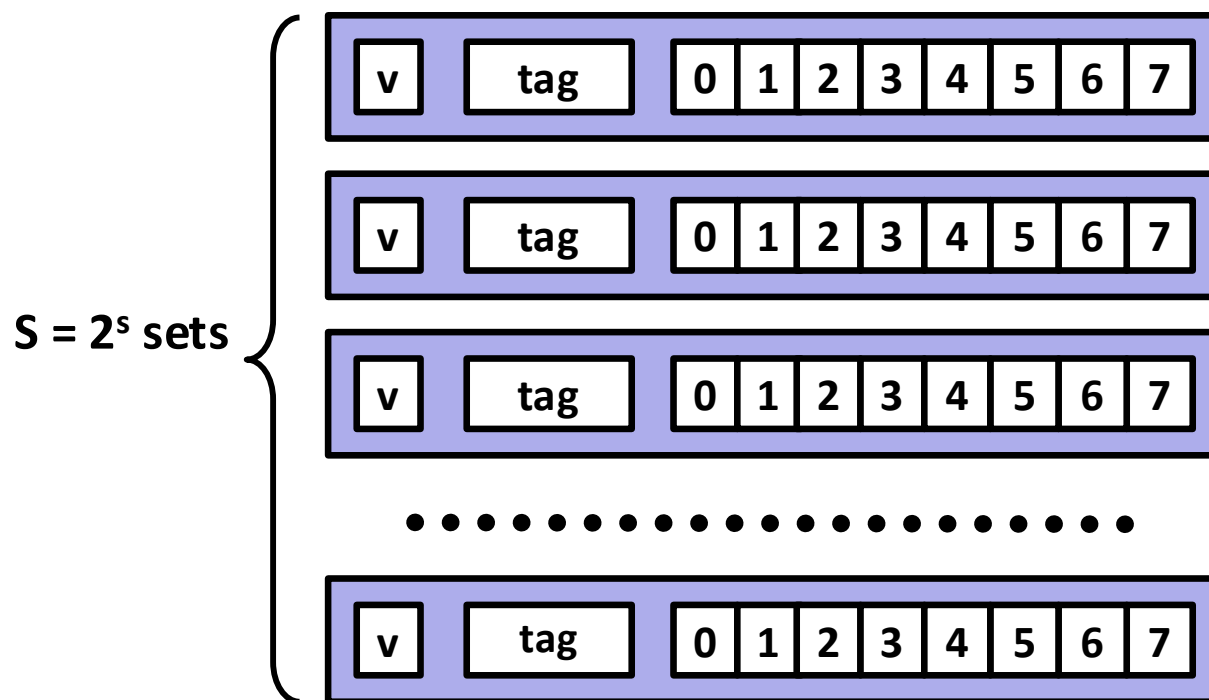
- Write-through + No-write-allocate
- **Write-back + Write-allocate**



Why Index Using Middle Bits?

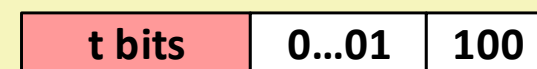
Direct mapped: One line per set

Assume: cache block size 8 bytes



**Standard Method:
Middle bits indexing**

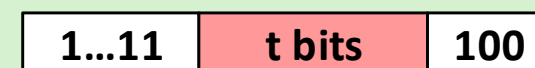
Address of int:



find set

**Alternative Method:
High bits indexing**

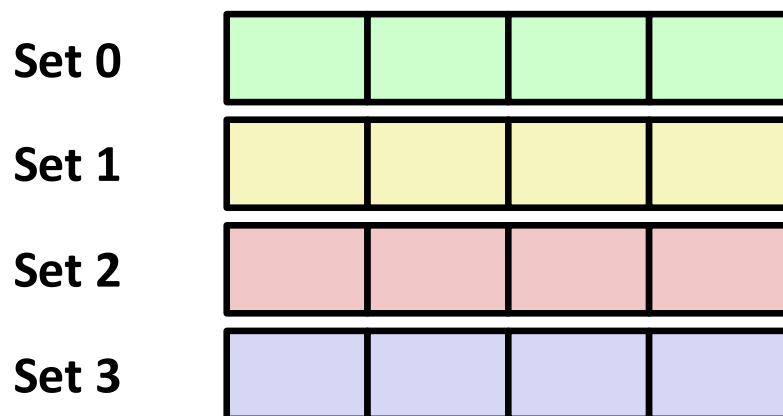
Address of int:



find set

Illustration of Indexing Approaches

- **64-byte memory**
 - 6-bit addresses
- **16 byte, direct-mapped cache**
- **Block size = 4. (Thus, 4 sets; why?)**
- **2 bits tag, 2 bits index, 2 bits offset**



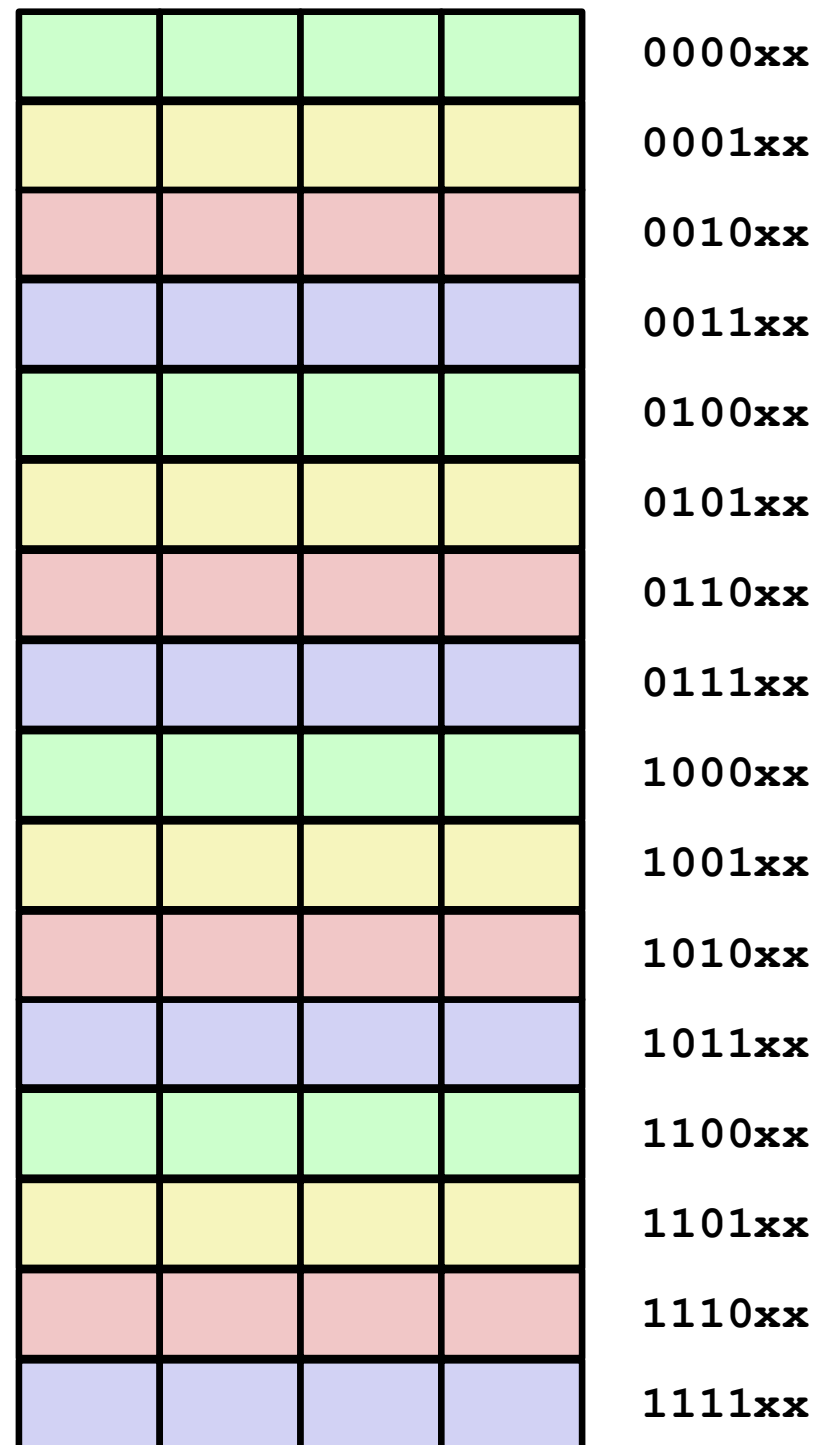
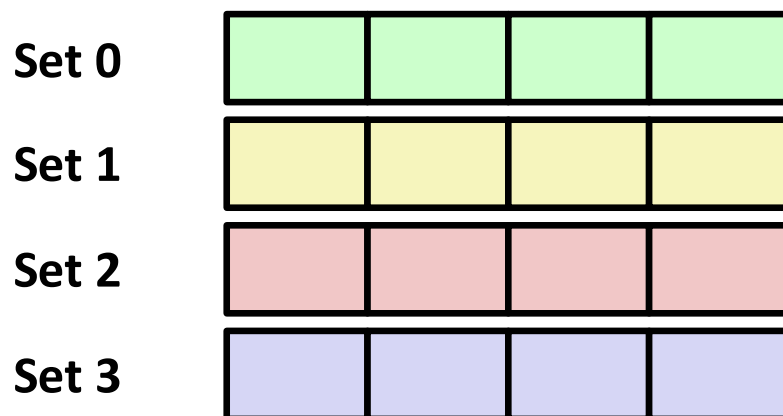
				0000xx
				0001xx
				0010xx
				0011xx
				0100xx
				0101xx
				0110xx
				0111xx
				1000xx
				1001xx
				1010xx
				1011xx
				1100xx
				1101xx
				1110xx
				1111xx

Middle Bits Indexing

■ Addresses of form **TTSSBB**

- **TT** Tag bits
- **SS** Set index bits
- **BB** Offset bits

■ Makes good use of spatial locality

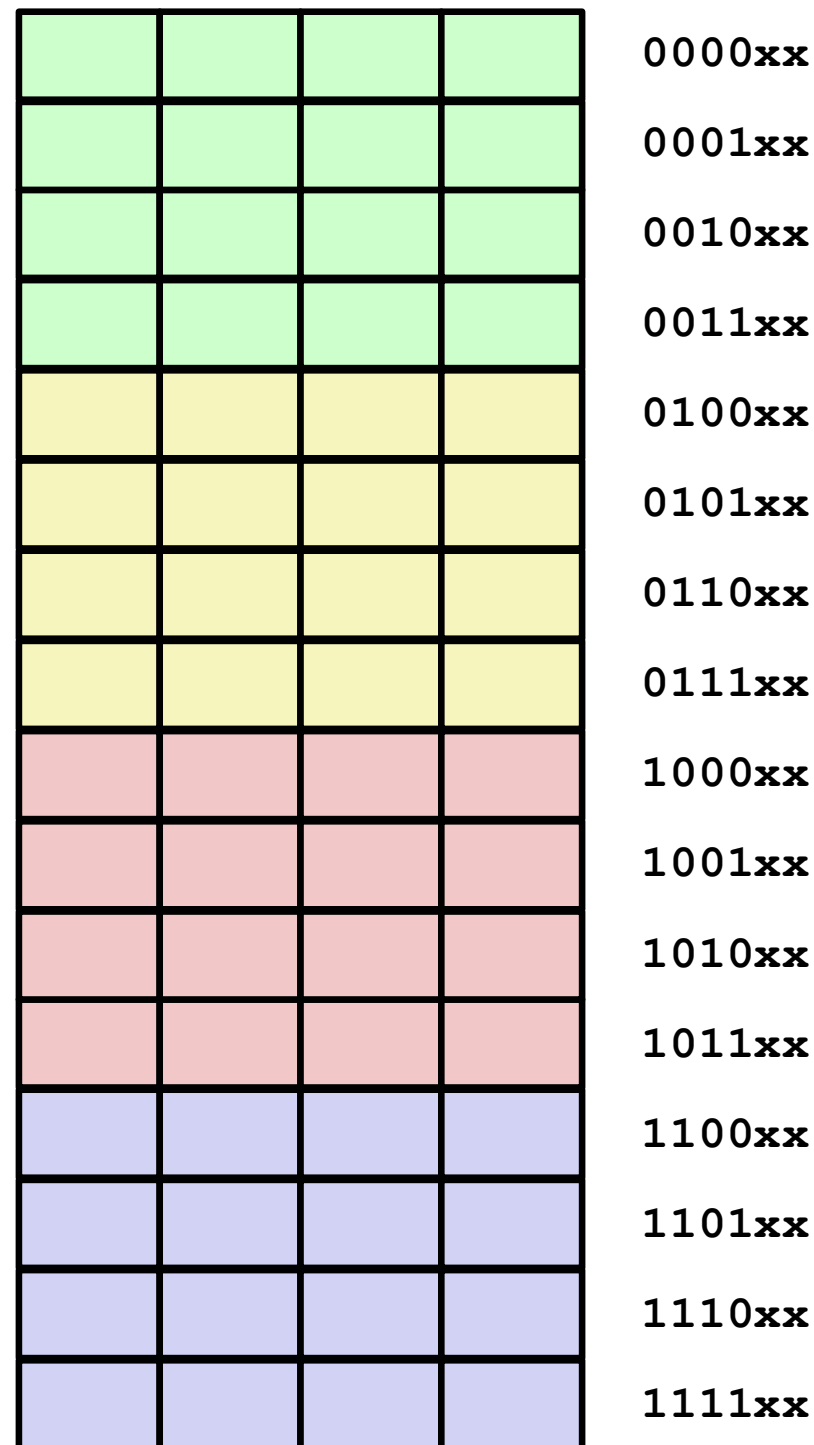
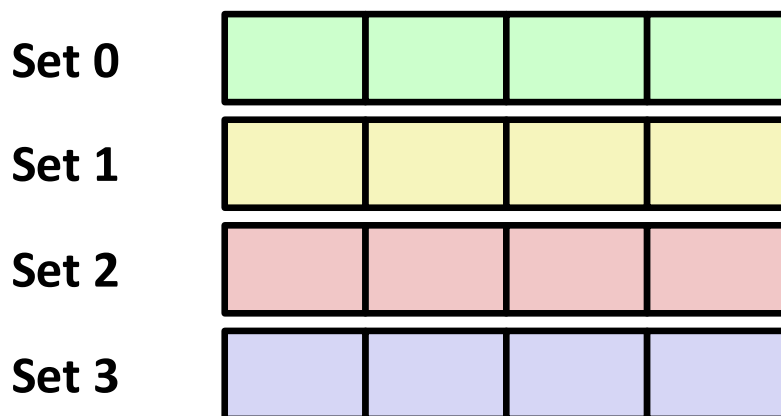


High Bits Indexing

■ Addresses of form **SS****TT****BB**

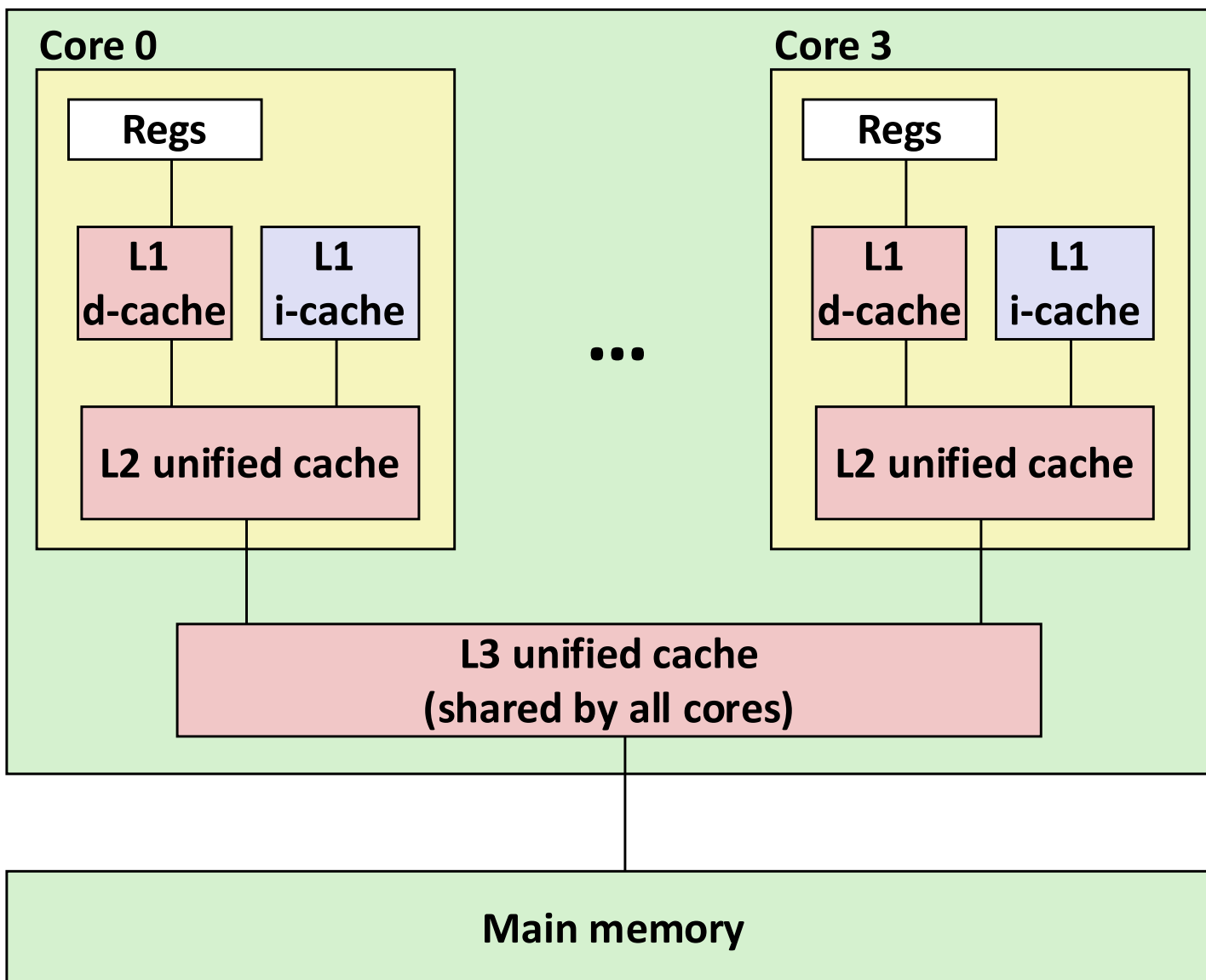
- **SS** Set index bits
- **TT** Tag bits
- **BB** Offset bits

■ Program with high spatial locality would generate lots of conflicts



Intel Core i7 Cache Hierarchy

Processor package



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:

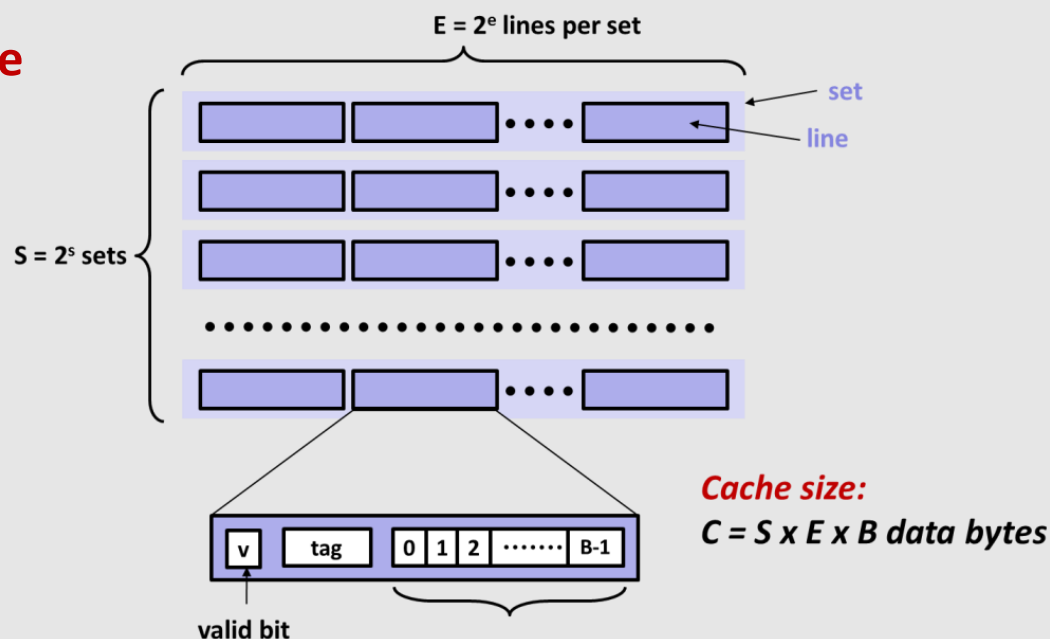
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for
all caches.

Example: Core i7 L1 Data Cache

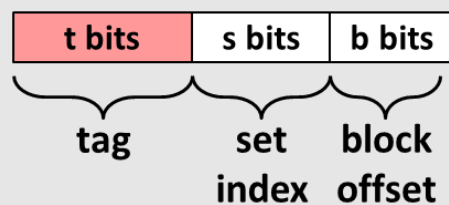
32 KB 8-way set associative
64 bytes/block
47 bit address range

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

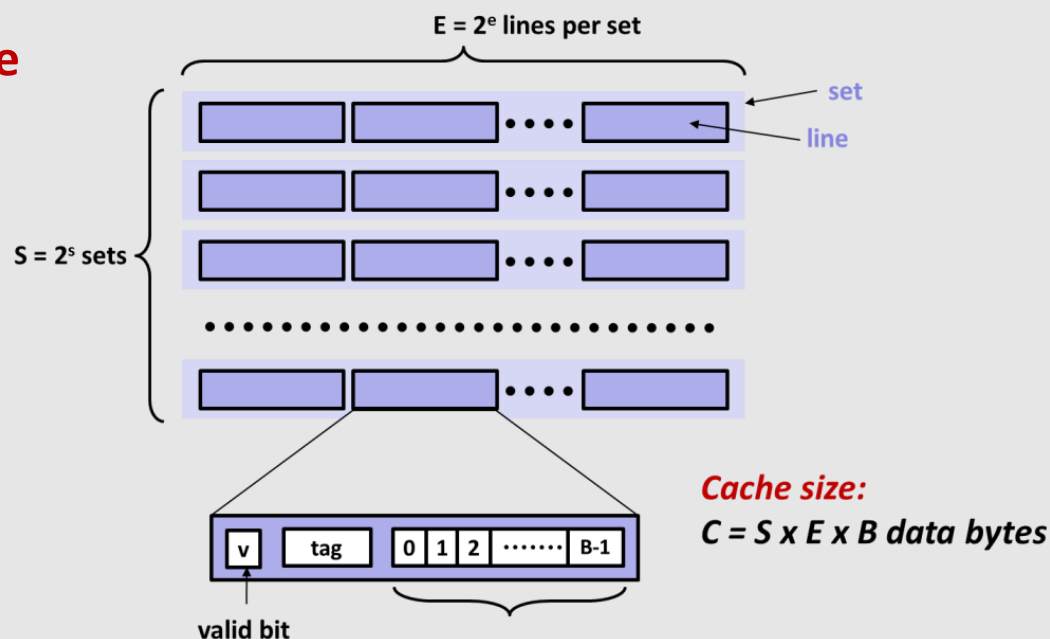
32 kB 8-way set associative
64 bytes/block
47 bit address range

$B = 64$

$S = 64, s = 6$

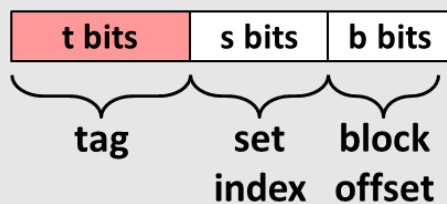
$E = 8, e = 3$

$C = 64 \times 64 \times 8 = 32,768$



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

Block offset:

0x??

Set index:

0x??

Tag:

0x??

Example: Core i7 L1 Data Cache

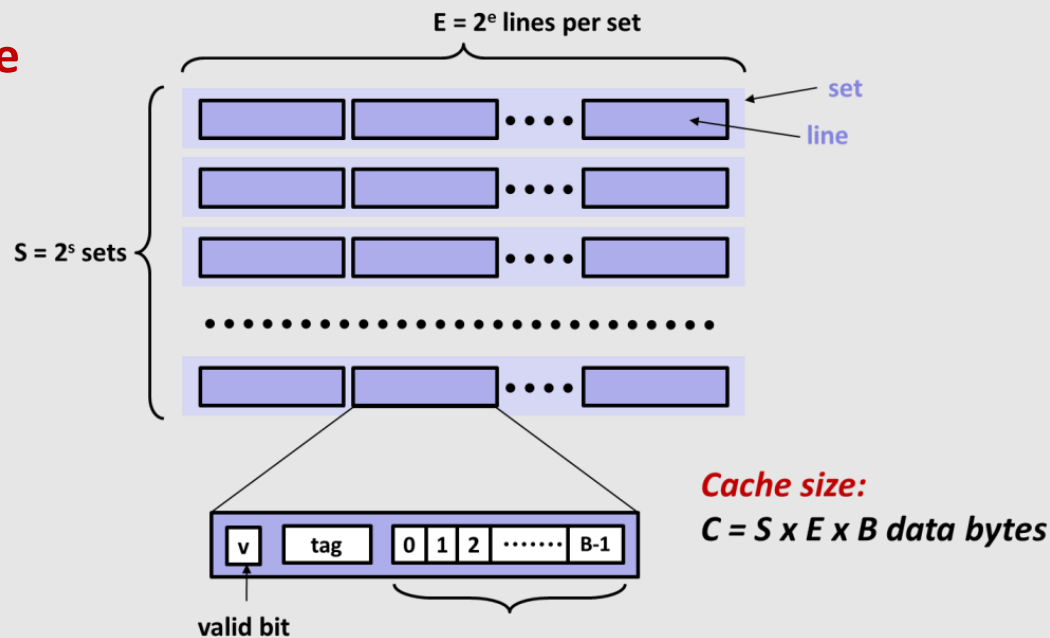
32 kB 8-way set associative
64 bytes/block
47 bit address range

$B = 64$

$S = 64, s = 6$

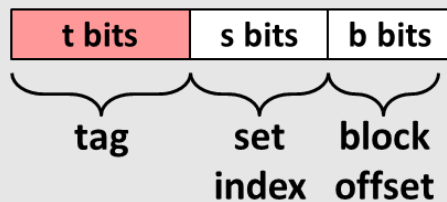
$E = 8, e = 3$

$C = 64 \times 64 \times 8 = 32,768$



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**

Cache Performance Metrics

■ Miss Rate

- Fraction of memory accesses not found in cache (misses / accesses)
= 1 – hit rate
- Typical numbers (as %):
 - 3-10% for L1
 - can be quite small (e.g., < 1%) for L2, depending on size, etc.

■ Hit Time

- Time to deliver a cached block to the processor
 - includes time to determine whether line is in cache
- Typical numbers:
 - 4 clock cycle 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!)

How Bad Can a Few Cache Misses Be?

- **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 this simplified example:
 - cache hit time of 1 cycle
 - miss penalty of 100 cycles
 - Average access time:
 - 97% hits: $1 \text{ cycle} + 0.03 \times 100 \text{ cycles} = 4 \text{ cycles}$
 - 99% hits: $1 \text{ cycle} + 0.01 \times 100 \text{ cycles} = 2 \text{ cycles}$
- **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

Quiz Time!

Canvas Quiz: Day 10 – Cache Memories

Today

- Cache organization and operation
- **Performance impact of caches**
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

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 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.

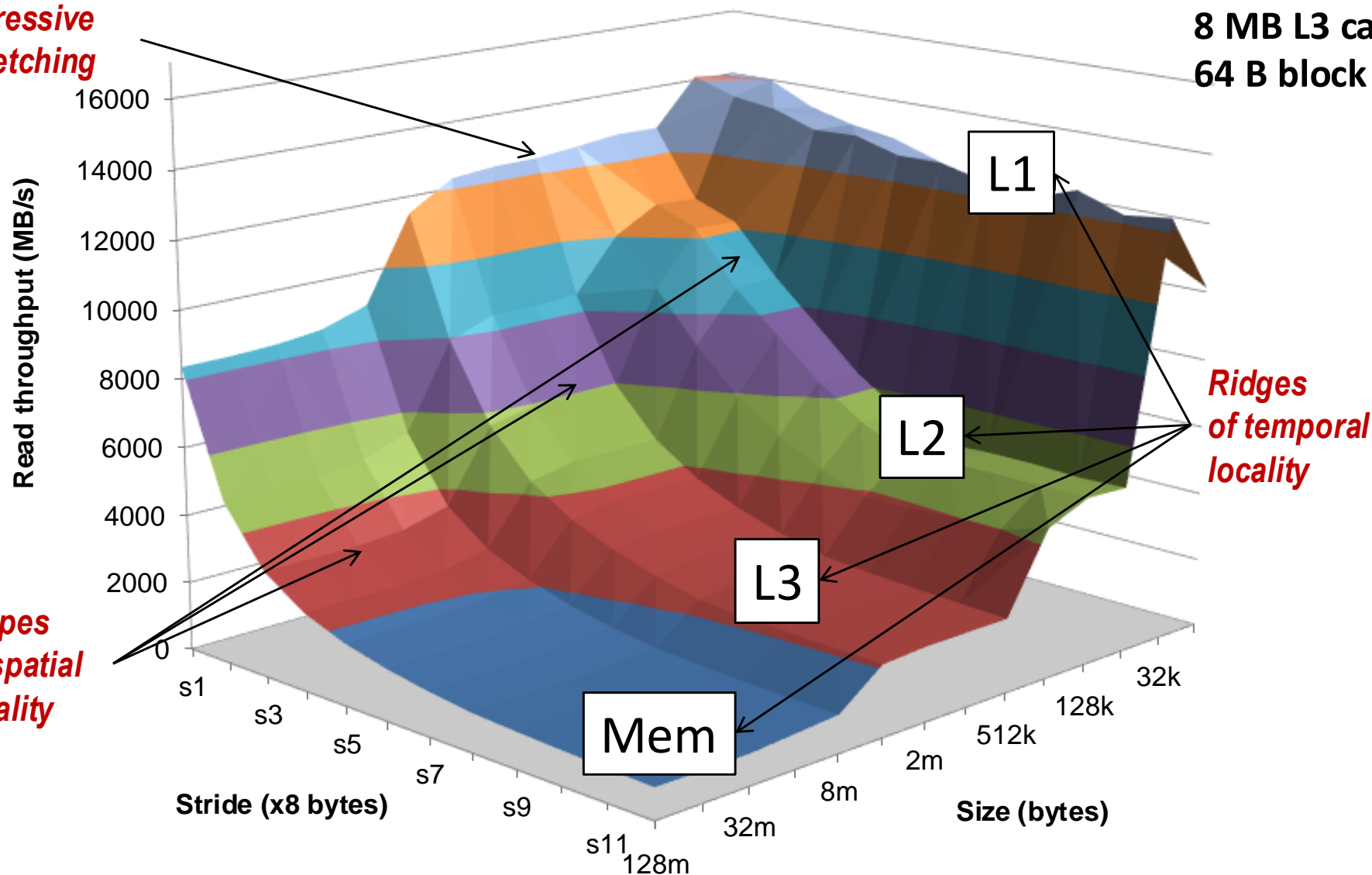
For each elems and stride:

1. Call test() once to warm up the caches.
2. Call test() again and measure the read throughput(MB/s)

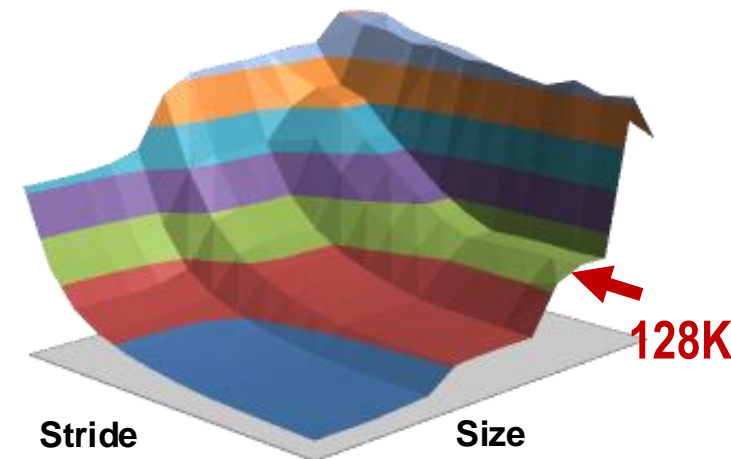
The Memory Mountain

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

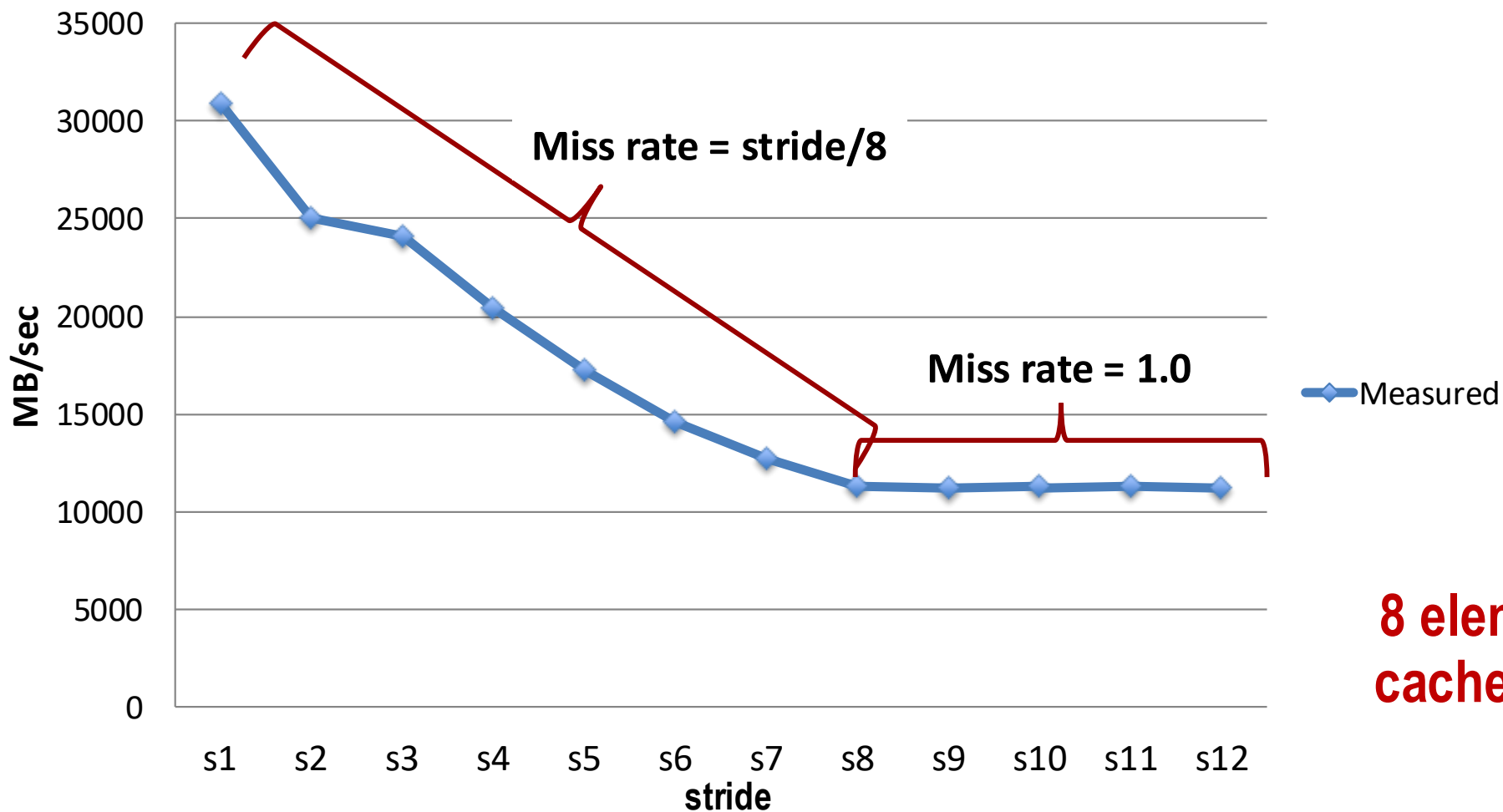
*Aggressive
prefetching*



Closer Look at Stride Effects



Throughput for size = 128K



**8 elems per
cache block**

Today

- Cache organization and operation
- Performance impact of caches
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

Matrix Multiplication Example

■ Description:

- Multiply $N \times N$ matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination
 - but 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*

matmult/mm.c

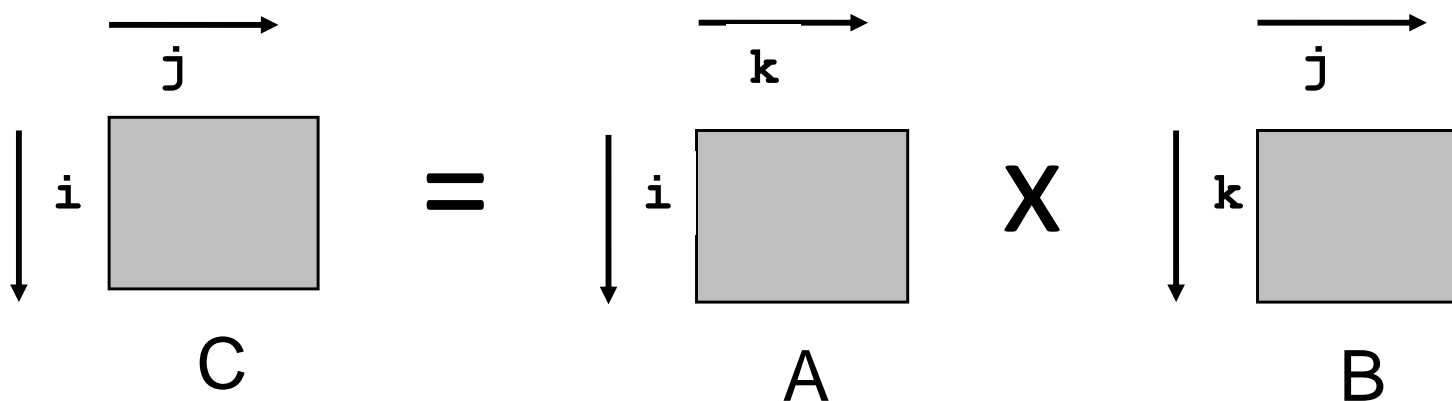
Miss Rate Analysis for Matrix Multiply

■ Assume:

- Block size = 32B (big enough for four doubles)
- 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



Layout of C Arrays in Memory (review)

- **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) > sizeof(a_{ij}) bytes, exploit spatial locality
 - miss rate = sizeof(a_{ij}) / B
- **Stepping through rows in one column:**
 - `for (i = 0; i < n; i++)`
 `sum += a[i][0];`
 - accesses distant elements
 - no spatial locality!
 - miss rate = 1 (i.e. 100%)

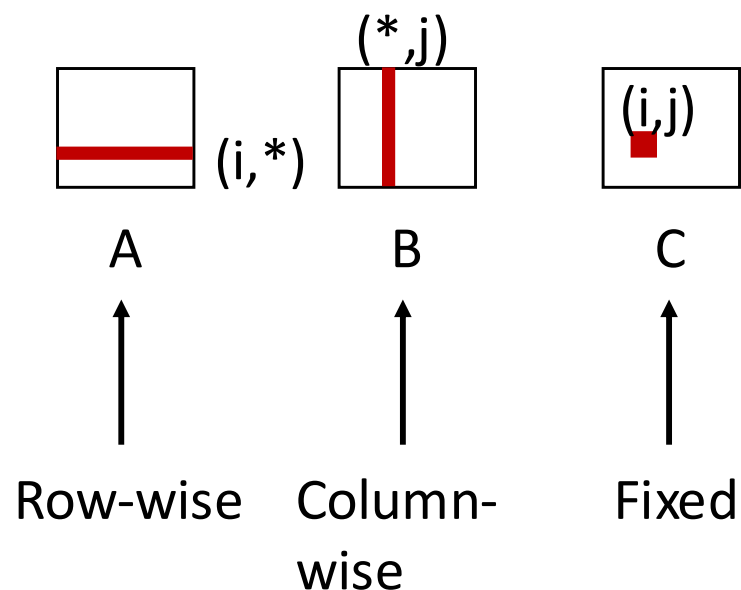
Matrix Multiplication (i j k)

```

/* 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;
  }
}
                                     matmult/mm.c

```

Inner loop:



Miss rate for inner loop iterations:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Block size = 32B (four doubles)

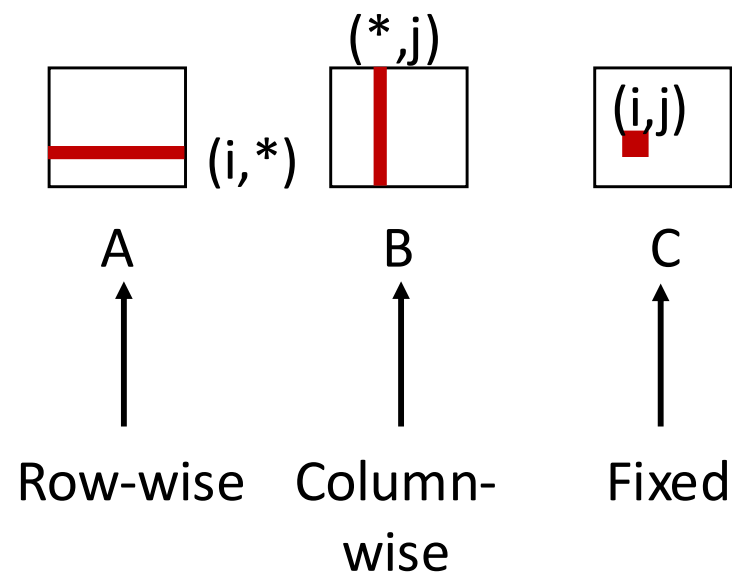
Matrix Multiplication (j i k)

```

/* j i k */
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
  }
}
                                     matmult/mm.c

```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Same analysis as i j k

Block size = 32B (four doubles)

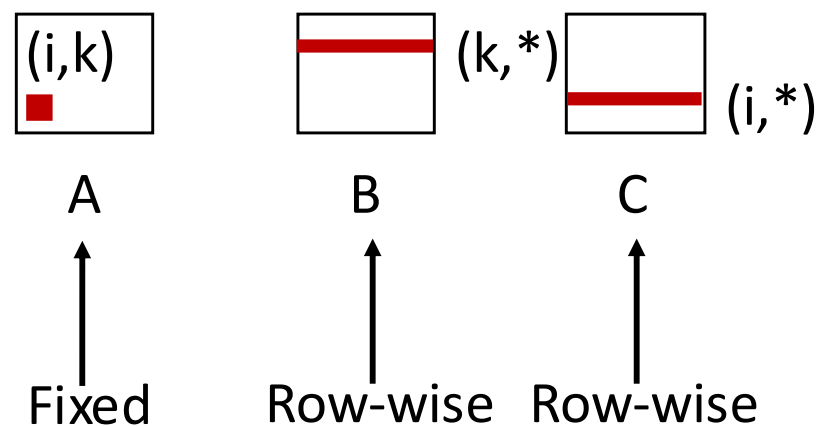
Matrix Multiplication (kij)

```

/* 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];
    }
}
                                     matmult/mm.c

```

Inner loop:



Miss rate for inner loop iterations:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Block size = 32B (four doubles)

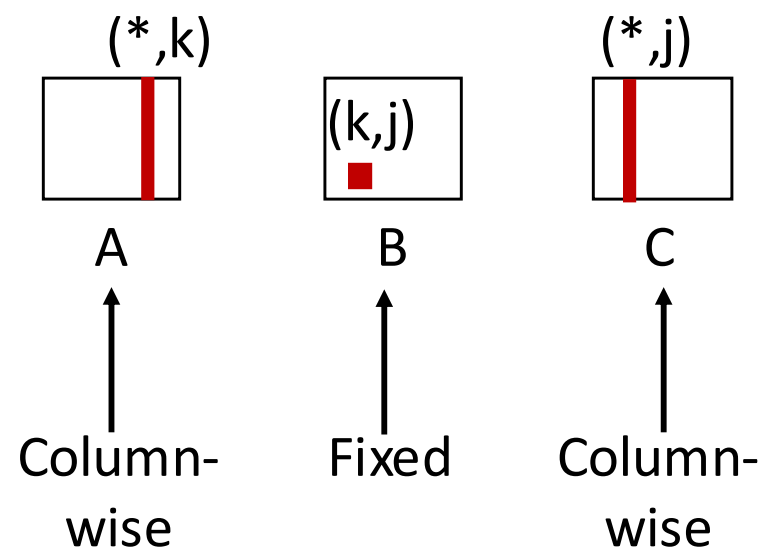
Matrix Multiplication (jki)

```

/* 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;
  }
}
                                     matmult/mm.c

```

Inner loop:



Miss rate for inner loop iterations:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Block size = 32B (four doubles)

Summary of Matrix Multiplication

```

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 (& jik):

- 2 loads, 0 stores
- avg misses/iter = **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 (& ikj):

- 2 loads, 1 store
- avg misses/iter = **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;
  }
}

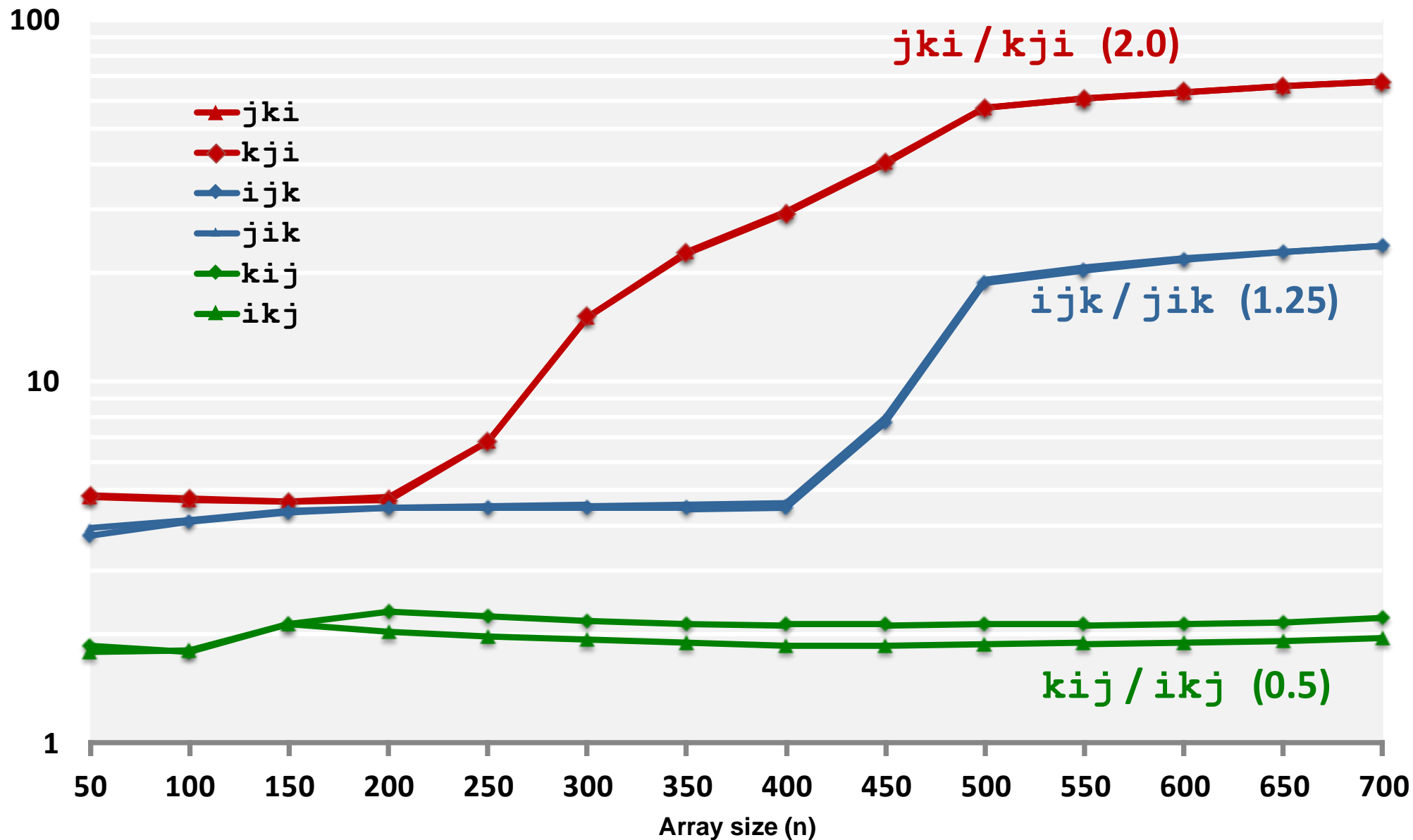
```

jki (& kji):

- 2 loads, 1 store
- avg misses/iter = **2.0**

Core i7 Matrix Multiply Performance

Cycles per inner loop iteration



Today

- Cache organization and operation
- Performance impact of caches
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

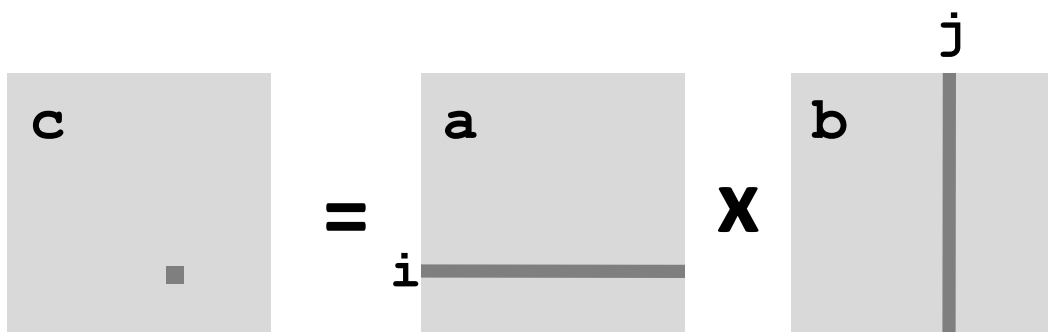
Example: Matrix Multiplication

```

c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}

```



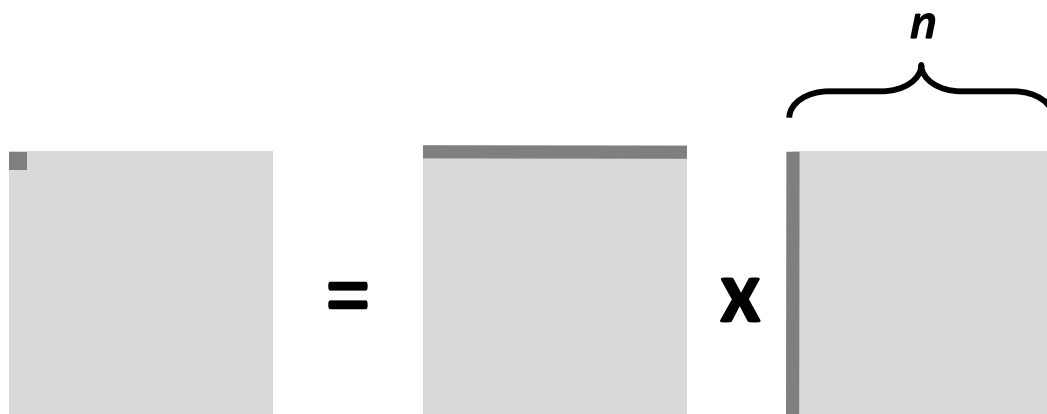
Cache Miss Analysis

■ Assume:

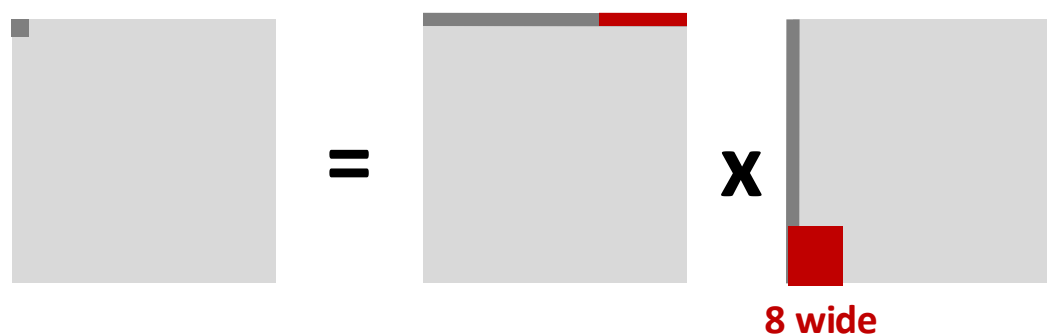
- Matrix elements are doubles
- Cache line = 8 doubles
- Cache size $C \ll n$ (much smaller than n)

■ First iteration:

- $n/8 + n = 9n/8$ misses



- Afterwards **in cache:**
(schematic)



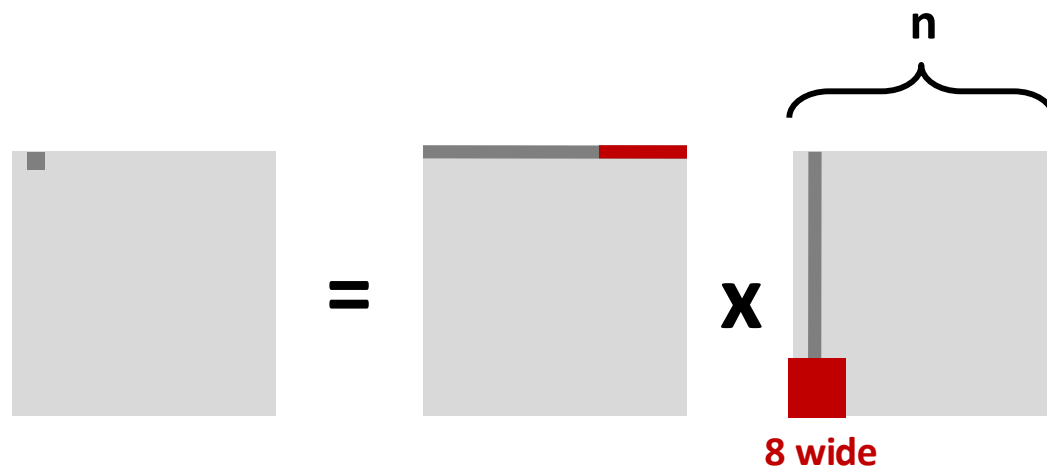
Cache Miss Analysis

■ Assume:

- Matrix elements are doubles
- Cache line = 8 doubles
- Cache size $C \ll n$ (much smaller than n)

■ Second iteration:

- Again:
 $n/8 + n = 9n/8$ misses



■ Total misses:

- $9n/8 n^2 = (9/8) n^3$

Blocked Matrix Multiplication

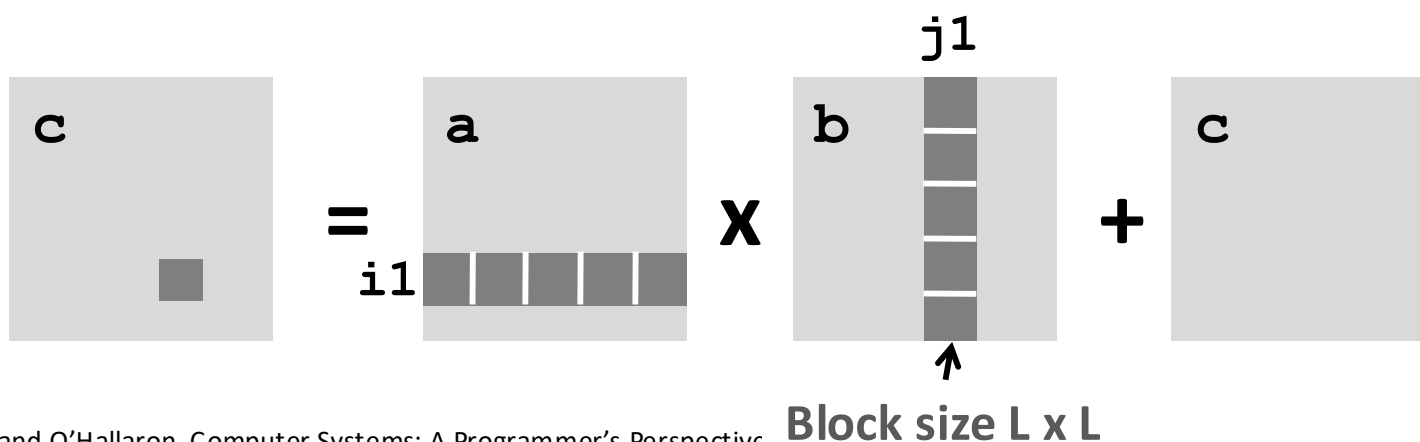
```

c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=L)
        for (j = 0; j < n; j+=L)
            for (k = 0; k < n; k+=L)
                /* L x L mini matrix multiplications */
                for (i1 = i; i1 < i+L; i1++)
                    for (j1 = j; j1 < j+L; j1++)
                        for (k1 = k; k1 < k+L; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}

```

matmult/bmm.c



Cache Miss Analysis

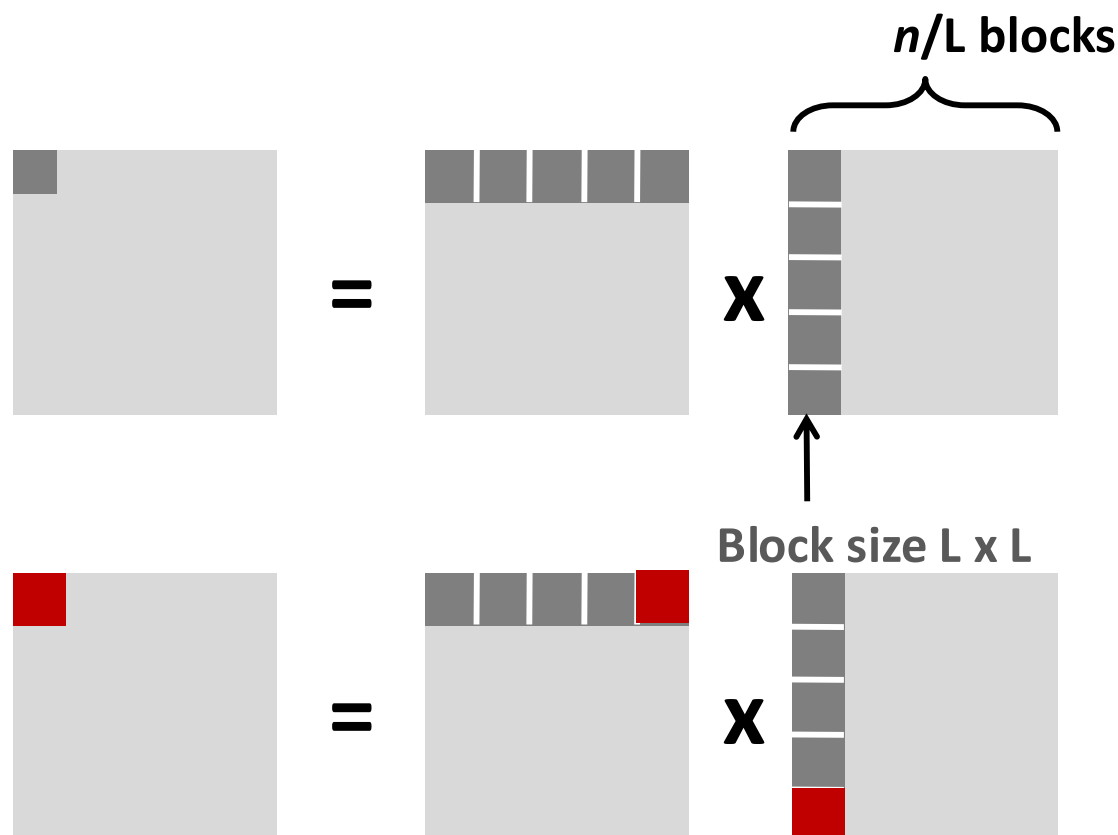
■ Assume:

- Cache line = 8 doubles. Blocking size $L \geq 8$
- Cache size $C \ll n$ (much smaller than n)
- Three blocks \blacksquare fit into cache: $3L^2 < C$

■ First (block) iteration:

- Misses per block: $L^2/8$
- Blocks per Iteration: $2n/L$
(omitting matrix c)
- Misses per Iteration:
 $2n/L \times L^2/8 = nL/4$

- Afterwards in cache
(schematic)



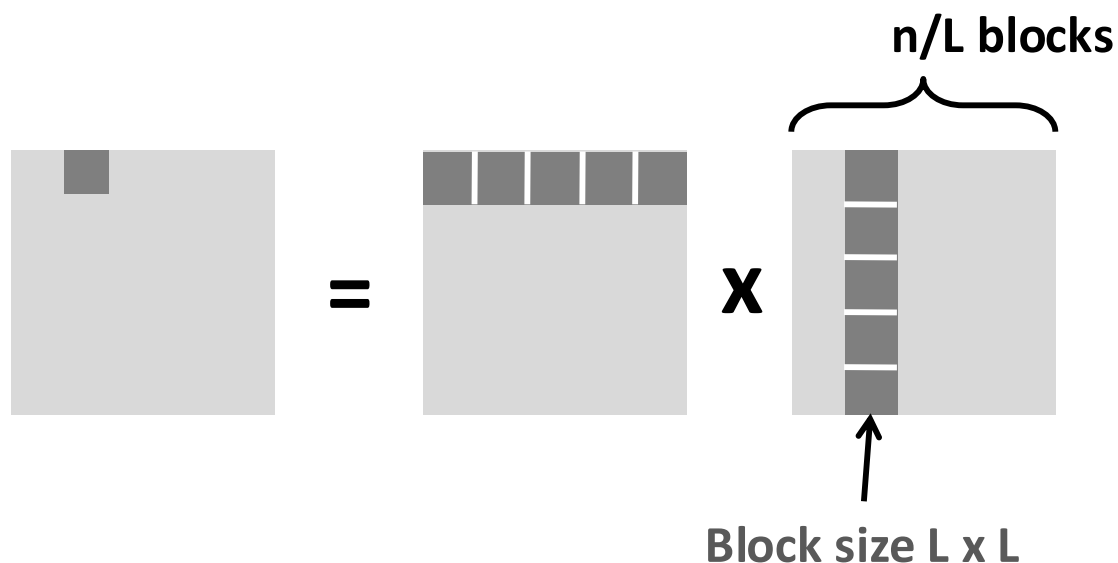
Cache Miss Analysis

■ Assume:

- Cache line = 8 doubles. Blocking size $L \geq 8$
- Cache size $C \ll n$ (much smaller than n)
- Three blocks \blacksquare fit into cache: $3L^2 < C$

■ Second (block) iteration:

- Same misses as first iteration
- $2n/L \times L^2/8 = nL/4$



■ Total misses:

- $nL/4$ misses per iteration $\times (n/L)^2$ iterations = $n^3/(4L)$ misses

Blocking Summary

- No blocking: $(9/8) n^3$ misses
- Blocking: $(1/(4L)) n^3$ misses

- Use largest block size L , such that L satisfies $3L^2 < C$
 - Fit three blocks in cache! Two input, one output.

- Reason for dramatic difference:
 - Matrix multiplication has inherent temporal locality:
 - Input data: $3n^2$, computation $2n^3$
 - Every array elements used $O(n)$ times!
 - But program has to be written properly

Cache Summary

- **Cache memories can have significant performance impact**
- **You can write your programs to exploit this!**
 - Focus on the inner loops, where bulk of computations and memory accesses occur.
 - Try to maximize spatial locality by reading data objects sequentially with stride 1.
 - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

Supplemental slides

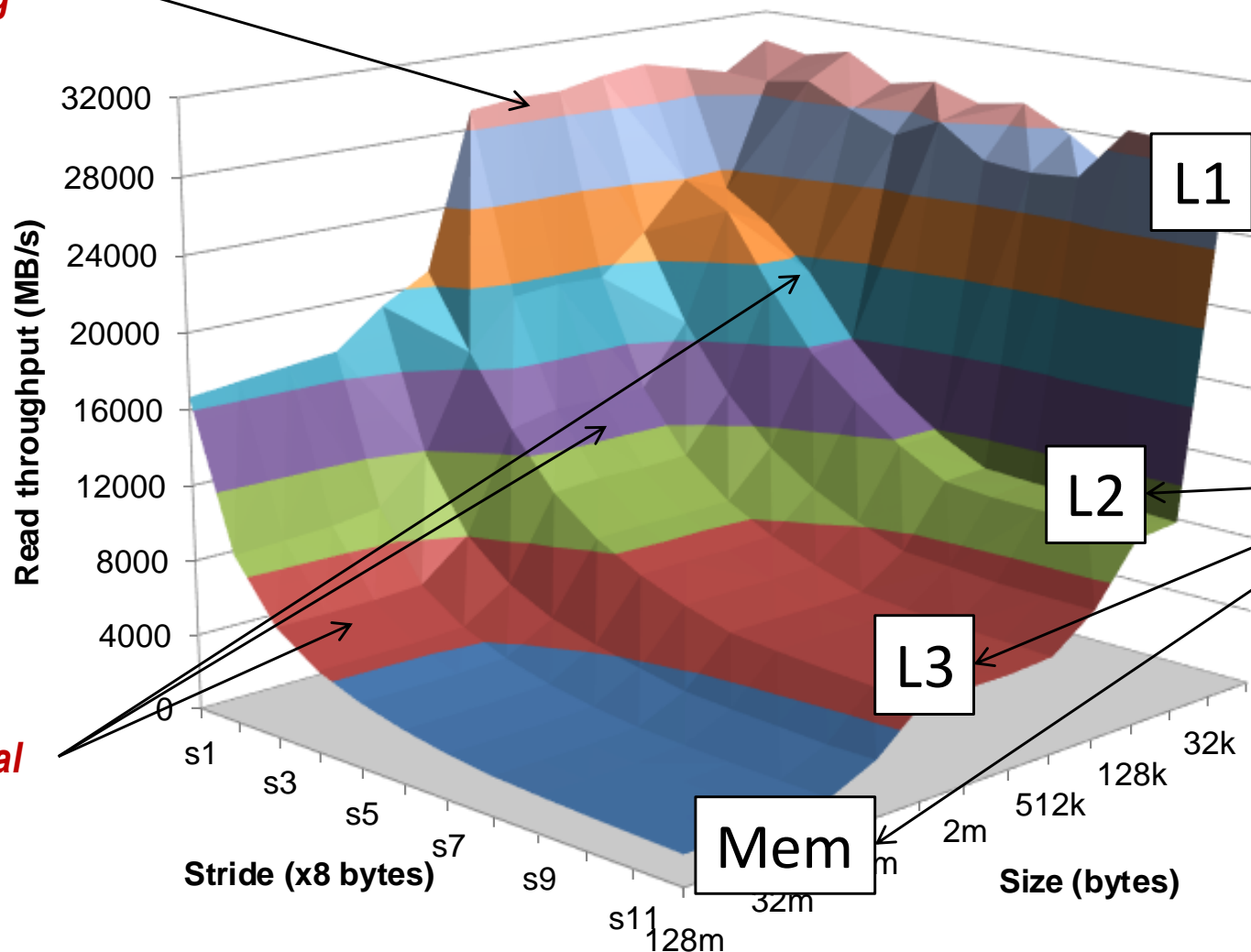
The Memory Mountain

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

*Aggressive
prefetching*

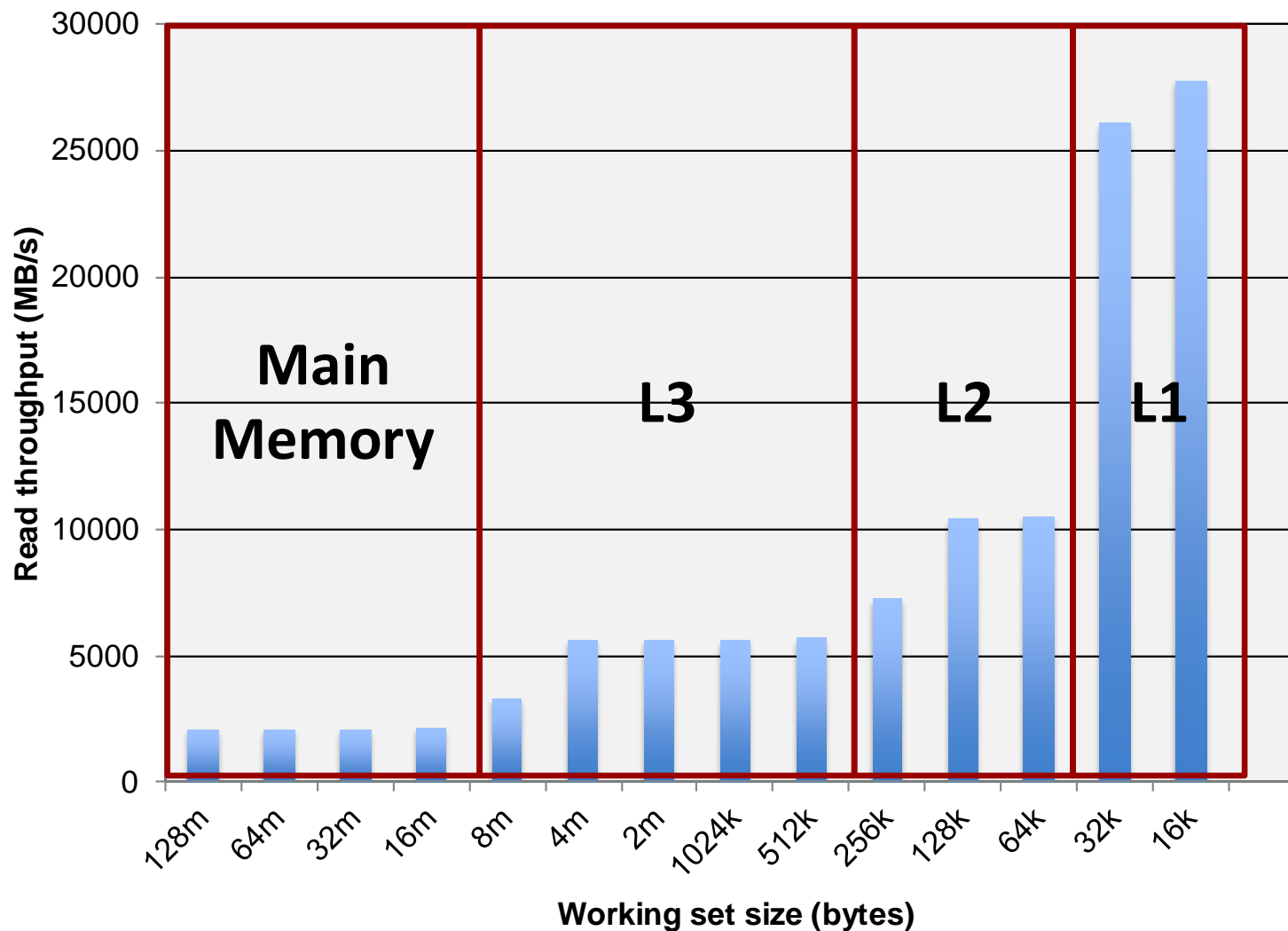
*Slopes
of spatial
locality*

*Ridges
of temporal
locality*



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

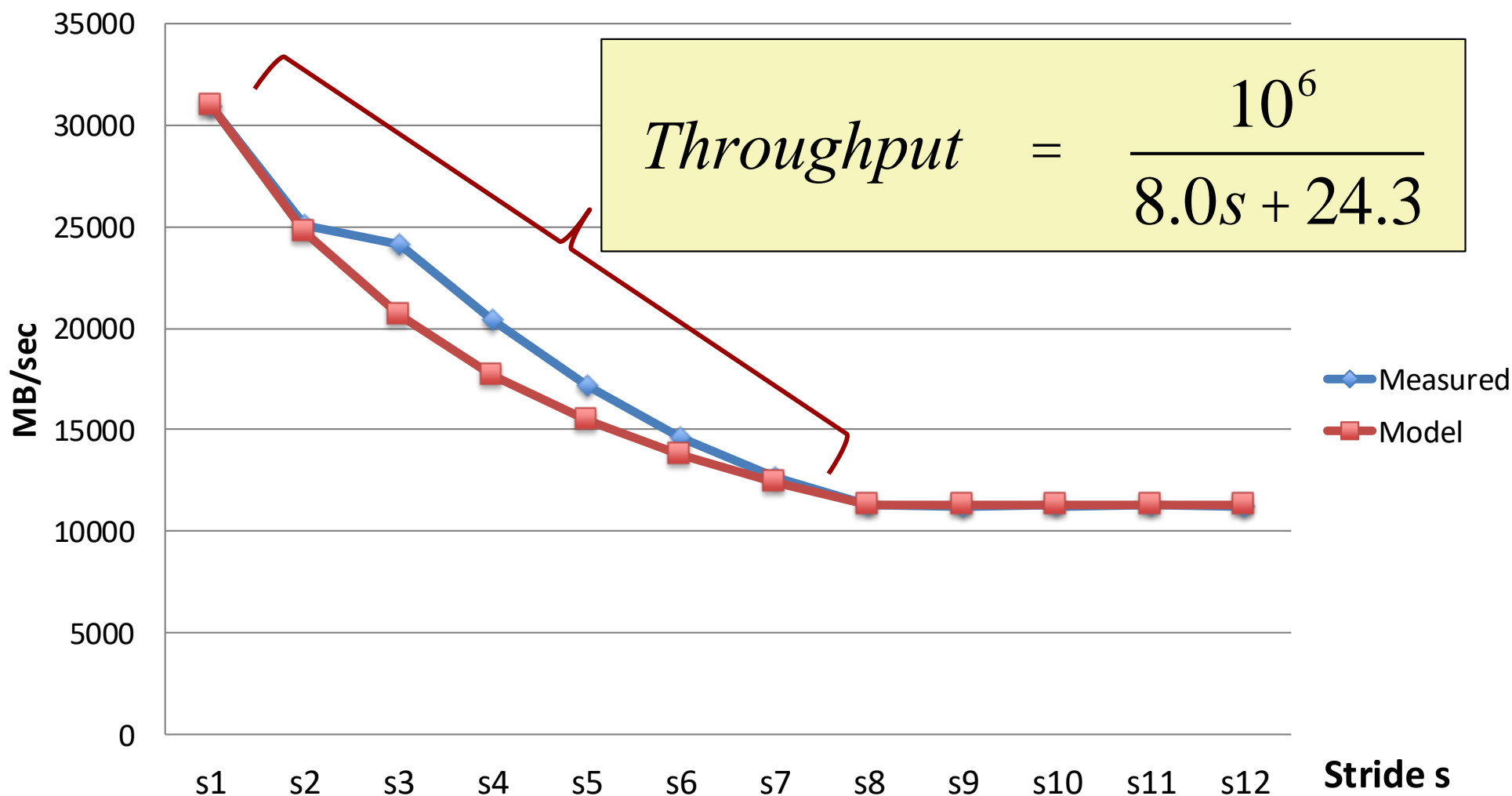


Slice through
memory
mountain with
stride=8

Modeling Block Size Effects from Memory Mountain

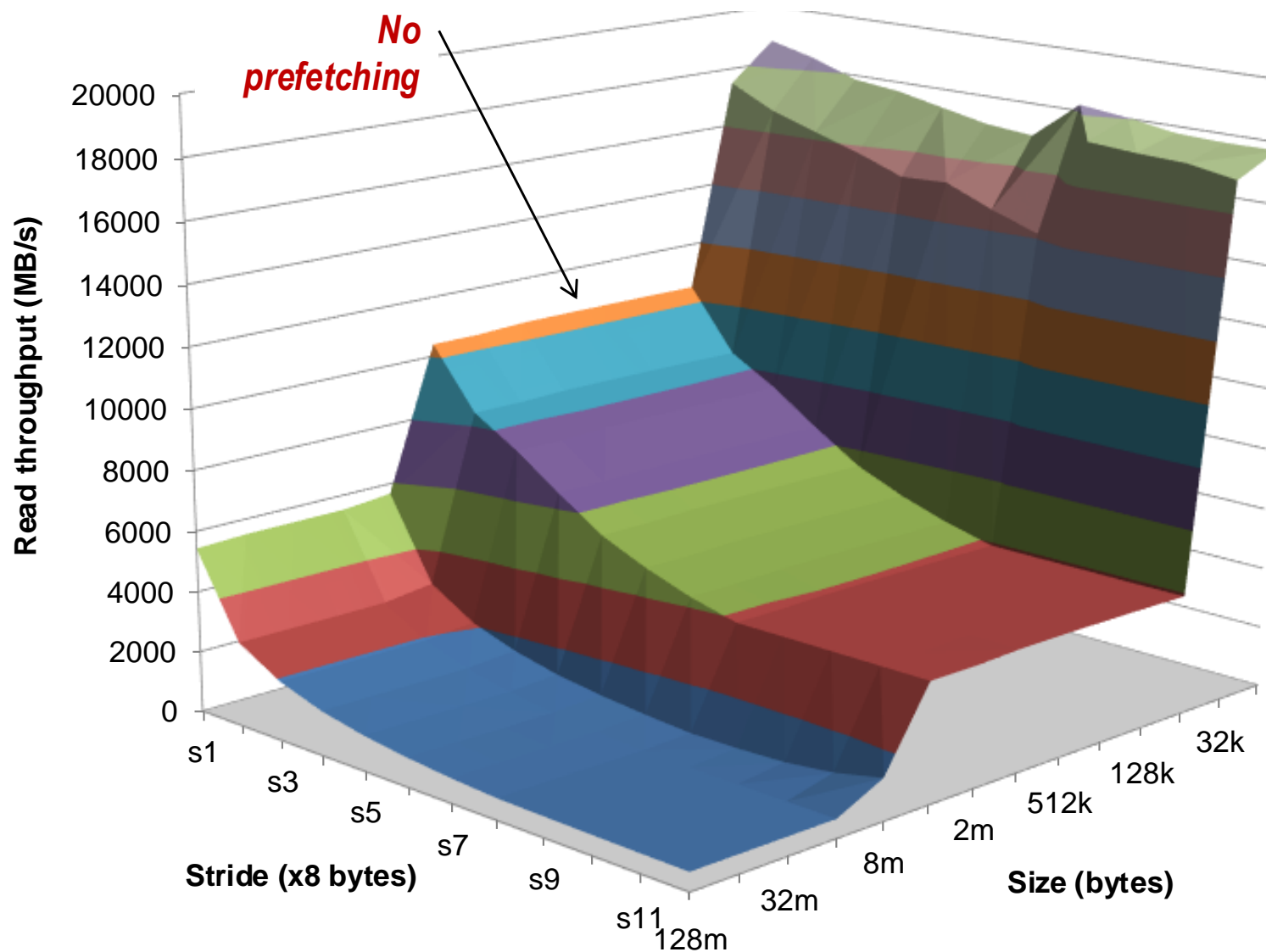
Core i7 Haswell
2.26 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Throughput for size = 128K



2008 Memory Mountain

Core 2 Duo
 2.4 GHz
 32 KB L1 d-cache
 6MB L2 cache
 64 B block size



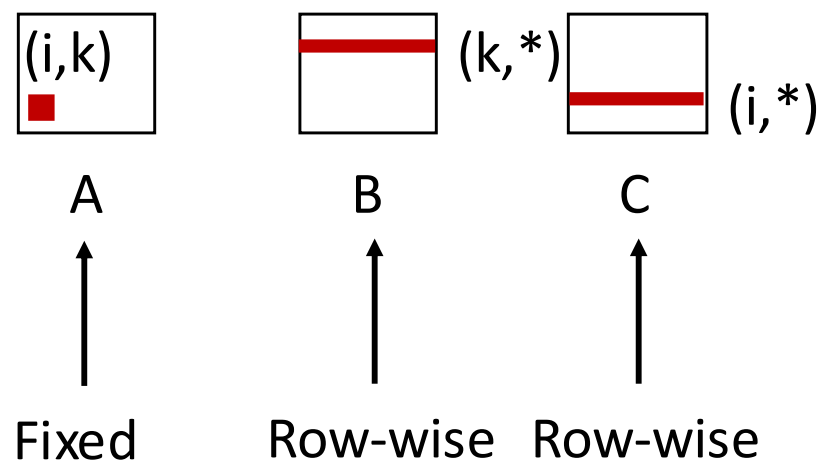
Matrix Multiplication (ikj)

```

/* 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];
  }
}
                                     matmult/mm.c

```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

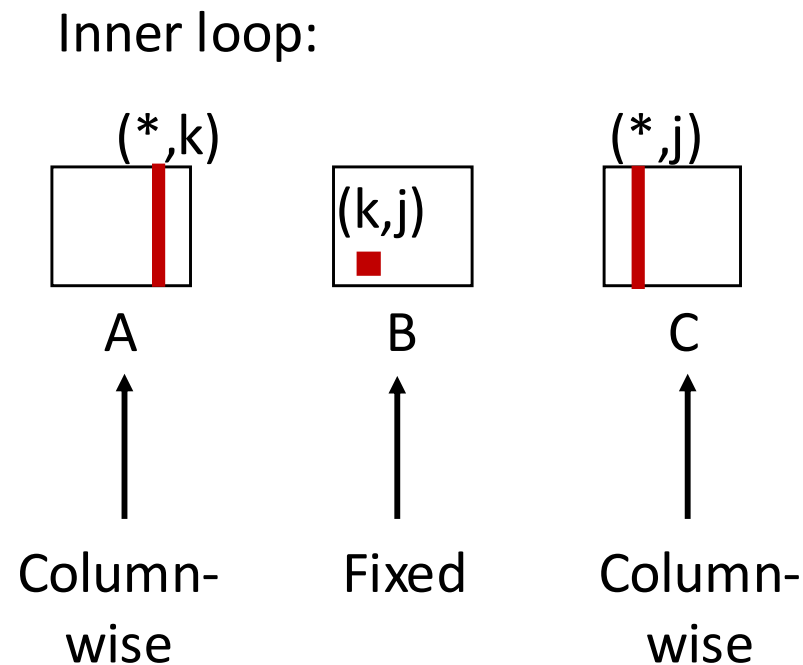
Block size = 32B (four doubles)

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;
  }
}
                                     matmult/mm.c

```



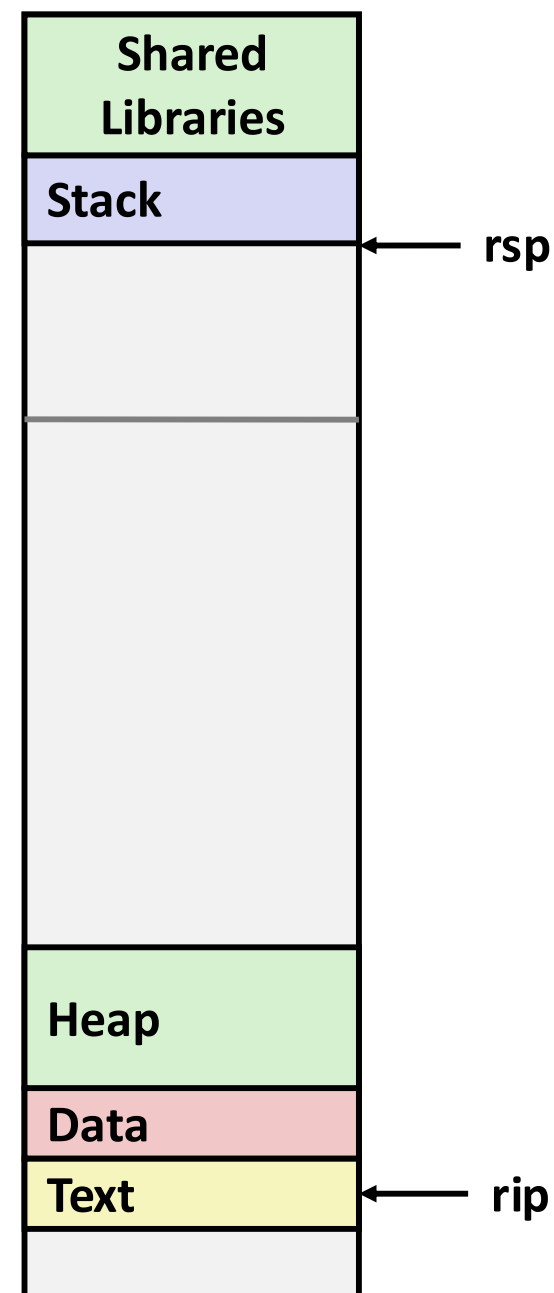
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Block size = 32B (four doubles)

Recap: Stack and instruction pointers

- The stack pointer (`%rsp`) points to the top of the stack
- The instruction pointer (`%rip`) points to the next instruction to be executed
- They are independent
 - But linked by `call` and `ret` instructions



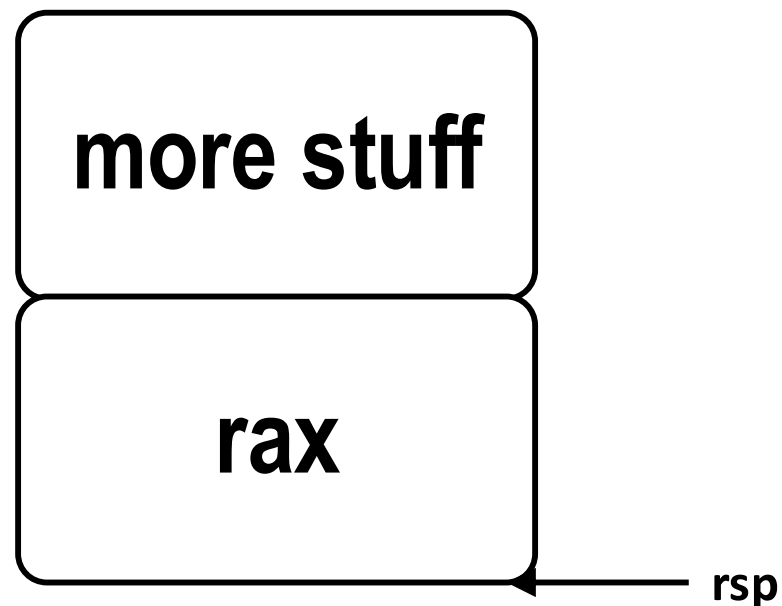
Recap: stack operations

- `push %rax =`



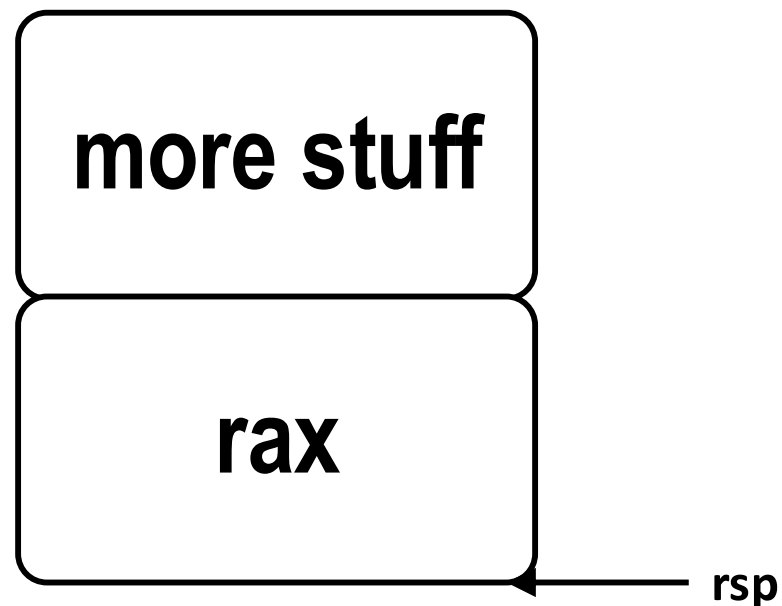
Recap: stack operations

- `push %rax =`
 - `sub %rsp, 8`
 - `mov %rax, (%rsp)`



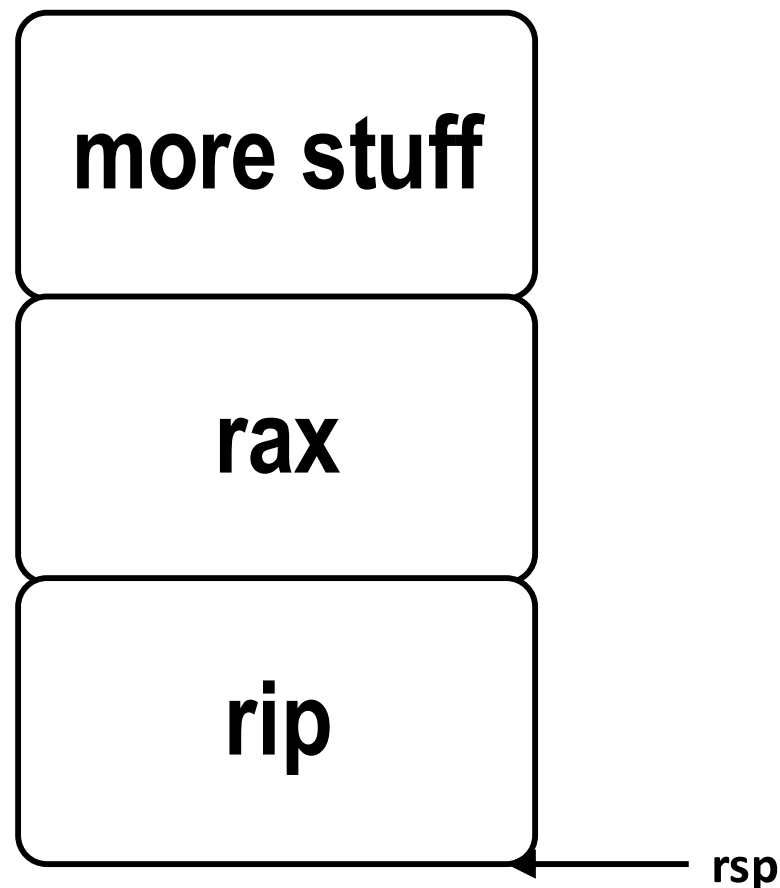
Recap: stack operations

- **push %rax =**
 - `sub %rsp, 8`
 - `mov %rax, (%rsp)`
- **call func =**



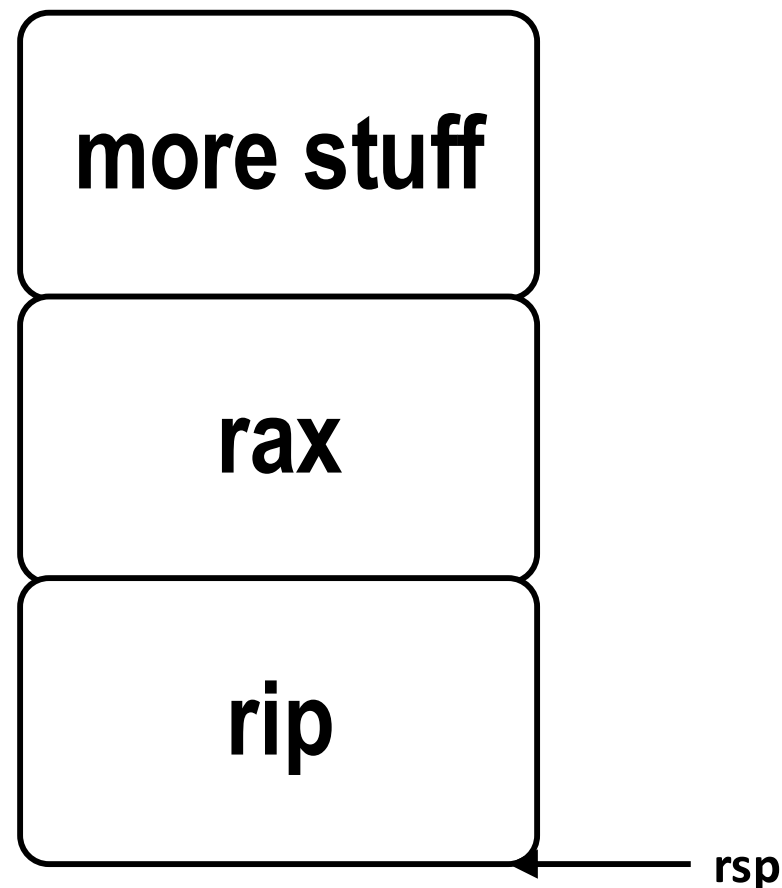
Recap: stack operations

- **push %rax =**
 - `sub %rsp, 8`
 - `mov %rax, (%rsp)`
- **call func =**
 - `sub %rsp, 8`
 - `mov %rip, (%rsp)`
 - `jmp func`



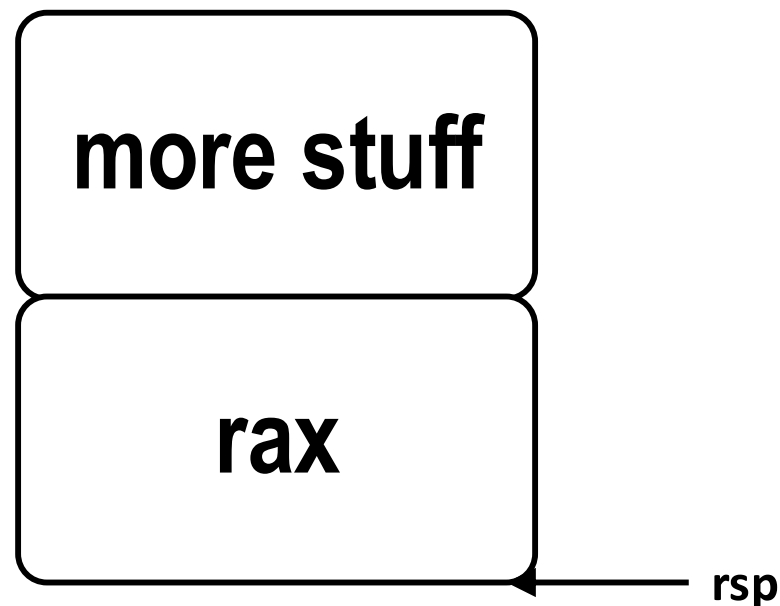
Recap: stack operations

- **push %rax =**
 - `sub %rsp, 8`
 - `mov %rax, (%rsp)`
- **call func =**
 - `sub %rsp, 8`
 - `mov %rip, (%rsp)`
 - `jmp func`
- **ret =**



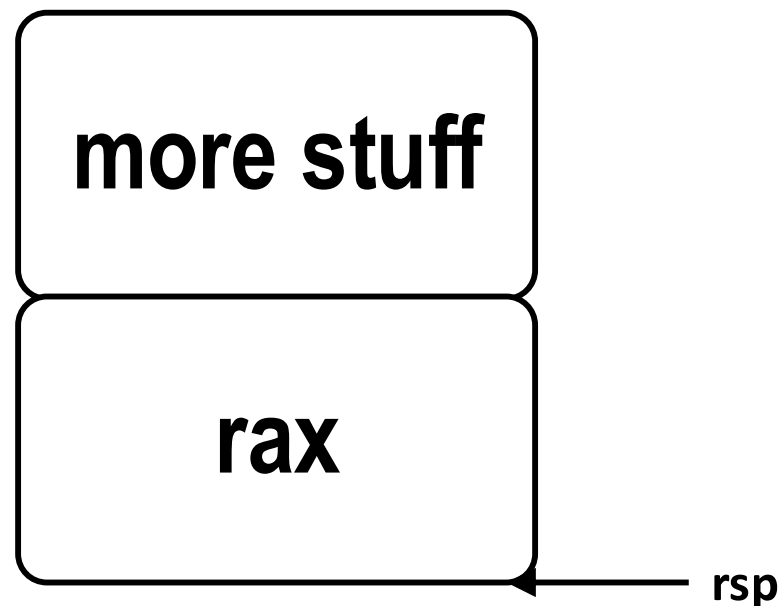
Recap: stack operations

- **push %rax =**
 - `sub %rsp, 8`
 - `mov %rax, (%rsp)`
- **call func =**
 - `sub %rsp, 8`
 - `mov %rip, (%rsp)`
 - `jmp func`
- **ret =**
 - `mov (%rsp), %rip`
 - `add %rsp, 8`



Recap: stack operations

- **push %rax =**
 - `sub %rsp, 8`
 - `mov %rax, (%rsp)`
- **call func =**
 - `sub %rsp, 8`
 - `mov %rip, (%rsp)`
 - `jmp func`
- **ret =**
 - `mov (%rsp), %rip`
 - `add %rsp, 8`
- **pop %rax =**



Recap: stack operations

- **push %rax =**
 - `sub %rsp, 8`
 - `mov %rax, (%rsp)`
- **call func =**
 - `sub %rsp, 8`
 - `mov %rip, (%rsp)`
 - `jmp func`
- **ret =**
 - `mov (%rsp), %rip`
 - `add %rsp, 8`
- **pop %rax =**
 - `mov (%rsp), %rax`
 - `add %rsp, 8`

