
WELCOME
TO
15-213
14-513
18-213
18-613
15-513

Cache Memories

15-213/14-513/15-513: Introduction to Computer Systems
10th Lecture, Sept 25, 2025

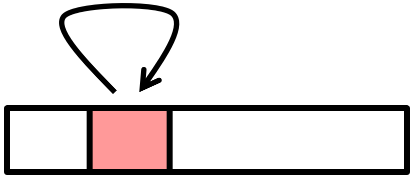
Today

- **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

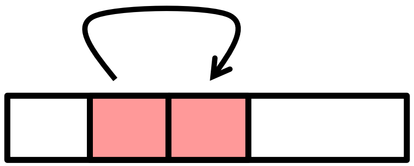
Recall: Locality

- **Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

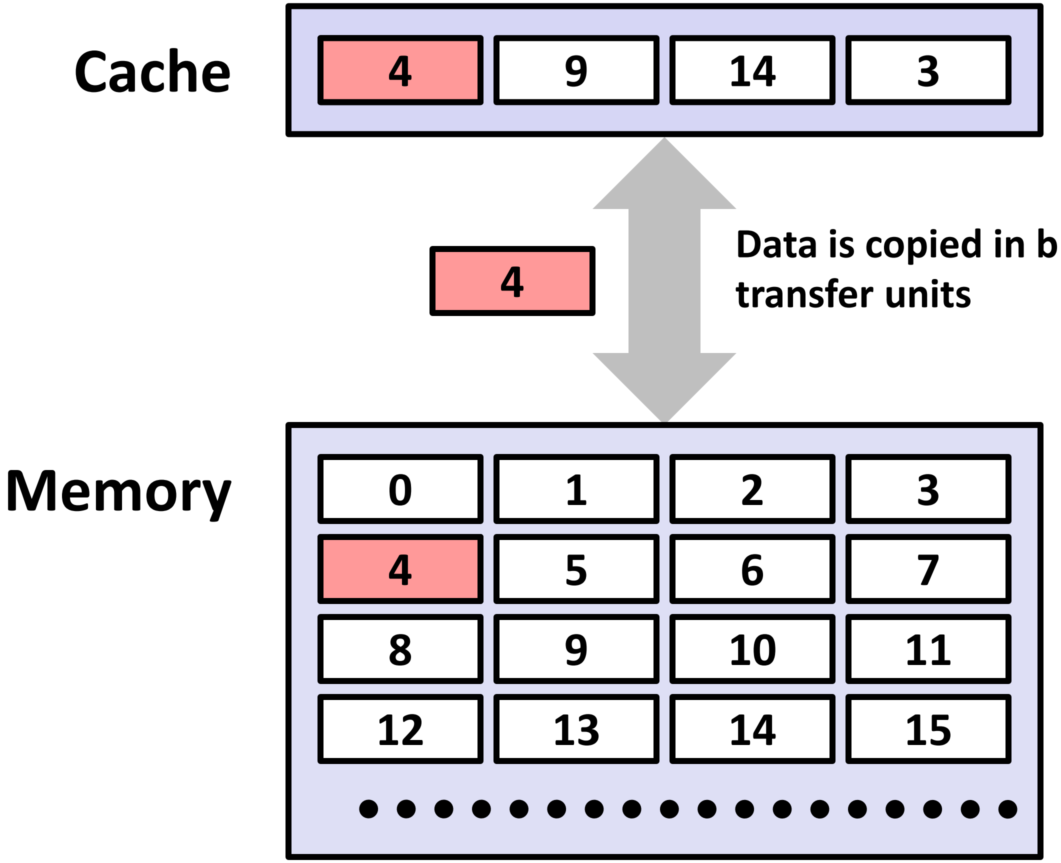
- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time



Recall: General Cache Concepts

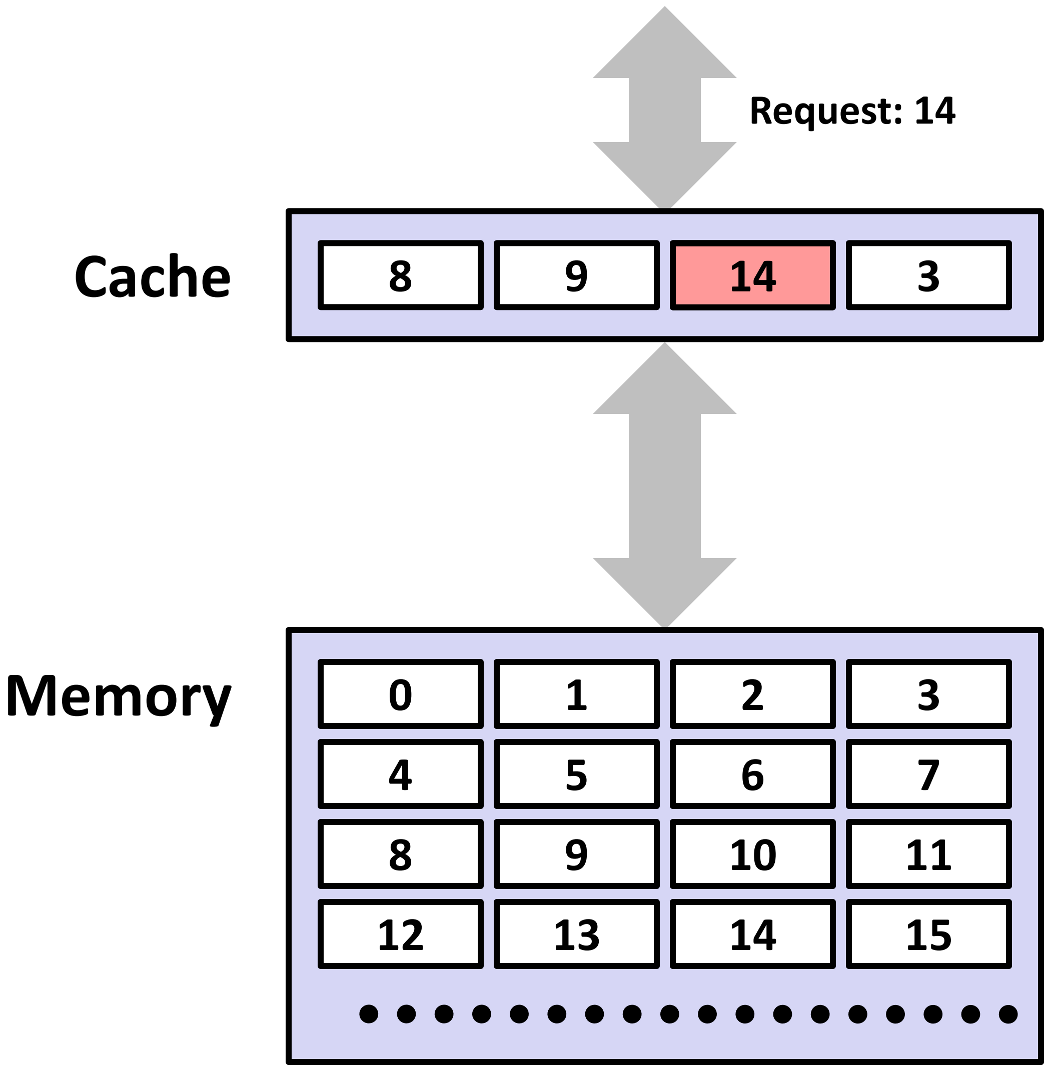


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks”

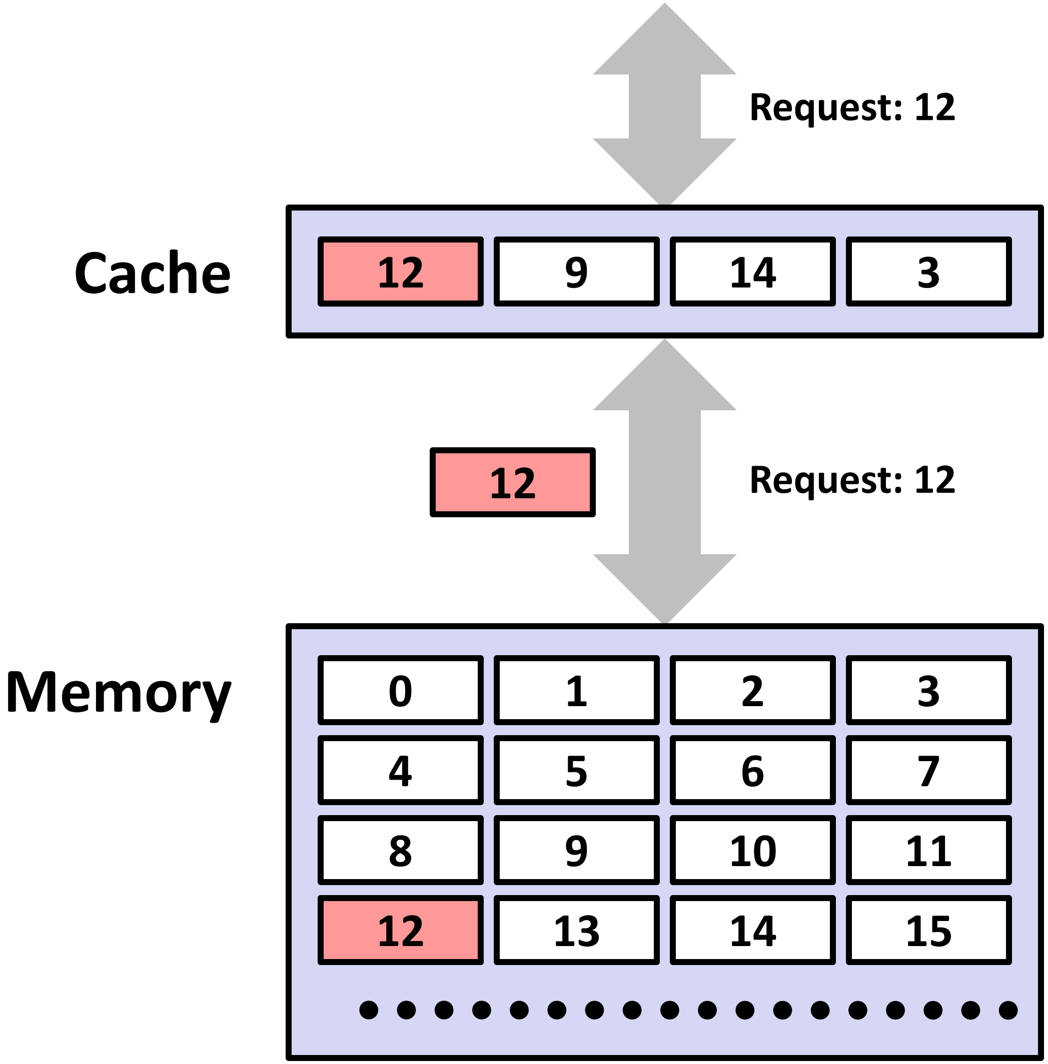
Recall: Cache Hit



Data in block b is needed

Block b is in cache:
Hit!

Recall: Cache Miss



Data in block b is needed

Block b is not in cache:
Miss!

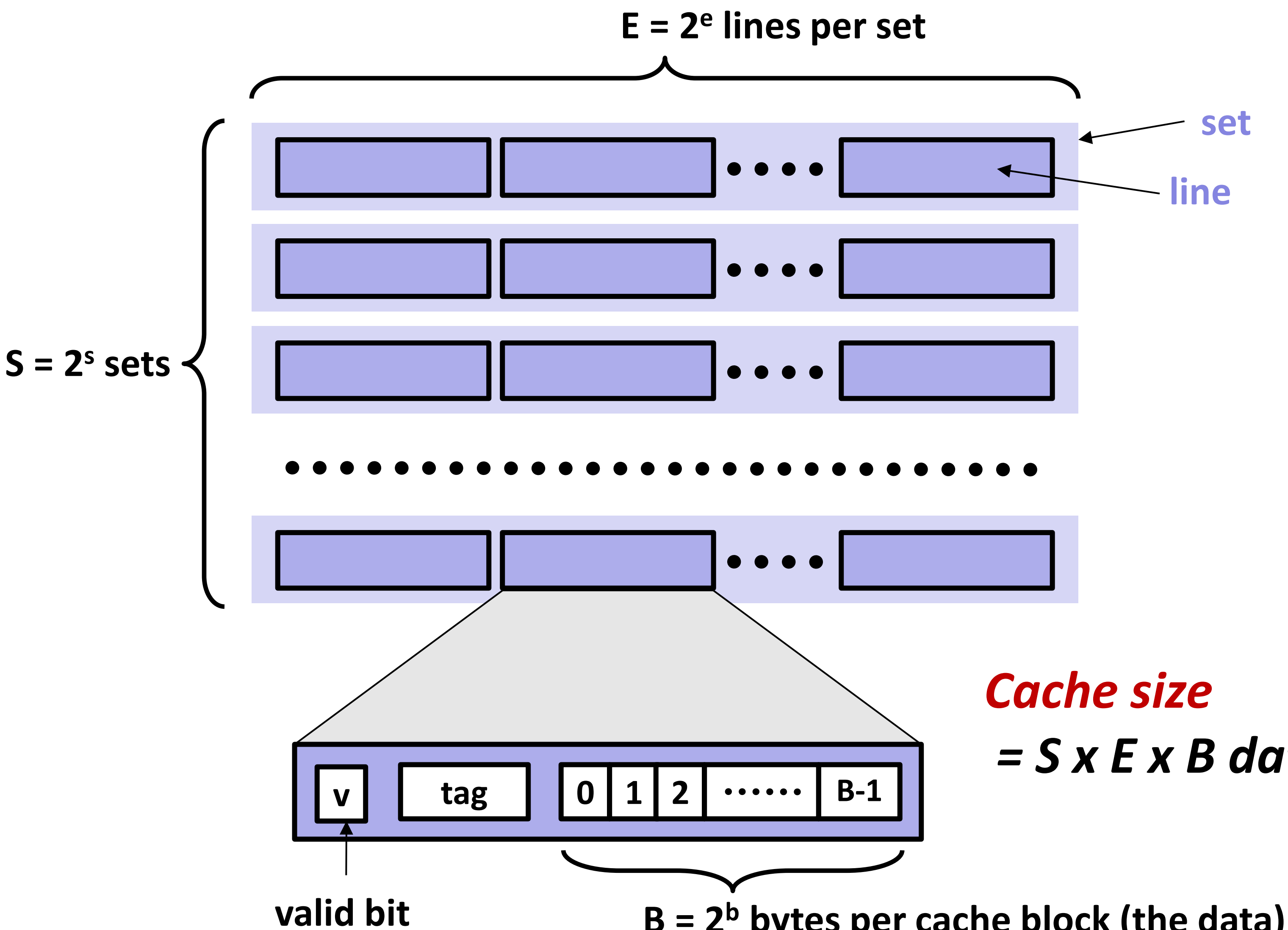
Block b is fetched from memory

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

Recall: General Caching Concepts: 3 Types of Cache Misses

- **Cold (compulsory) miss**
  - **Cold misses occur because this is the first reference to the block.**
    (Misses with infinitely large cache with no placement restrictions)
- **Capacity miss**
  - **Occurs when the set of active cache blocks is larger than the cache.**
    (***Additional*** misses from finite-sized cache with no placement restrictions)
- **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**
    (***Additional*** misses due to actual placement policy)

General Cache Organization (S, E, B)

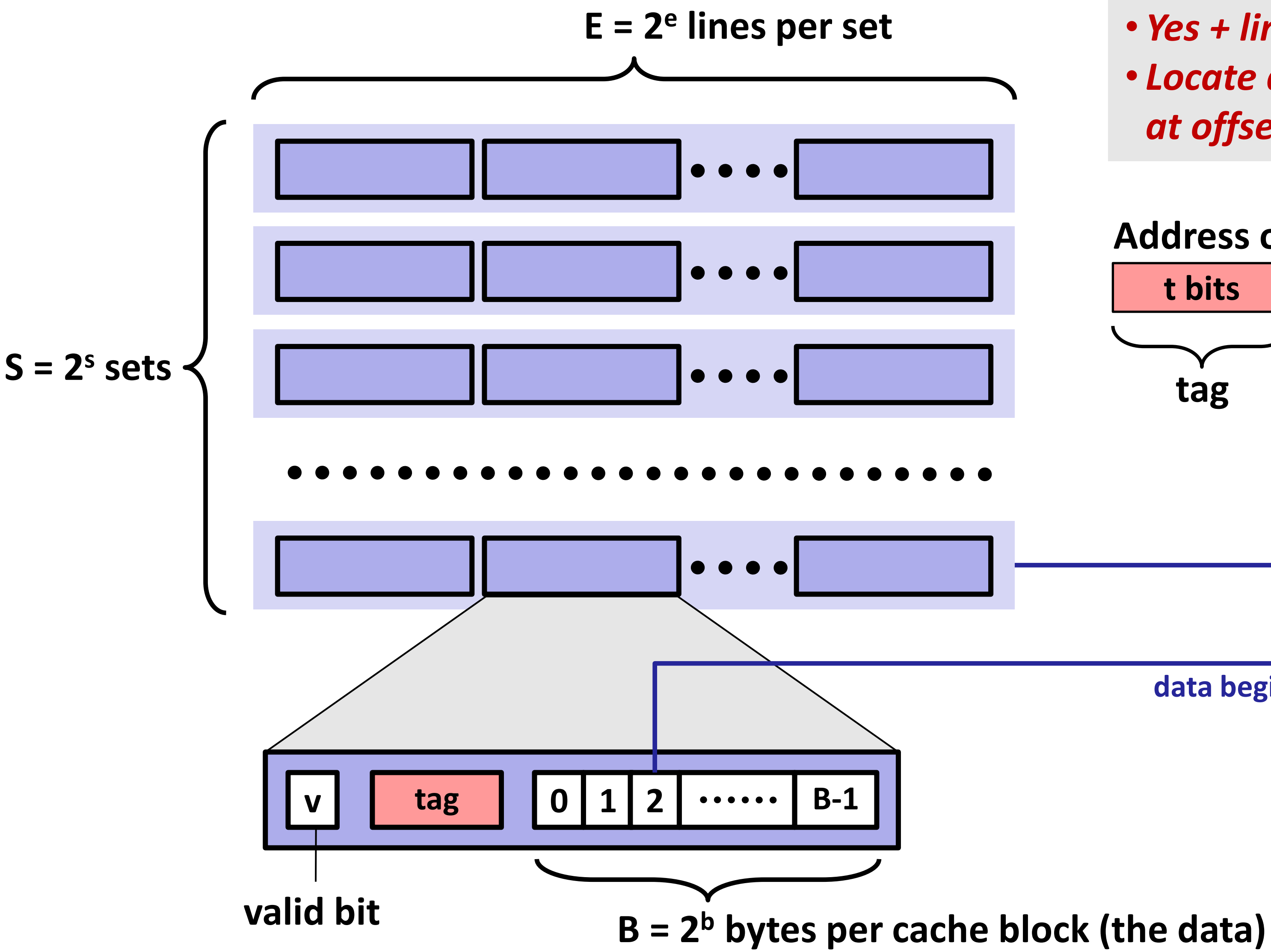


Cache size

= S x E x B data bytes

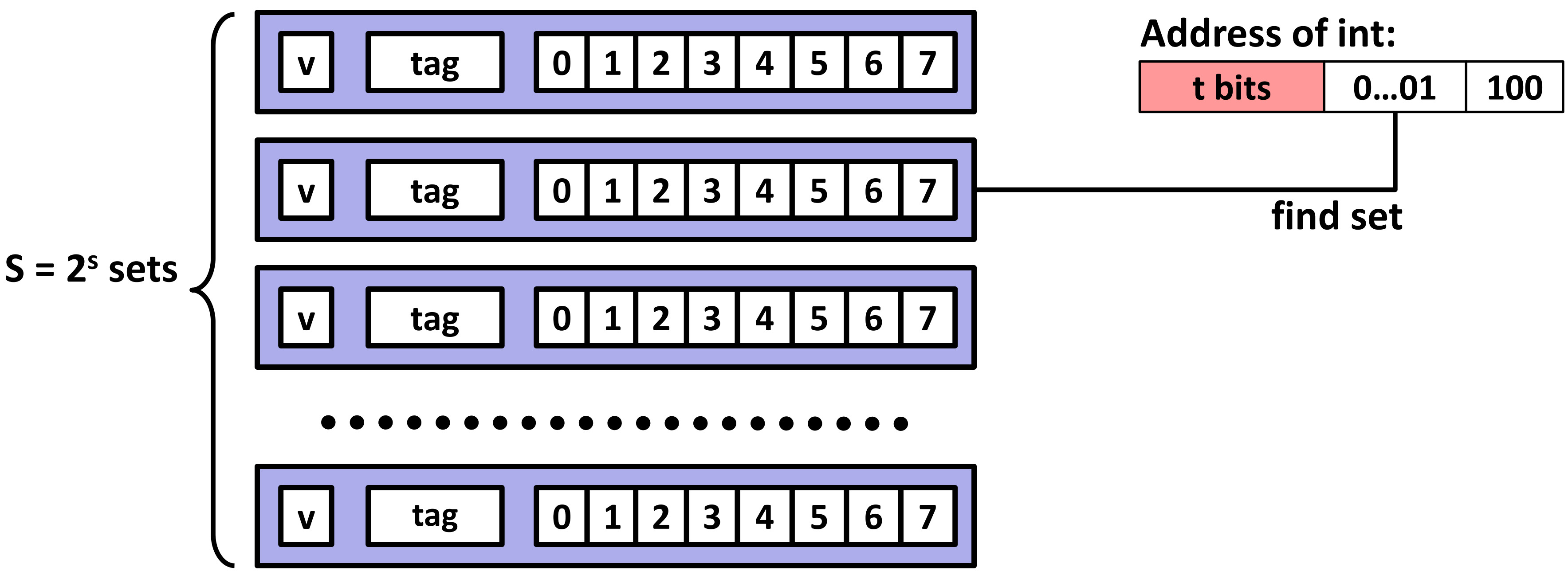
Cache Read

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



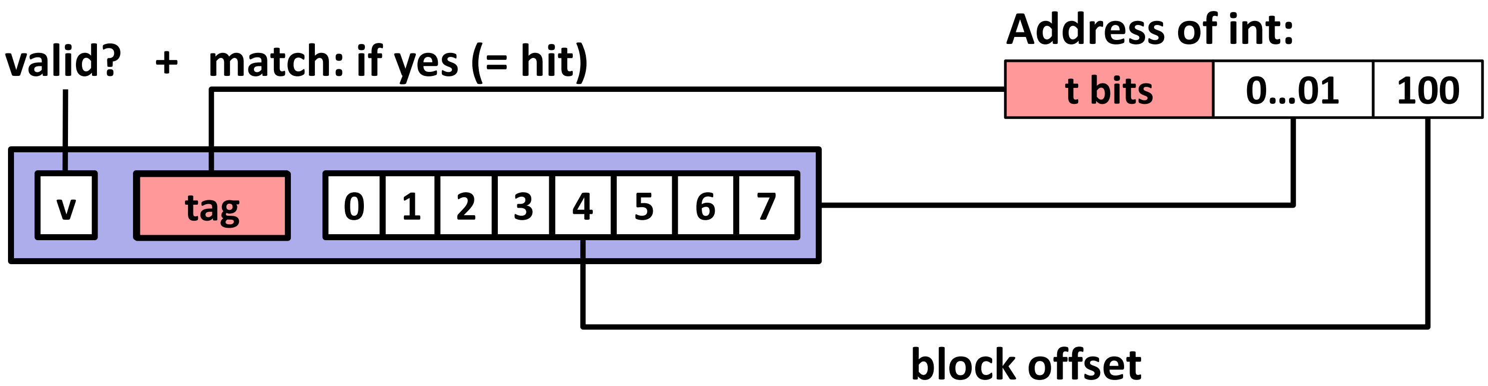
Example: Direct Mapped Cache (E = 1)

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



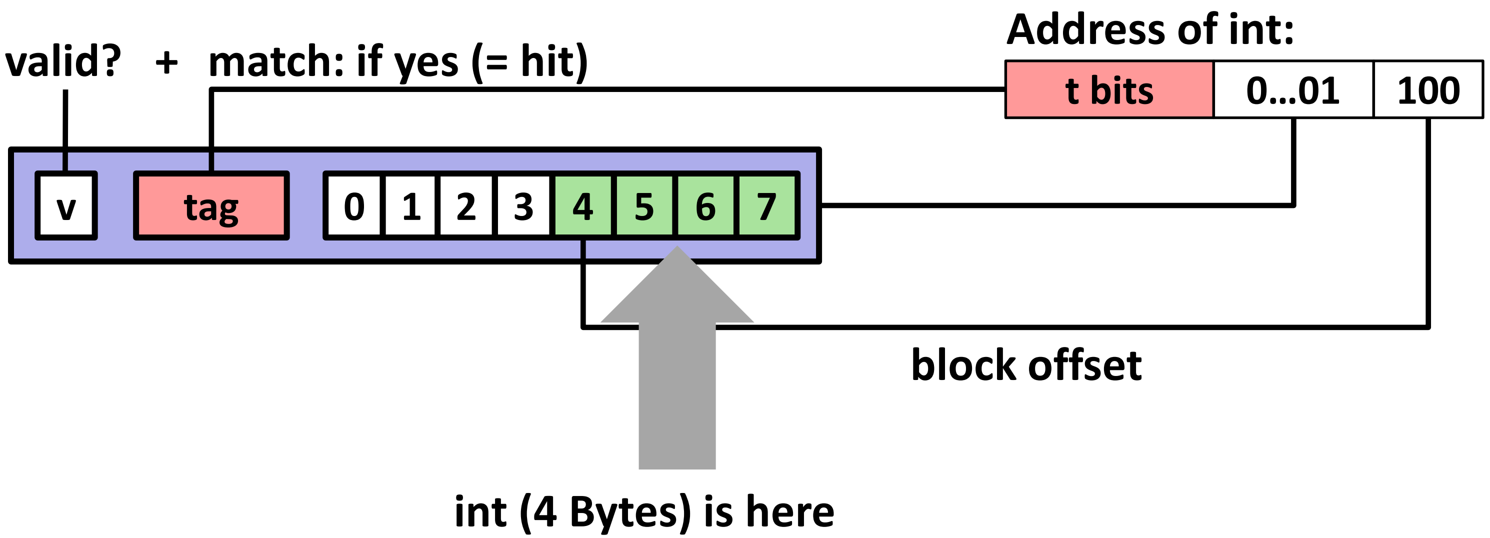
Example: Direct Mapped Cache (E = 1)

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):

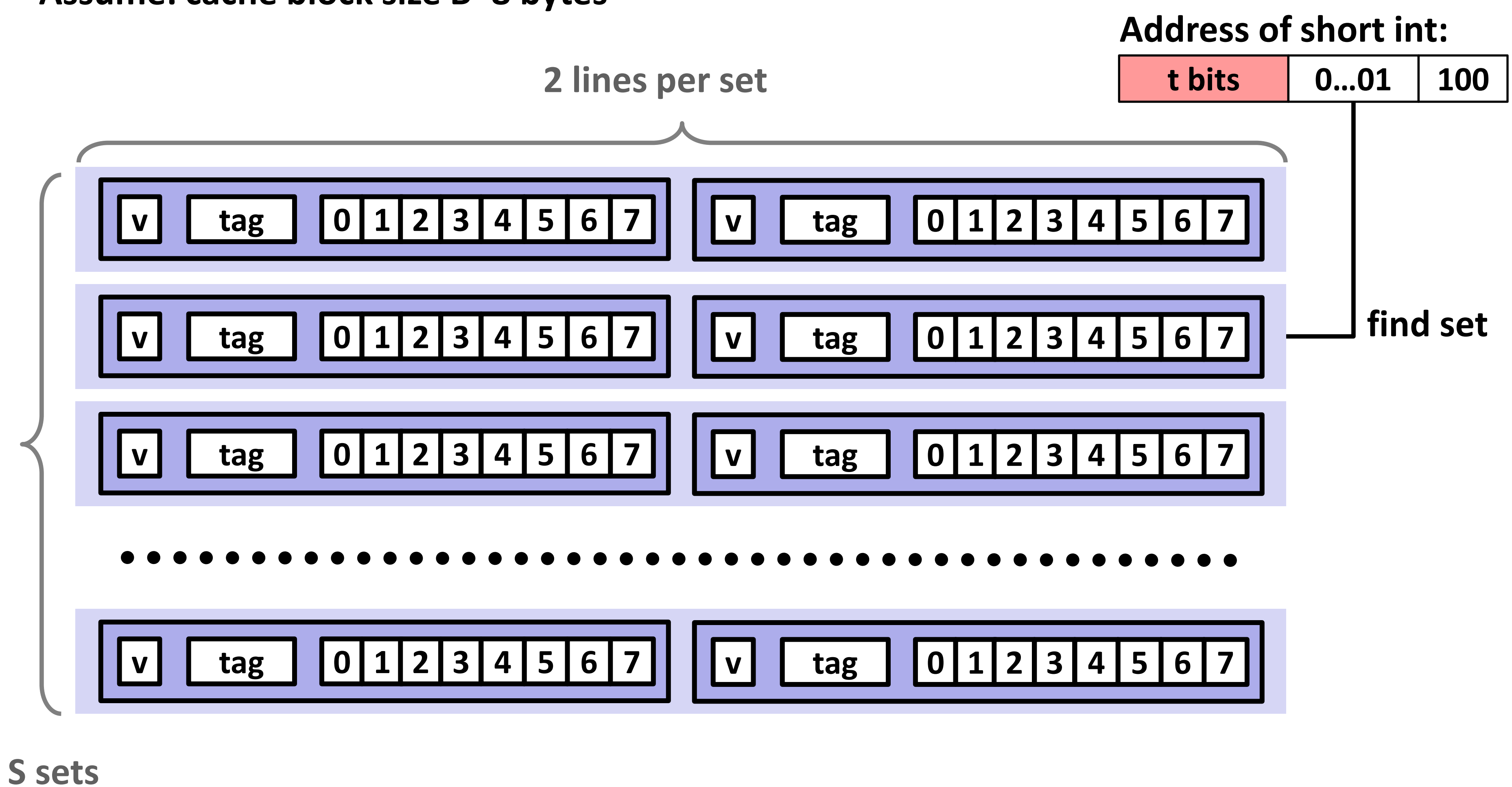
0	**[0000_2],**	miss
1	**[0001_2],**	hit
7	**[0111_2],**	miss
8	**[1000_2],**	miss
0	**[0000_2]**	miss

	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

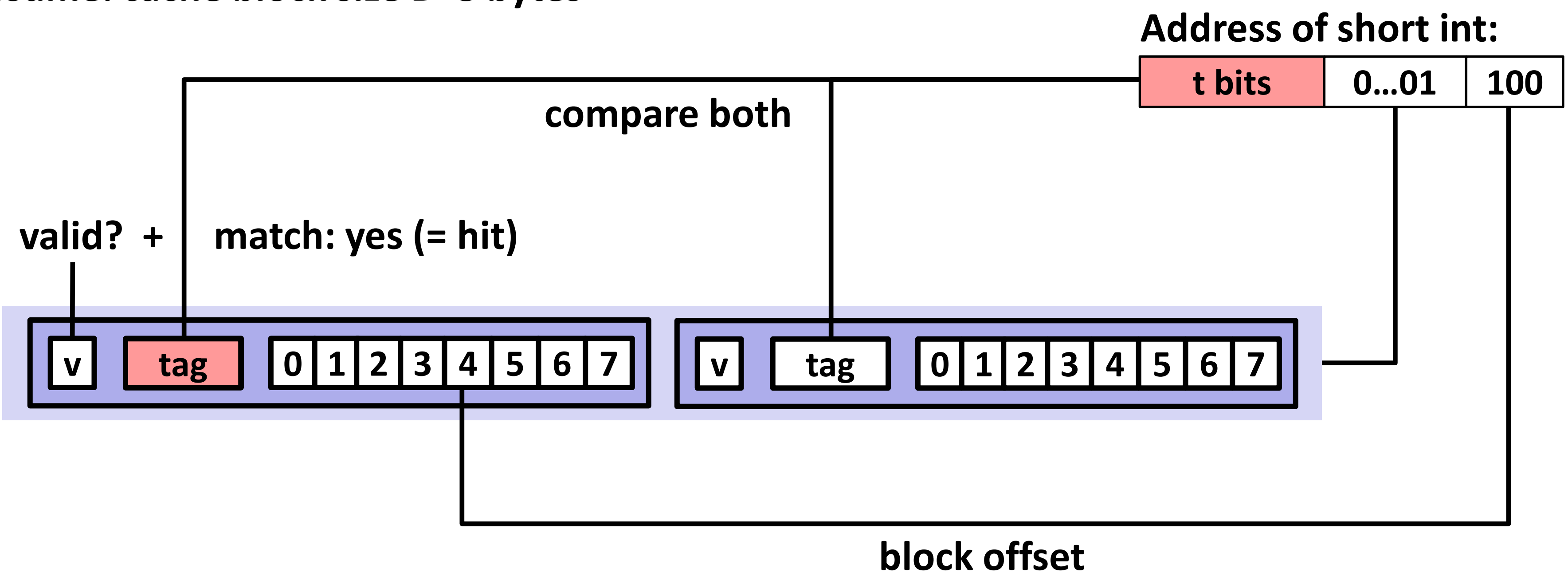
Assume: cache block size B=8 bytes



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

E = 2: Two lines per set

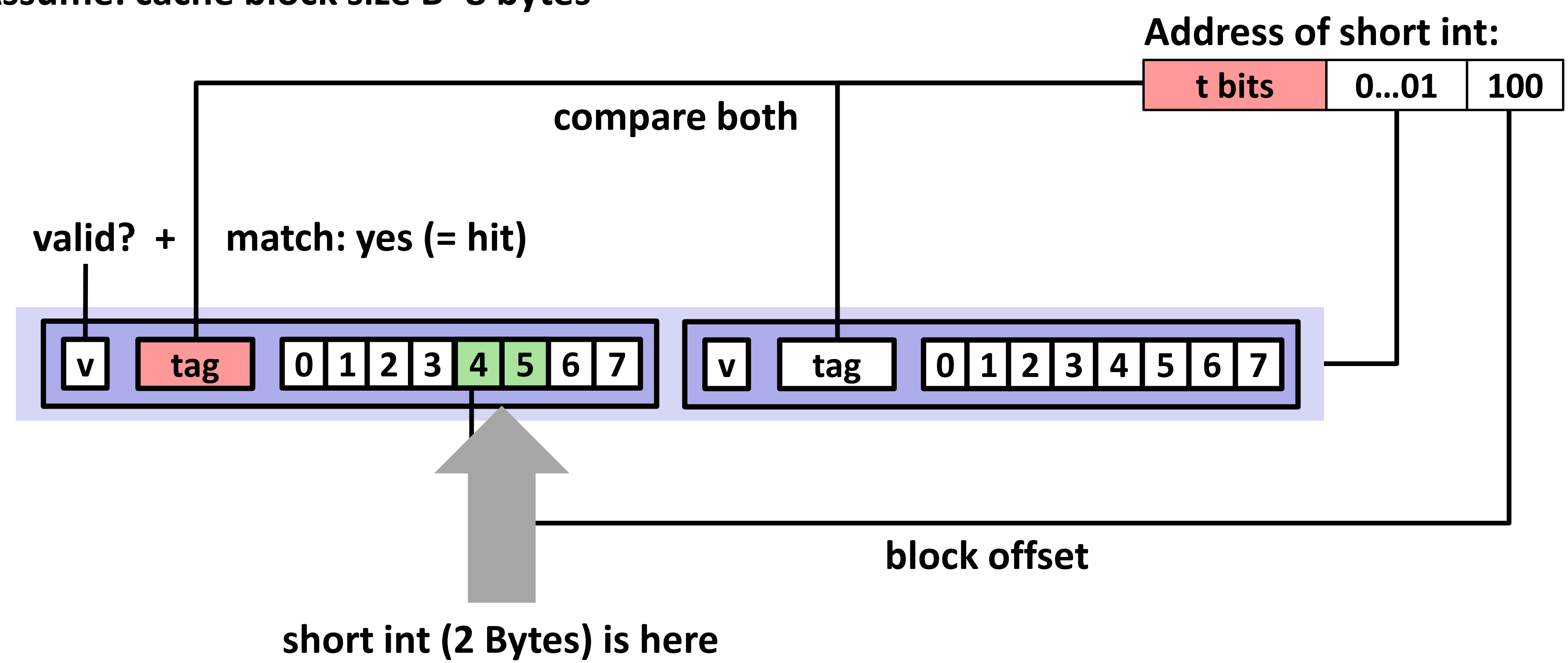
Assume: cache block size B=8 bytes



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

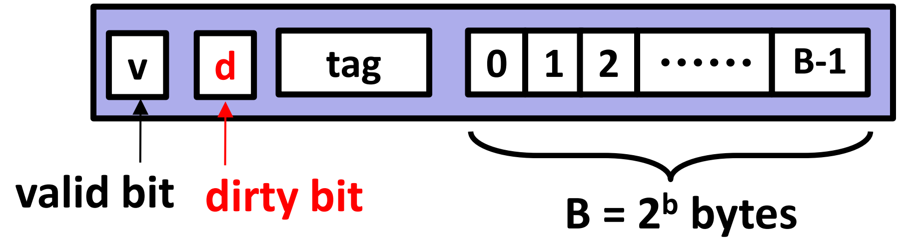
Address trace (reads, one byte per read):

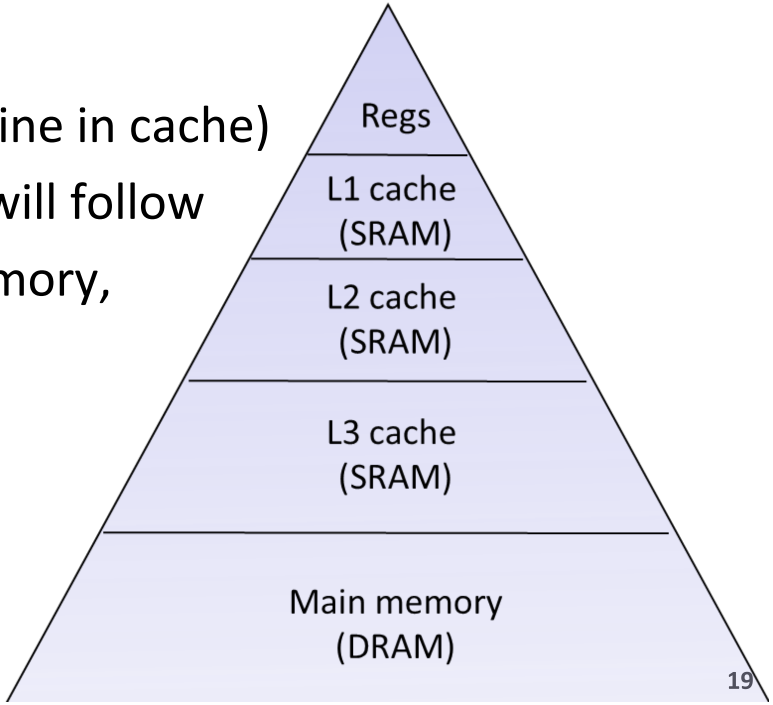
0	**[0000$_2$],**	miss
1	**[0001$_2$],**	hit
7	**[0111$_2$],**	miss
8	**[1000$_2$],**	miss
0	**[0000$_2$]**	hit

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

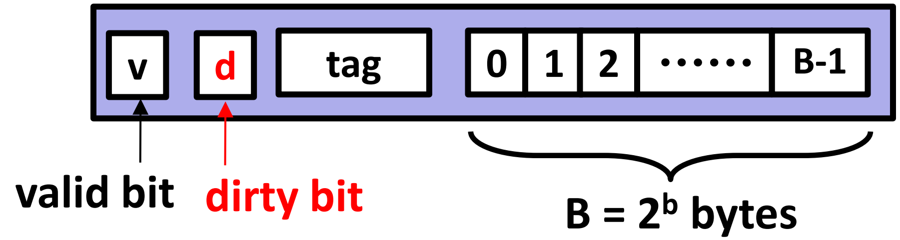
What about writes?

- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory
- **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 has been written to)
- **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**



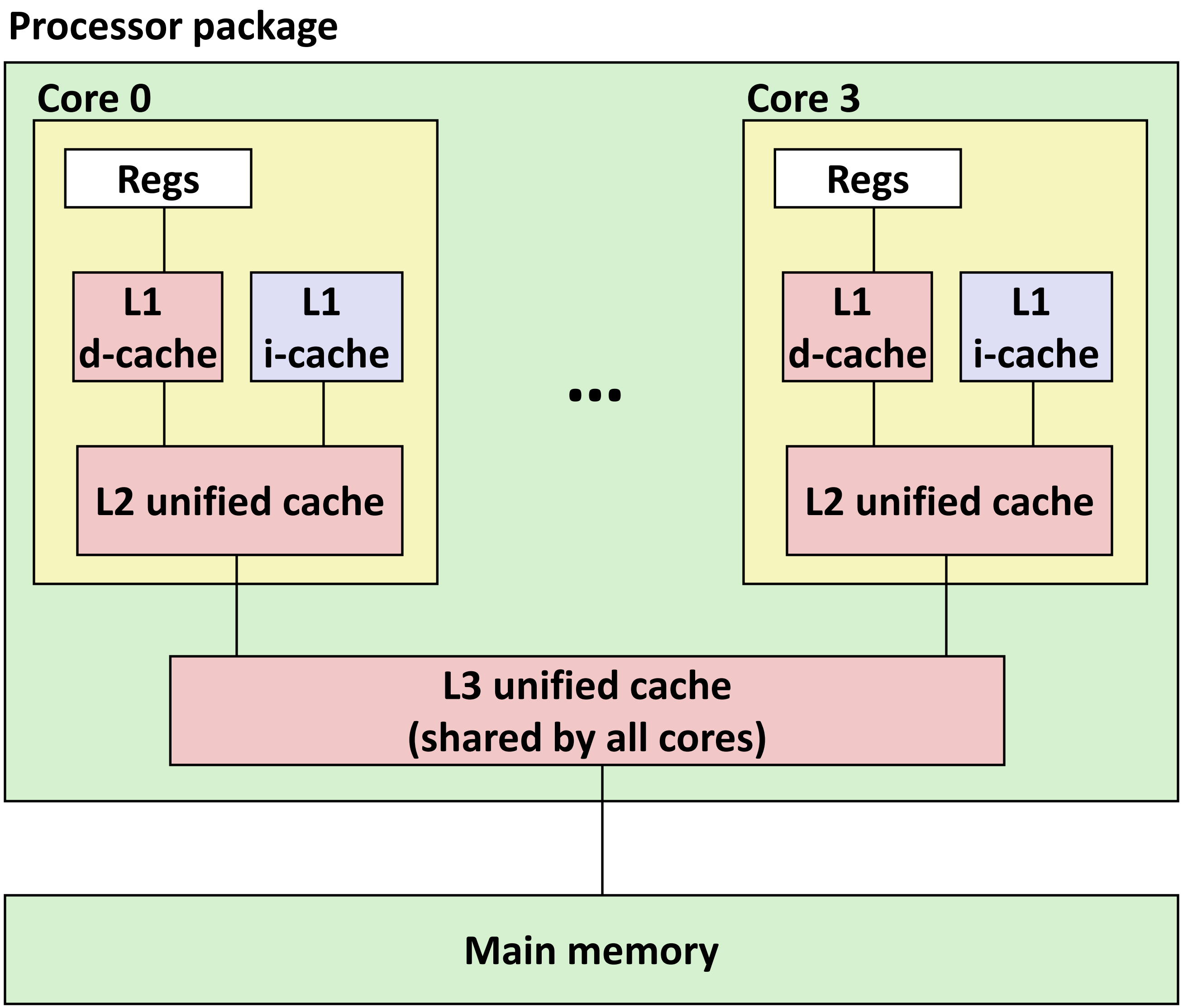


Practical Write-back Write-allocate



- **A write to address X is issued**
- **If it is a hit**
  - Update the contents of block
  - Set dirty bit to 1 (bit is sticky and only cleared on eviction)
- **If it is a miss**
  - Fetch block from memory (per a read miss)
  - The perform the write operations (per a write hit)
- **If a line is evicted and dirty bit is set to 1**
  - The entire block of 2^b bytes are written back to memory
  - Dirty bit is cleared (set to 0)
  - Line is replaced by new contents

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: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

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 50-200 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 this simplified example:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:
    97% hits: 1 cycle + 0.03 x 100 cycles = **4 cycles**
    99% hits: 1 cycle + 0.01 x 100 cycles = **2 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

https://canvas.cmu.edu/courses/49105/quizzes/150049

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`.

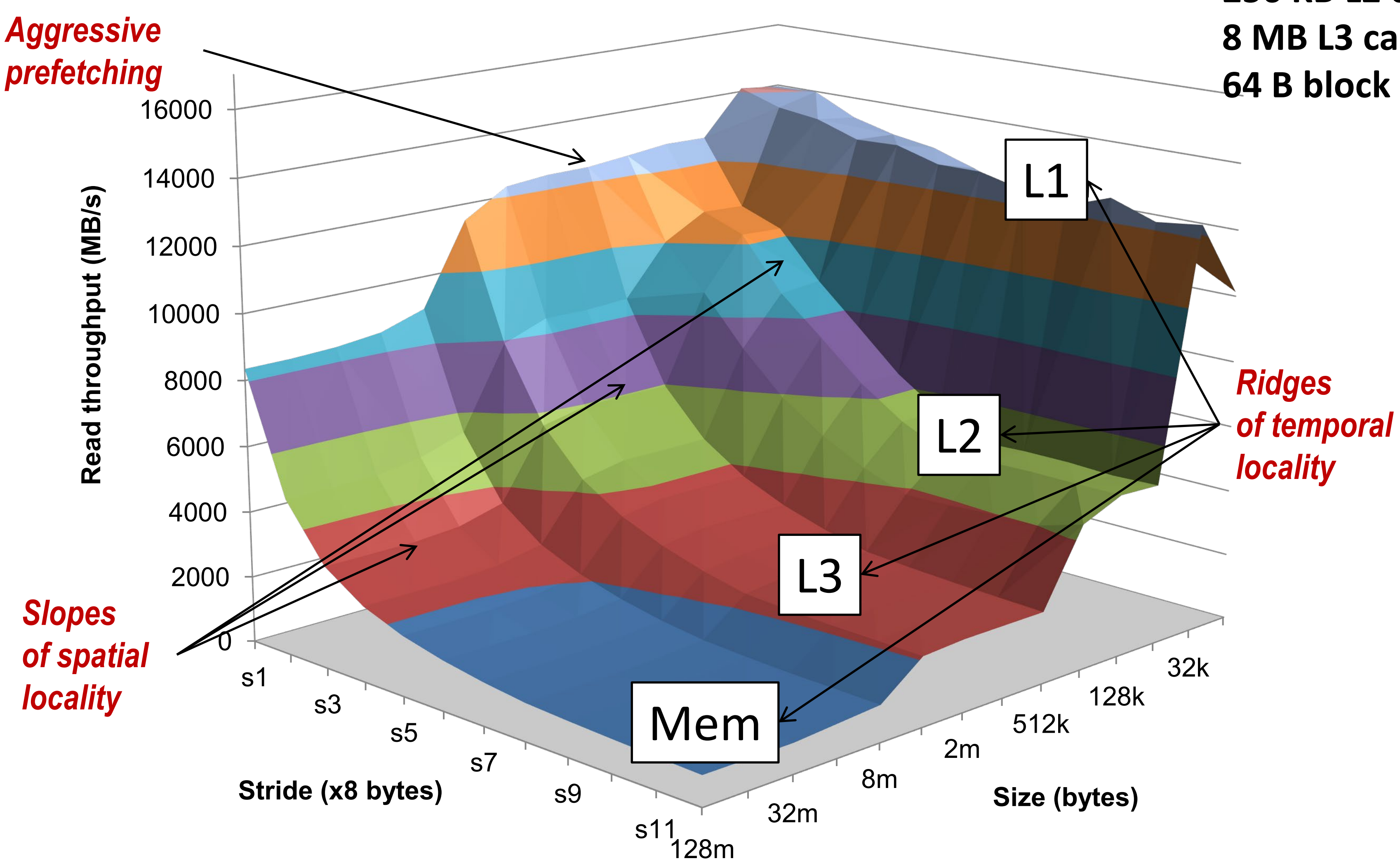
For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

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

The Memory Mountain

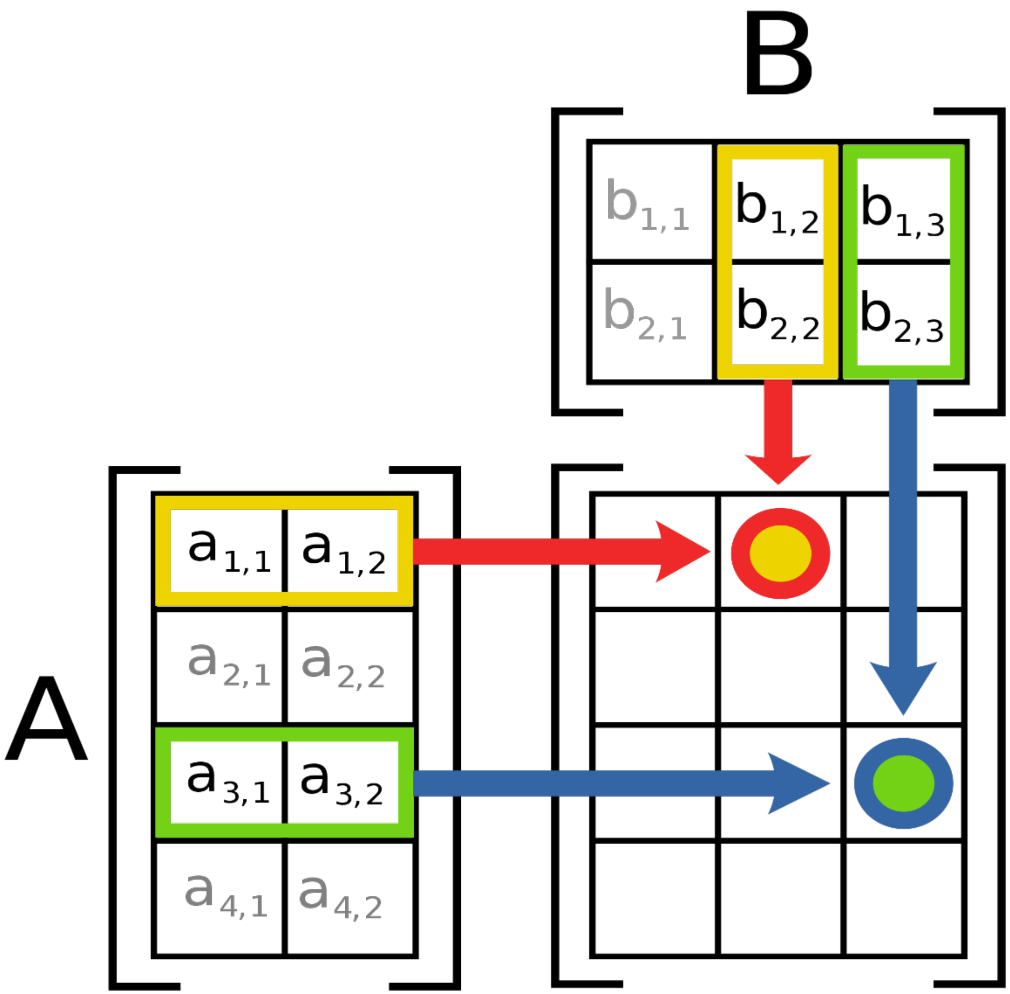
Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Today

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

Remember matrix multiplication



Out[i, j] =
dot product(A[i, ..], B[..,j])
= sum (
a[i, 0] * b[0, j],
a[I, 1] * b[1, j]
)

Matrix Multiplication Example

- **Description:**
  - Multiply *N* x *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

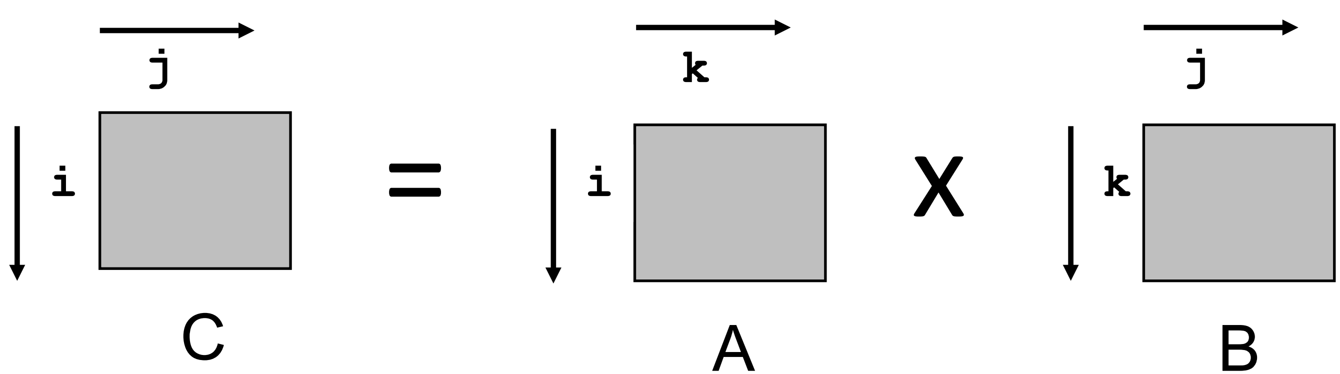
Variable ***sum*** *held in register*

```
/* 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;
  }
}
```

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)

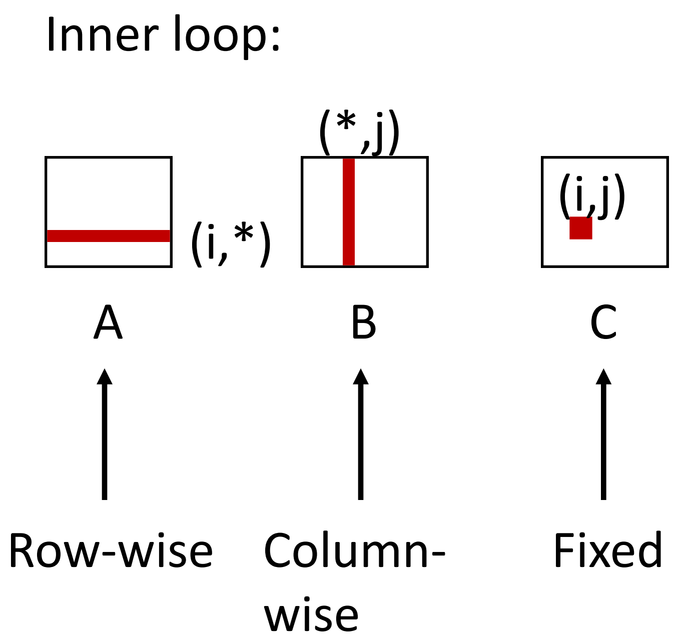
- **C arrays allocated in row-major order**

a [0] [0]	. . .	a [0] [N-1]	a [1] [0]	. . .	a [1] [N-1]	. . .	a [M-1] [0]	. . .	a [M-1] [N-1]

- **Stepping through columns in one row:**
  - ```
    for (i = 0; i < N; i++)
      sum += a[0][i]
    ```
  - 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 < M; i++)
      sum += a[i][0];
    ```
  - accesses distant elements: no spatial locality!
    - 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;
  }
}                              matmult/mm.c
```



Miss rate for inner loop iterations:

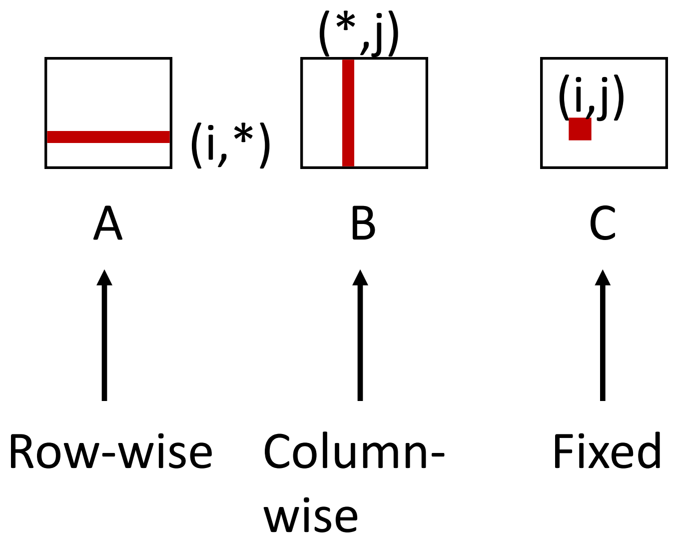
A B C

Block size = 32B (four doubles)

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;
  }
}
```
matmult/mm.c

Inner loop:



<u>Miss rate for inner loop iterations:</u>

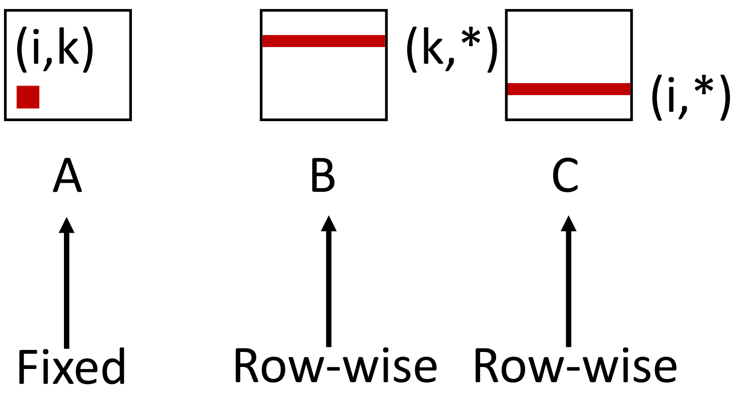
<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

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:



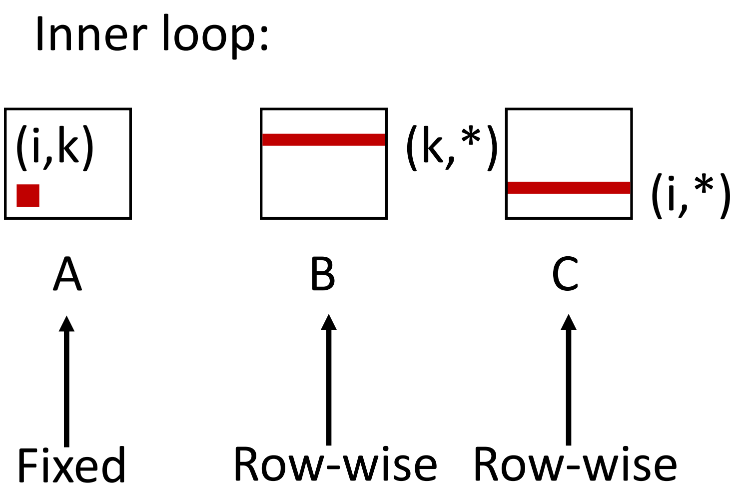
Miss rate for inner loop iterations:

A B C

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
```



Miss rate for inner loop iterations:

A	B	C
0.0	0.25	0.25

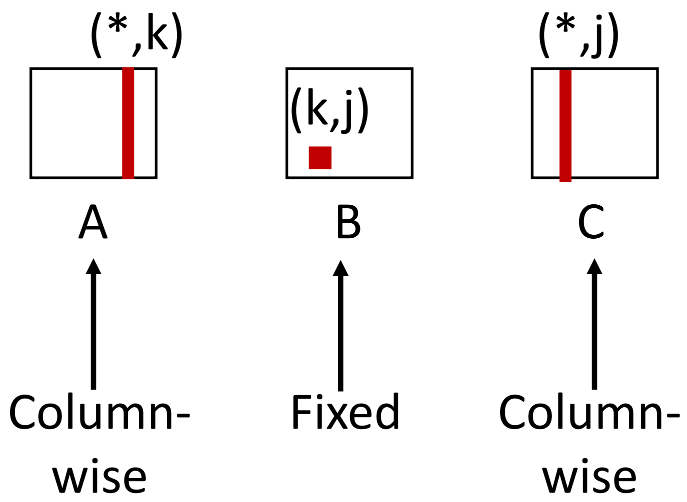
Block size = 32B (four doubles)

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:

A B C

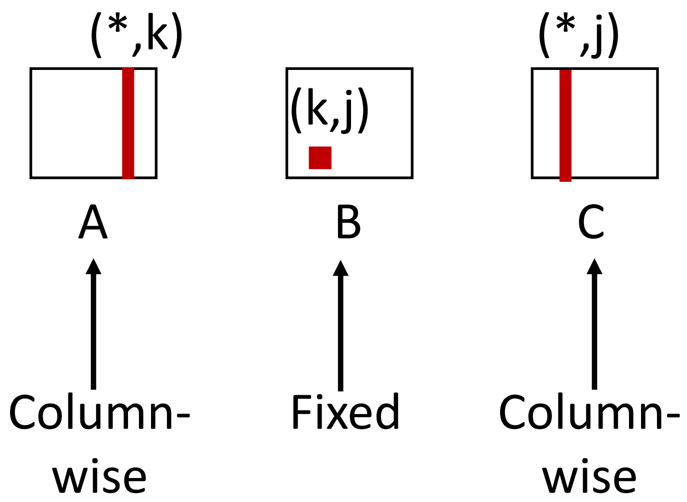
Block size = 32B (four doubles)

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:

A	B	C
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**

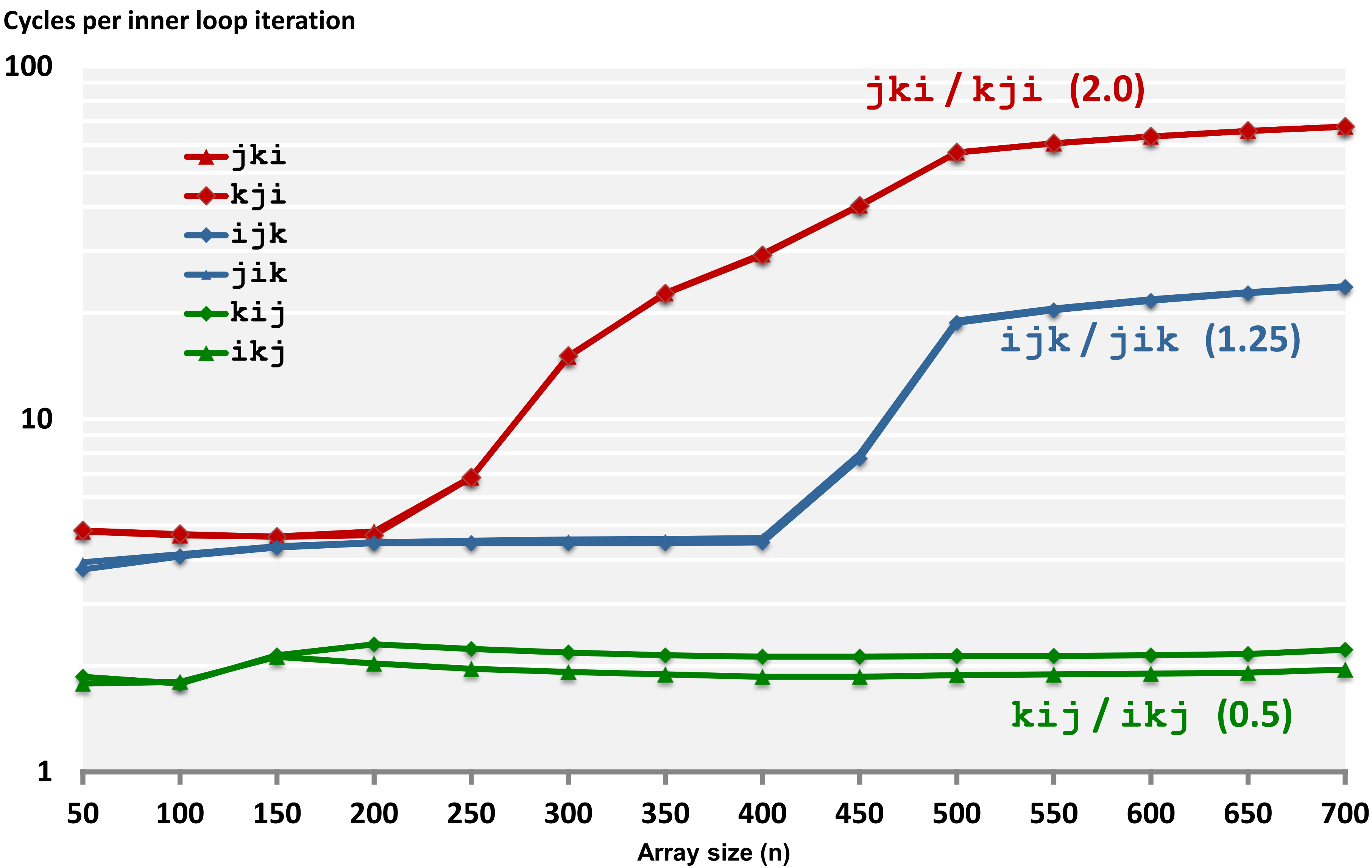
```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

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

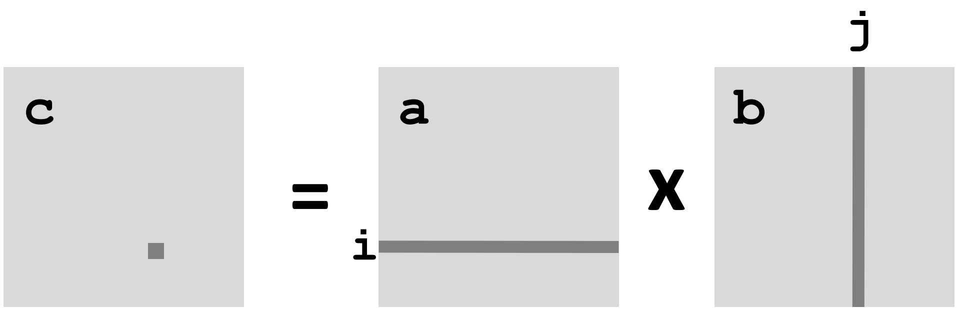


Today

- 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
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];
}
```

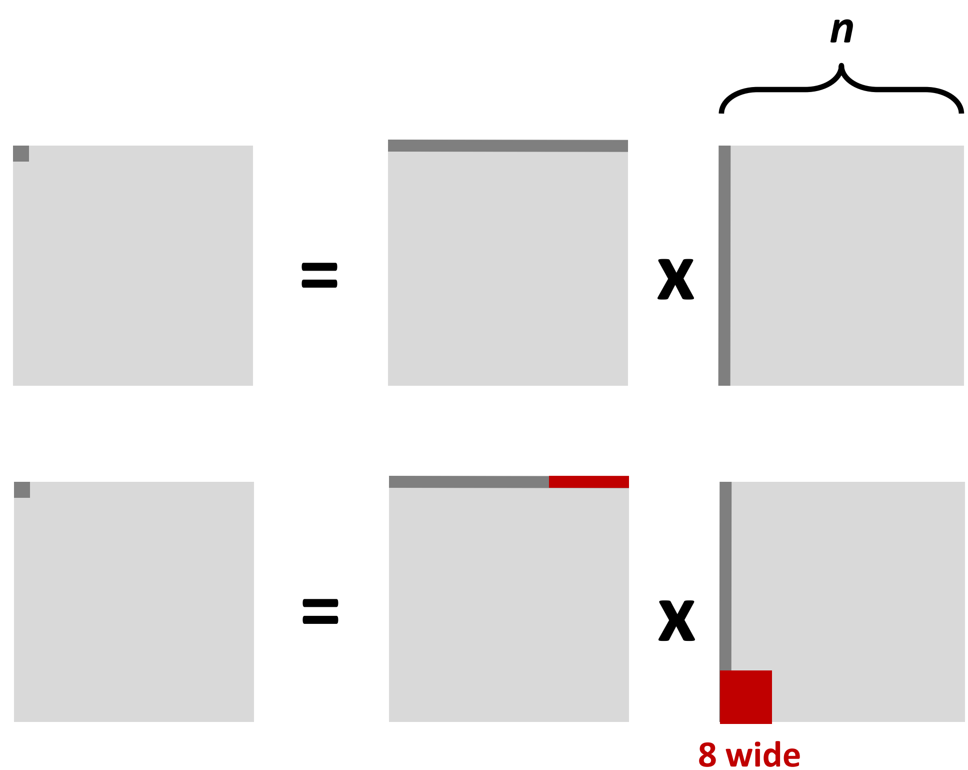


Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than *n*)

- **First iteration:**
  - *n*/8 + n = 9*n*/8 misses

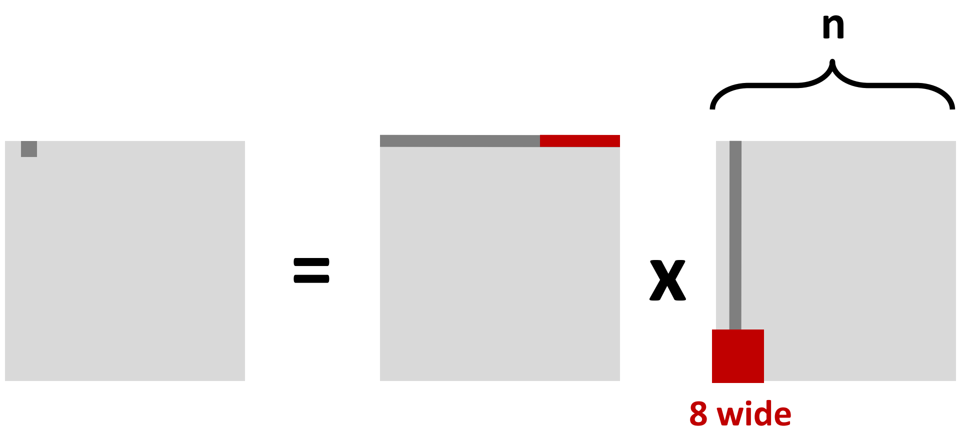
  - Afterwards in cache:
    (schematic)



Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

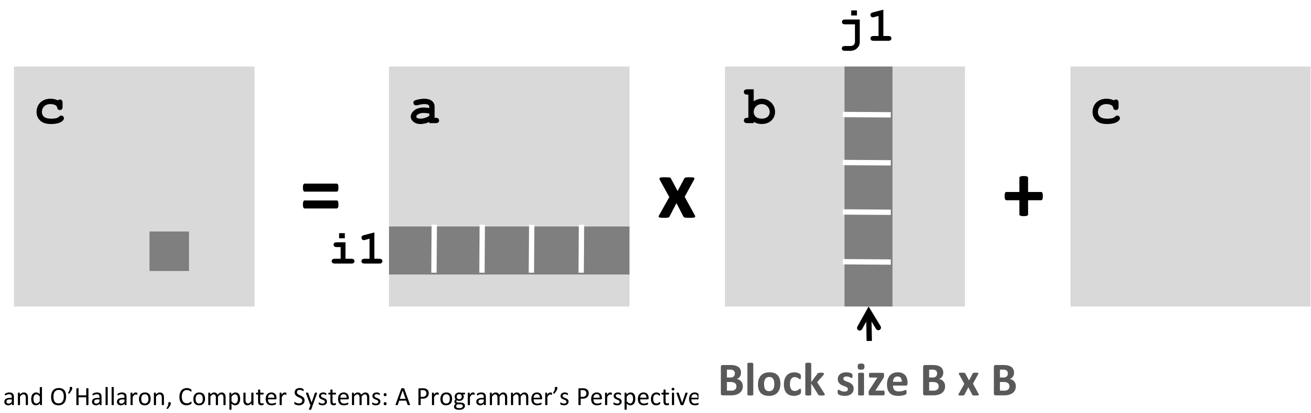
- **Second iteration:**
  - Again:
    $n/8 + n = 9n/8$ misses



- **Total misses:**
  - $9n/8 \; n^2 = (9/8) \; n^3$

Blocked Matrix Multiplication

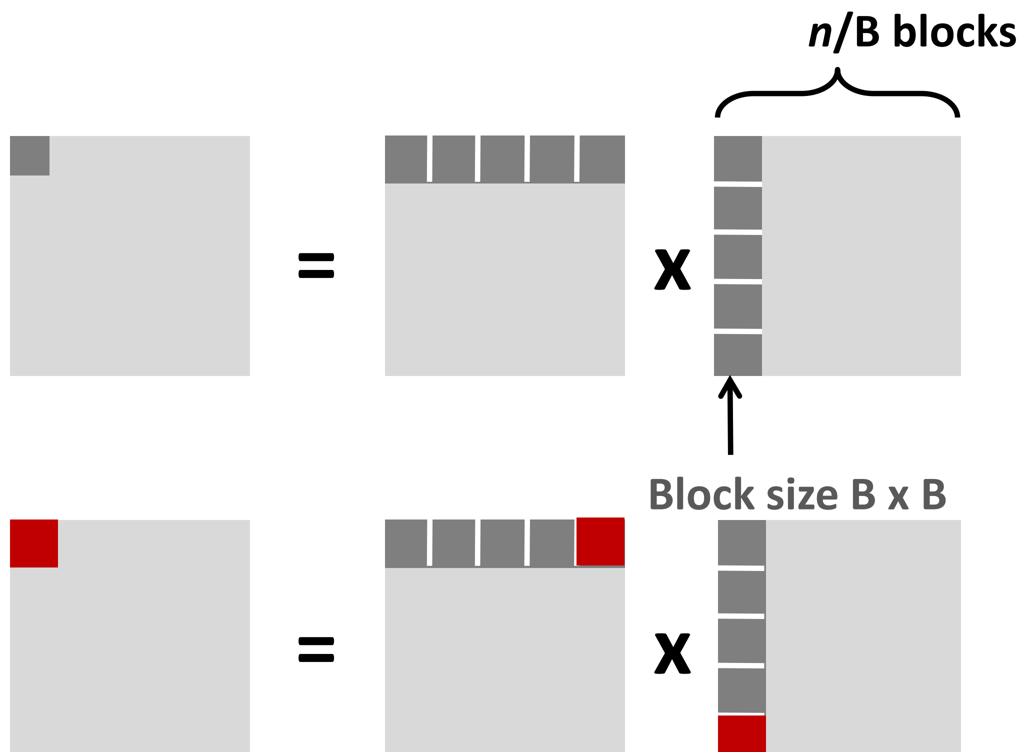
```
c = (double *) calloc(sizeof(double), n*n);

/* 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+=B)
        for (j = 0; j < n; j+=B)
             for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                  for (i1 = i; i1 < i+B; i1++)
                      for (j1 = j; j1 < j+B; j1++)
                          for (k1 = k; k1 < k+B; k1++)
                              c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
                                                            matmult/bmm.c
```



Notation Note
This "B" is not the cache block size B

Cache Miss Analysis

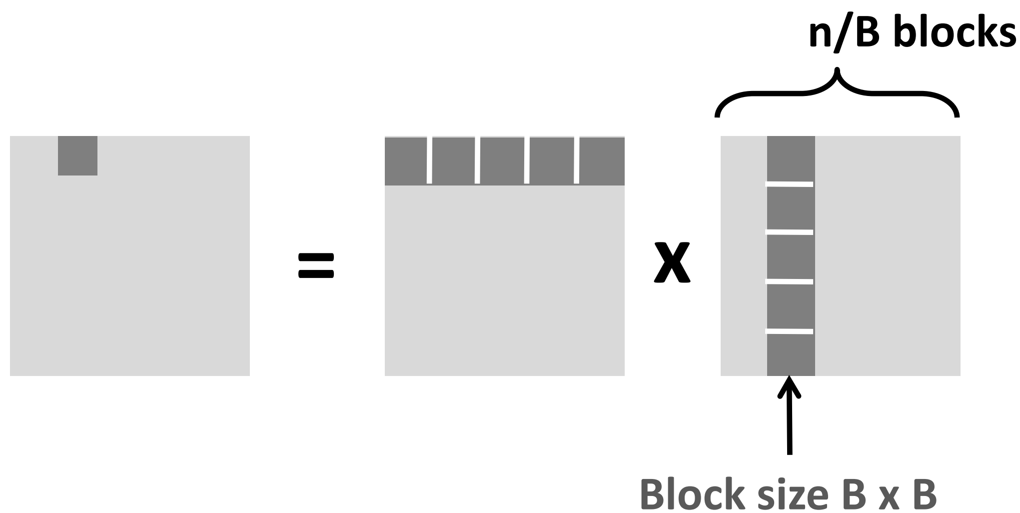
- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$
- **First (block) iteration:**
  - B*B/8 misses for each block
  - $2n/B \times B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)



Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- **Second (block) iteration:**
  - Same as first iteration
  - $2n/B \times B^2/8 = nB/4$



- **Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

Blocking Summary

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

- **Use largest block size B, such that B satisfies $3B^2 < C$**
  - 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

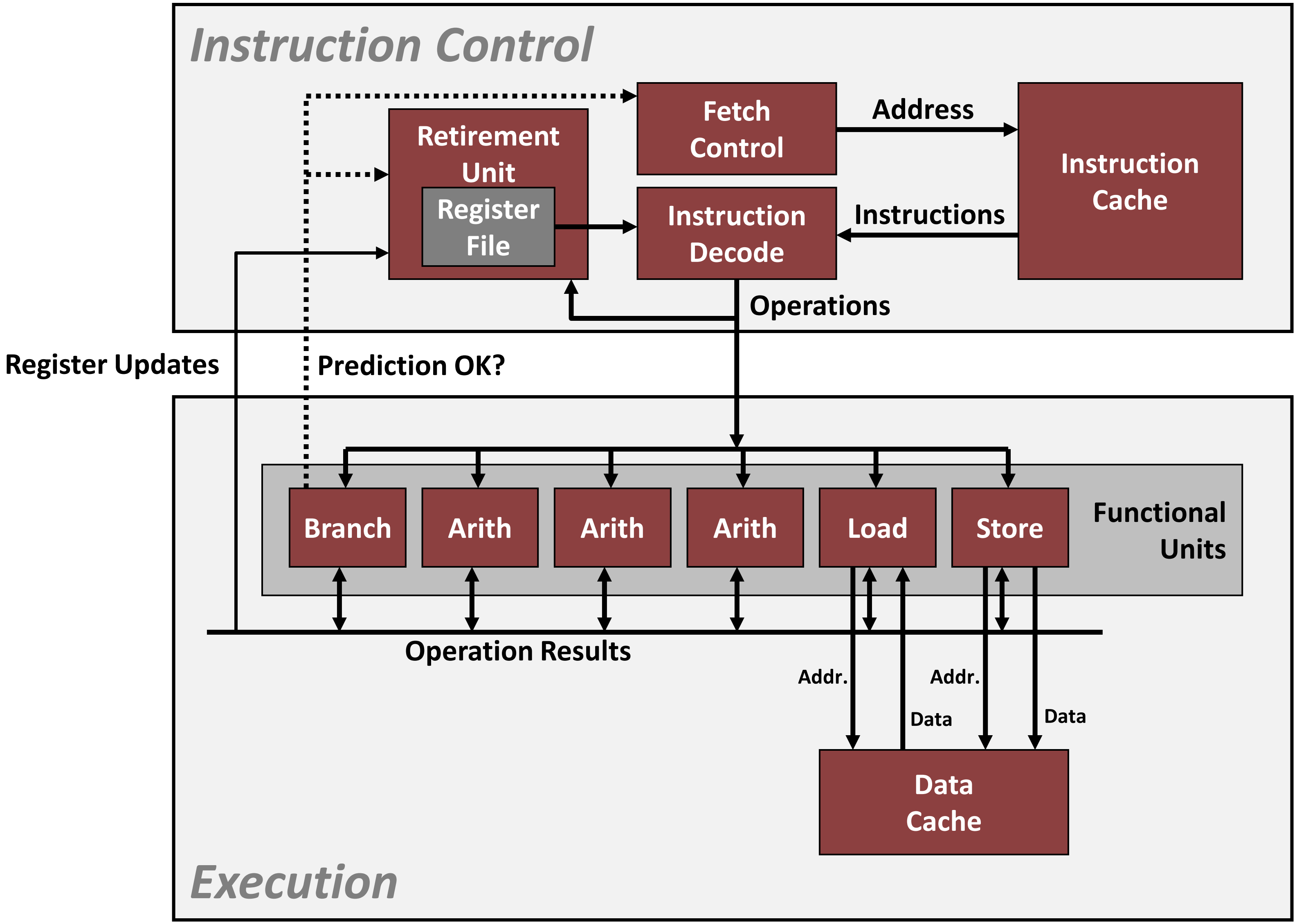
Cache Summary

- **Cache memories can have significant performance impact**

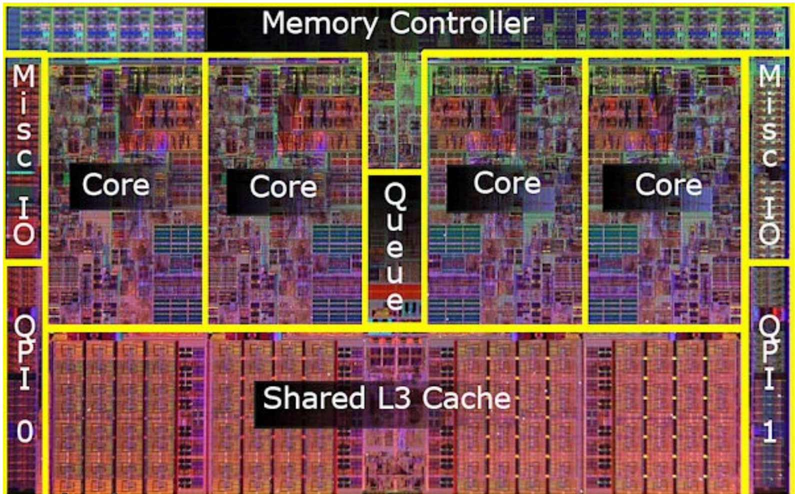
- **You can write your programs to exploit this!**
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible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

Supplemental slides

Recall: Modern CPU Design

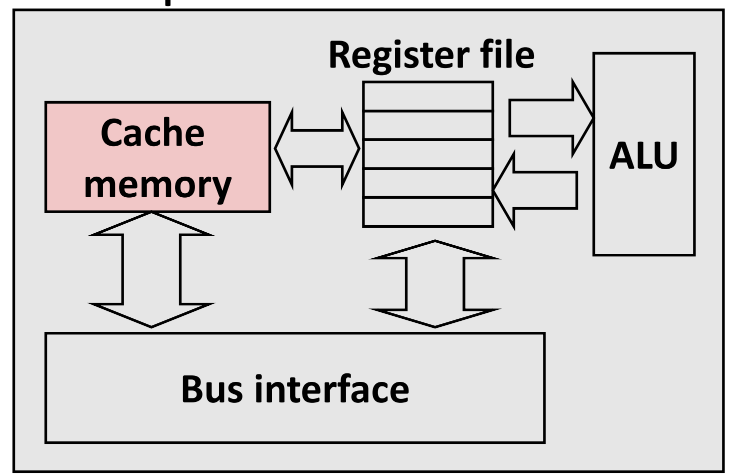


What it Really Looks Like

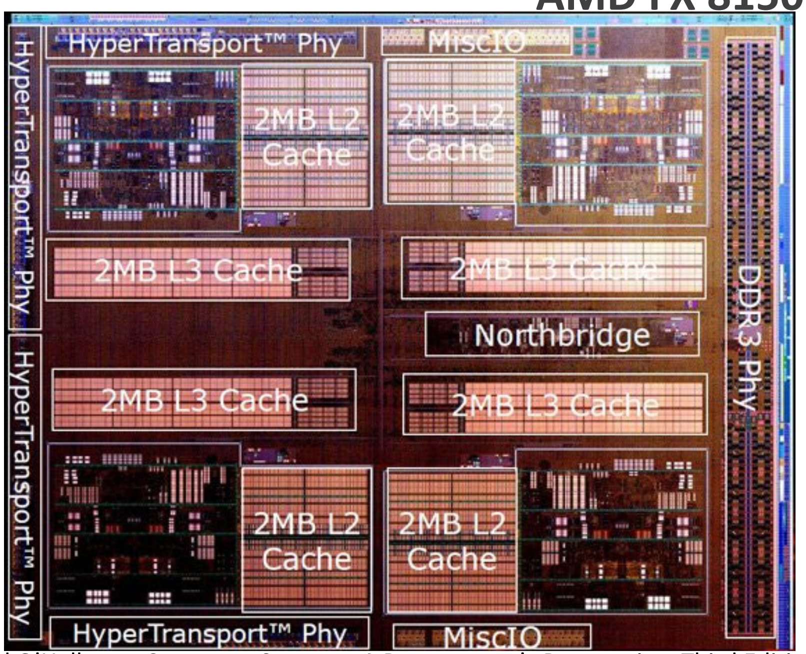


Nehalem

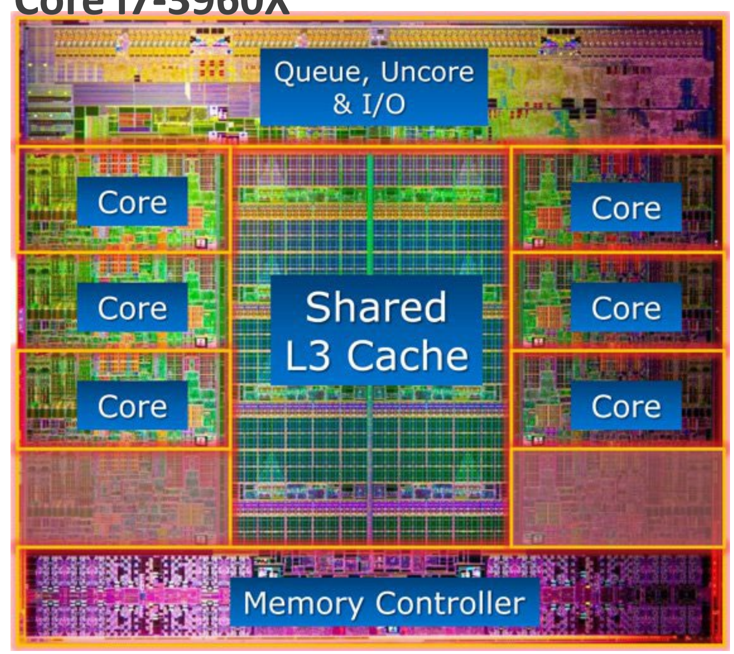
CPU chip



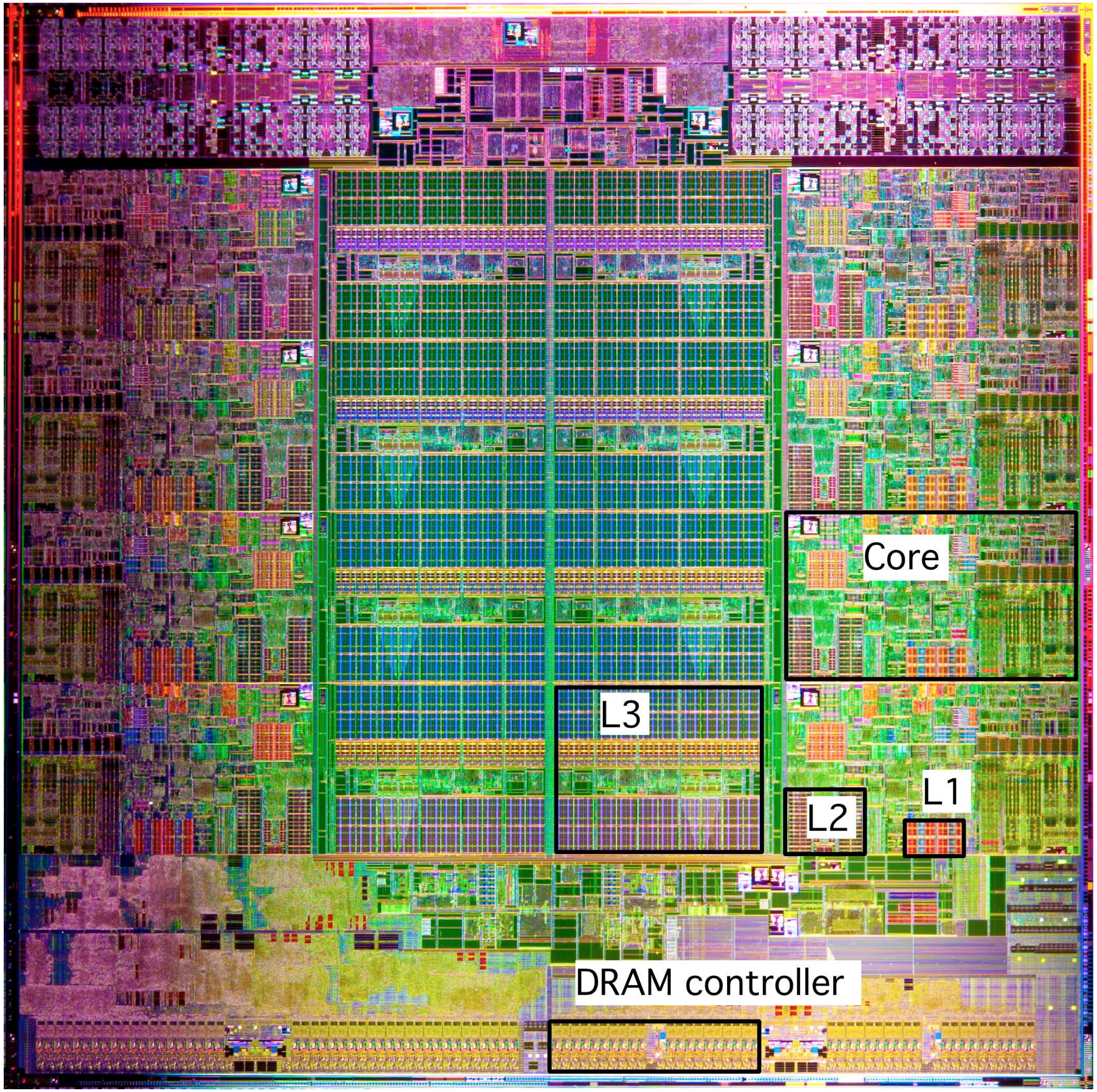
AMD FX 8150



Core i7-3960X



What it Really Looks Like (Cont.)



Intel Sandy Bridge
Processor Die

L1: 32KB Instruction + 32KB Data
L2: 256KB
L3: 3–20MB

Why Index Using Middle Bits?

Direct mapped: One line per set

Assume: cache block size 8 bytes

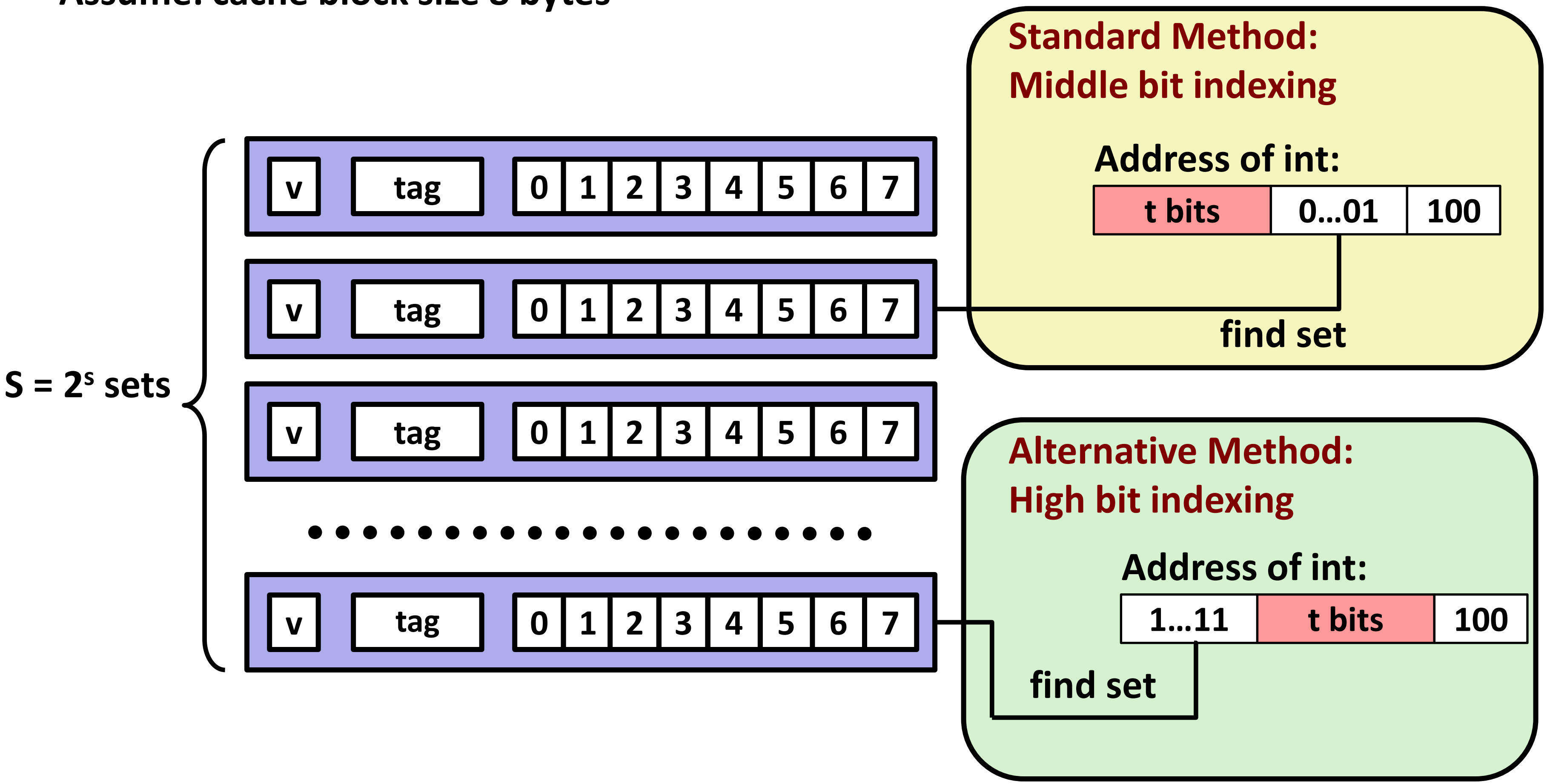
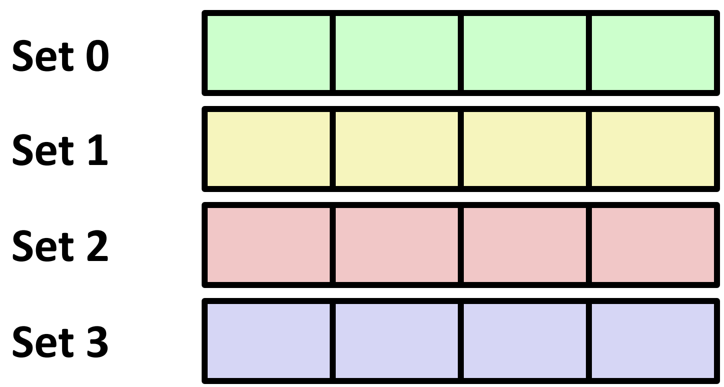
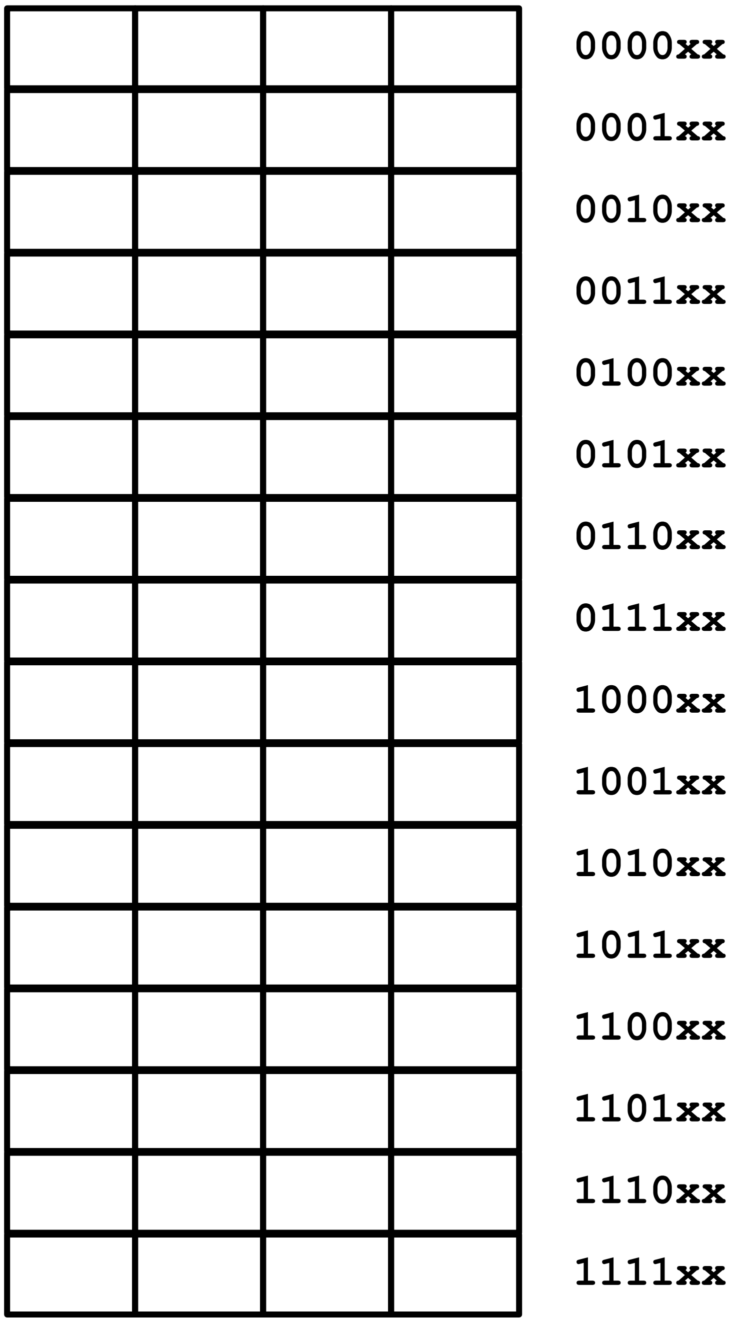


Illustration of Indexing Approaches

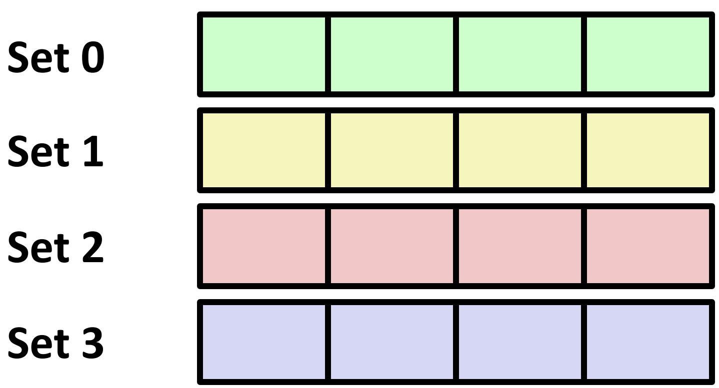
- **64-byte memory**
  - 6-bit addresses
- **16 byte, direct-mapped cache**
- **Block size = 4. (Thus, 4 sets; why?)**
- **2 bits tag, 2 bits index, 2 bits offset**

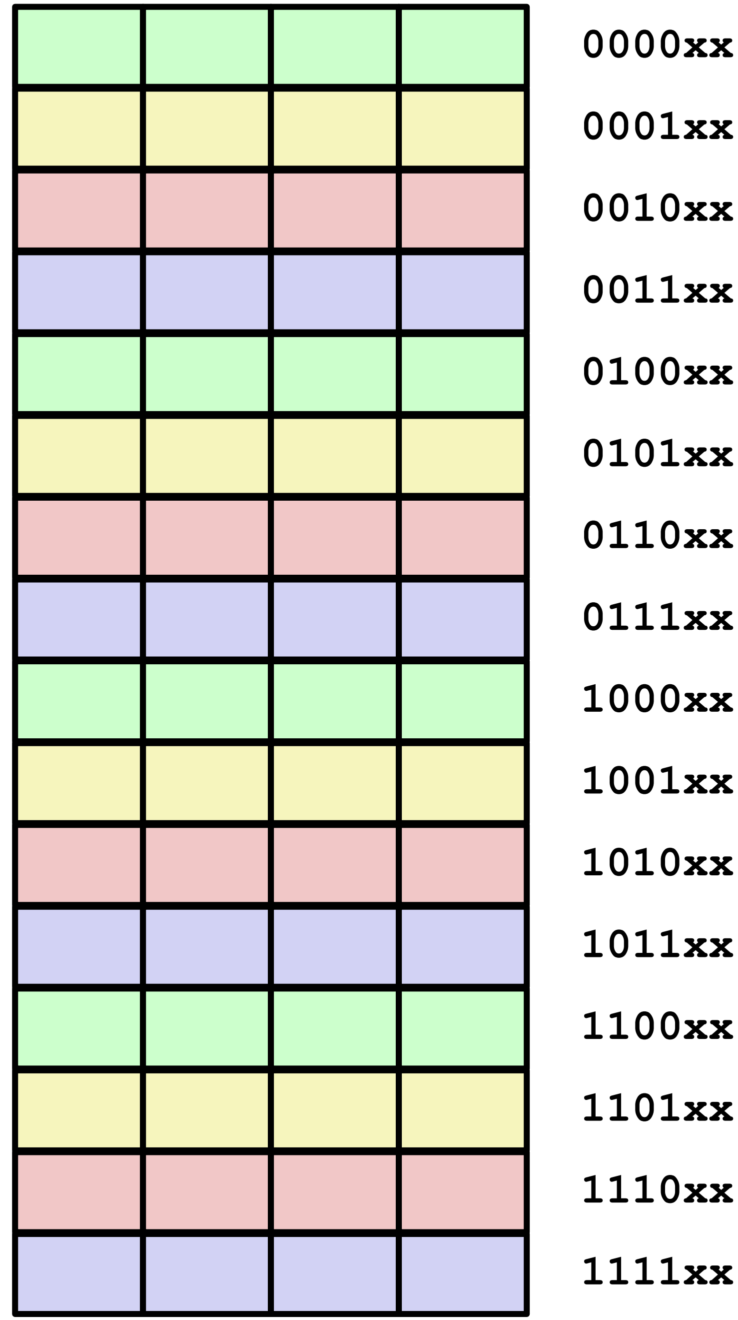




Middle Bit Indexing

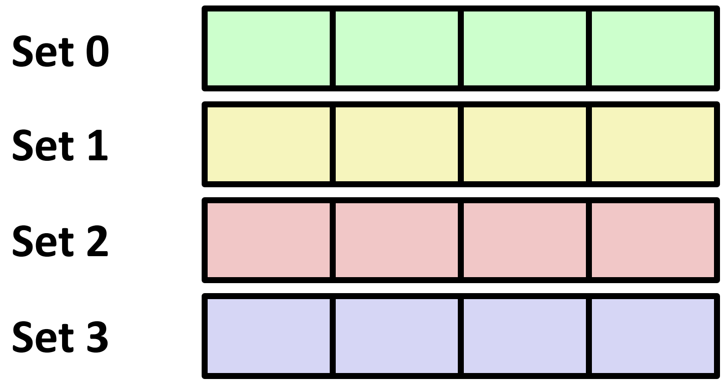
- Addresses of form **TTSSBB**
  - **TT** Tag bits
  - **SS** Set index bits
  - **BB** Offset bits
- Makes good use of spatial locality

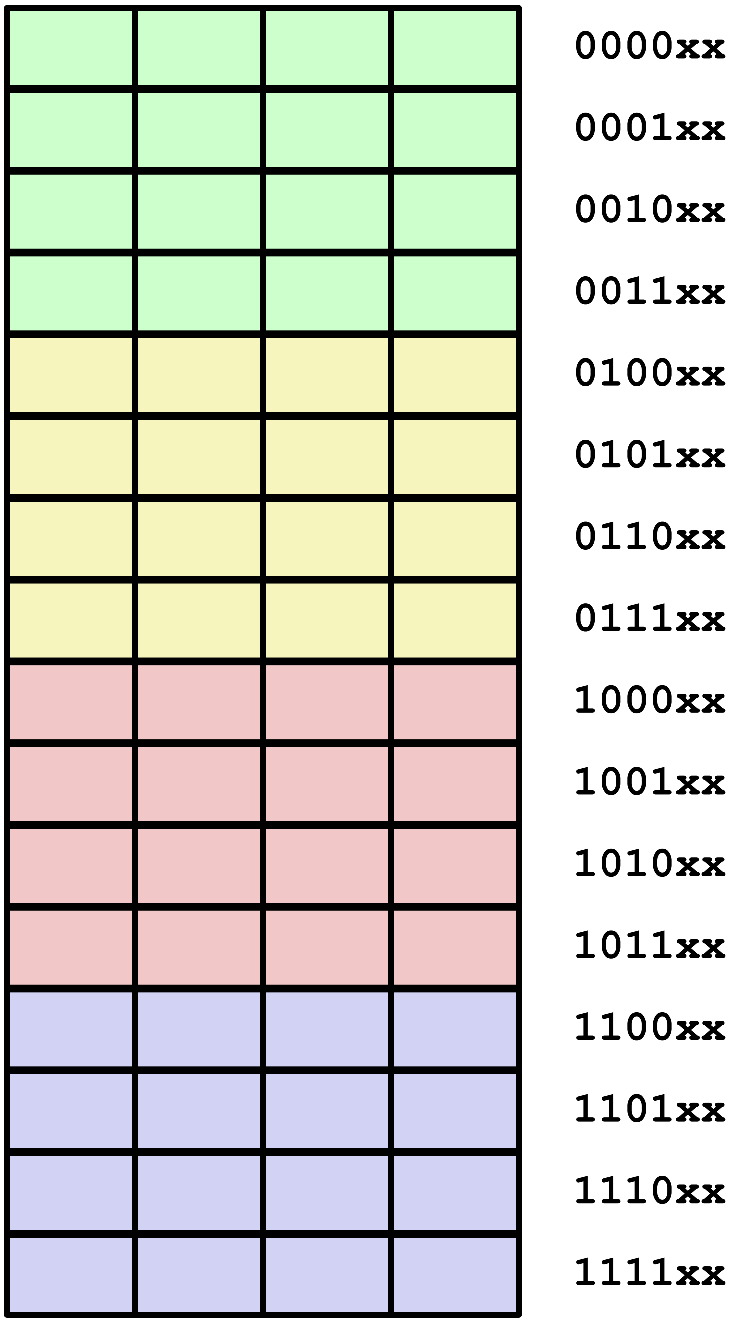




High Bit Indexing

- Addresses of form SSTTBB
  - SS Set index bits
  - TT Tag bits
  - BB Offset bits
- Program with high spatial locality would generate lots of conflicts

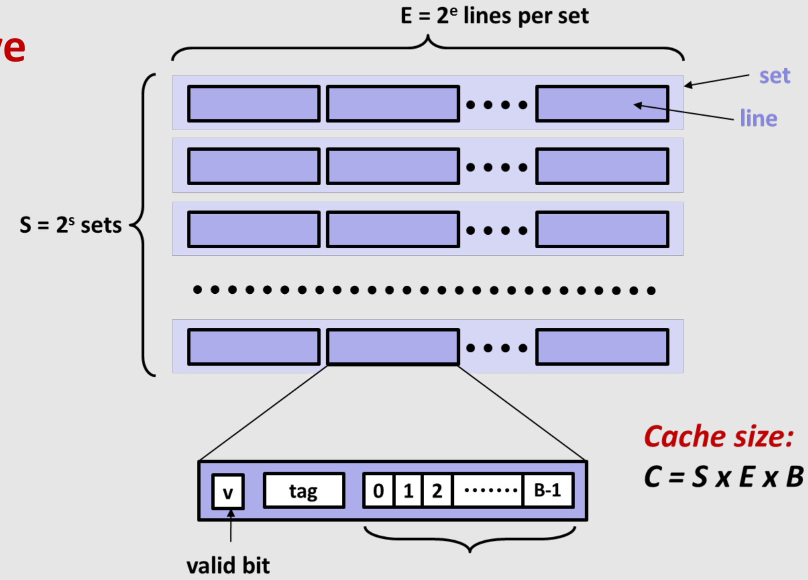




Example: Core i7 L1 Data Cache

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

B =
S = , s =
E = , e =
C =



Cache size:
C = S x E x B data bytes

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:

t bits	s bits	b bits
tag	set index	block offset

Block offset: . bits
Set index: . bits
Tag: . bits

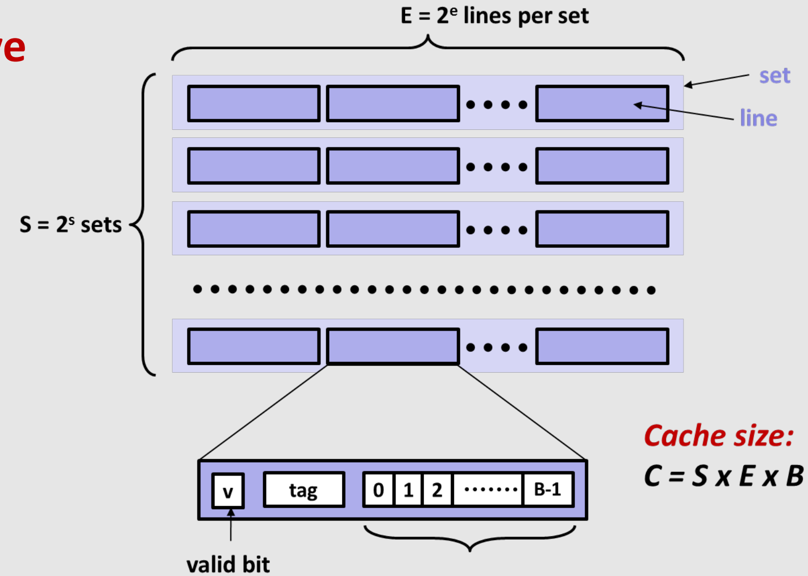
Stack Address:
`0x00007f7262a1e010`

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

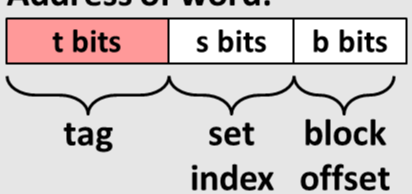
B = 64
S = 64, s = 6
E = 8, e = 3
C = 64 x 64 x 8 = 32,768



Cache size:
C = S x E x B data bytes

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

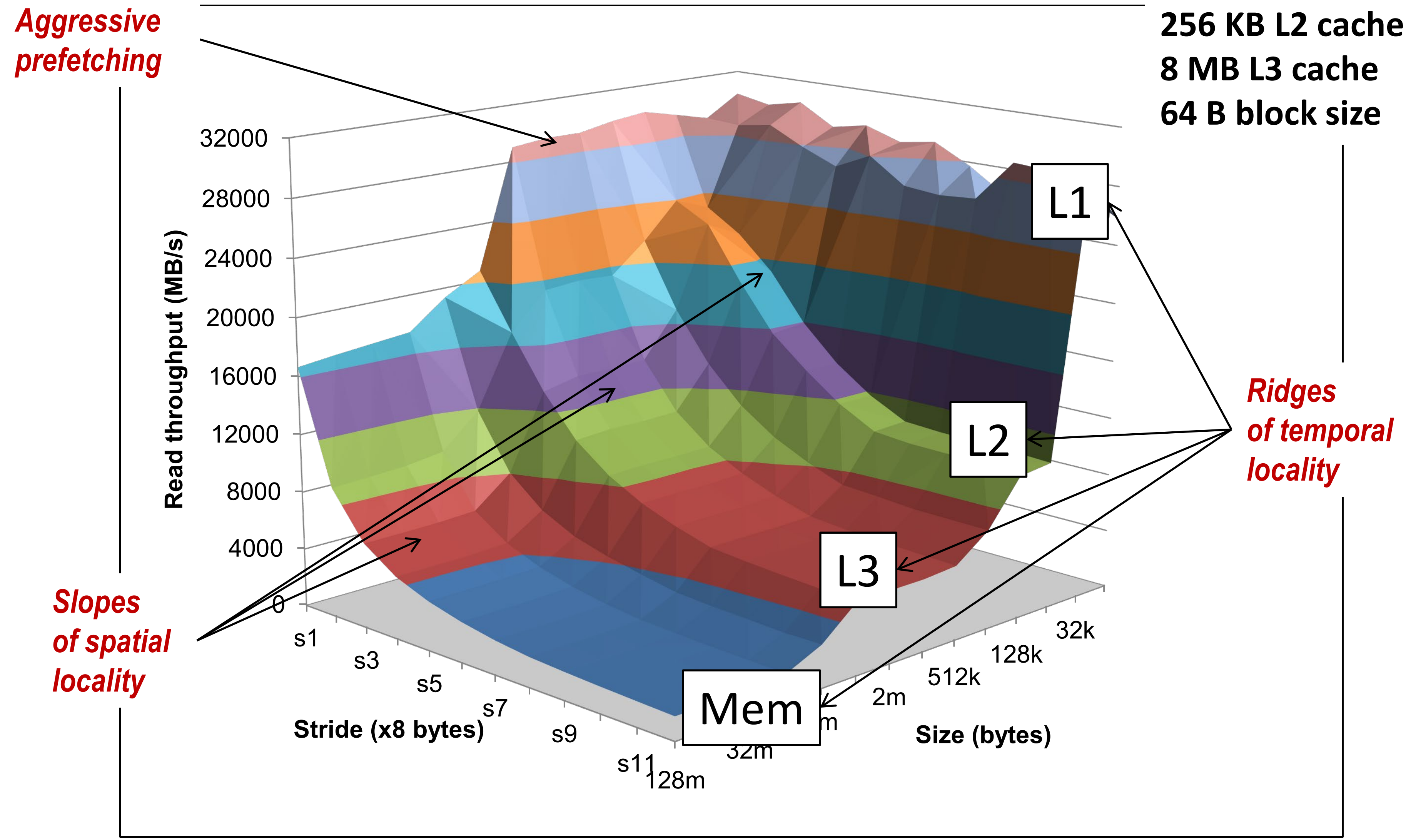
Stack Address:
`0x00007f7262a1e010`

`0000 0001 0000`

Block offset: `0x10`
Set index: `0x0`
Tag: `0x7f7262a1e`

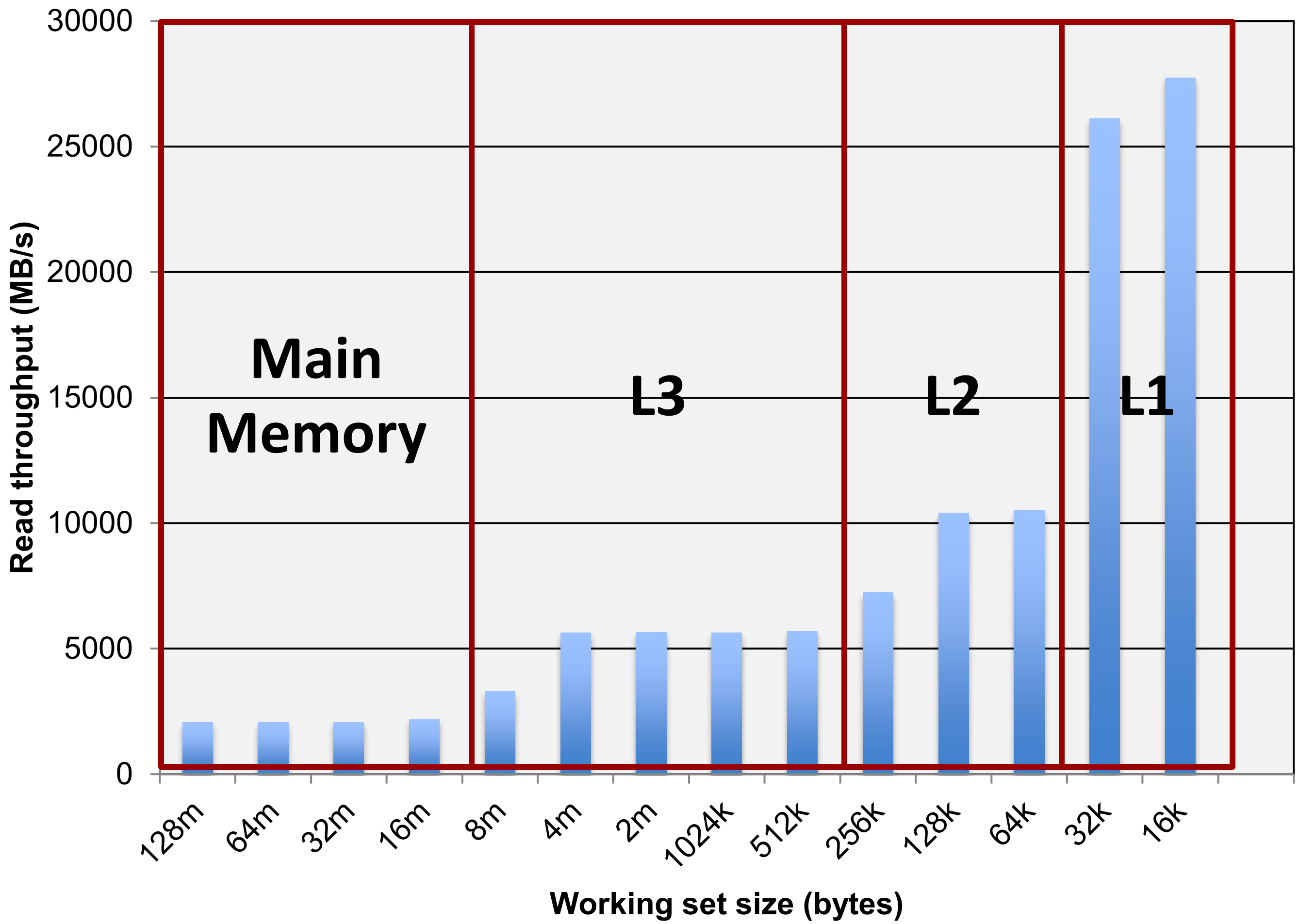
The Memory Mountain

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



Cache Capacity Effects from Memory Mountain

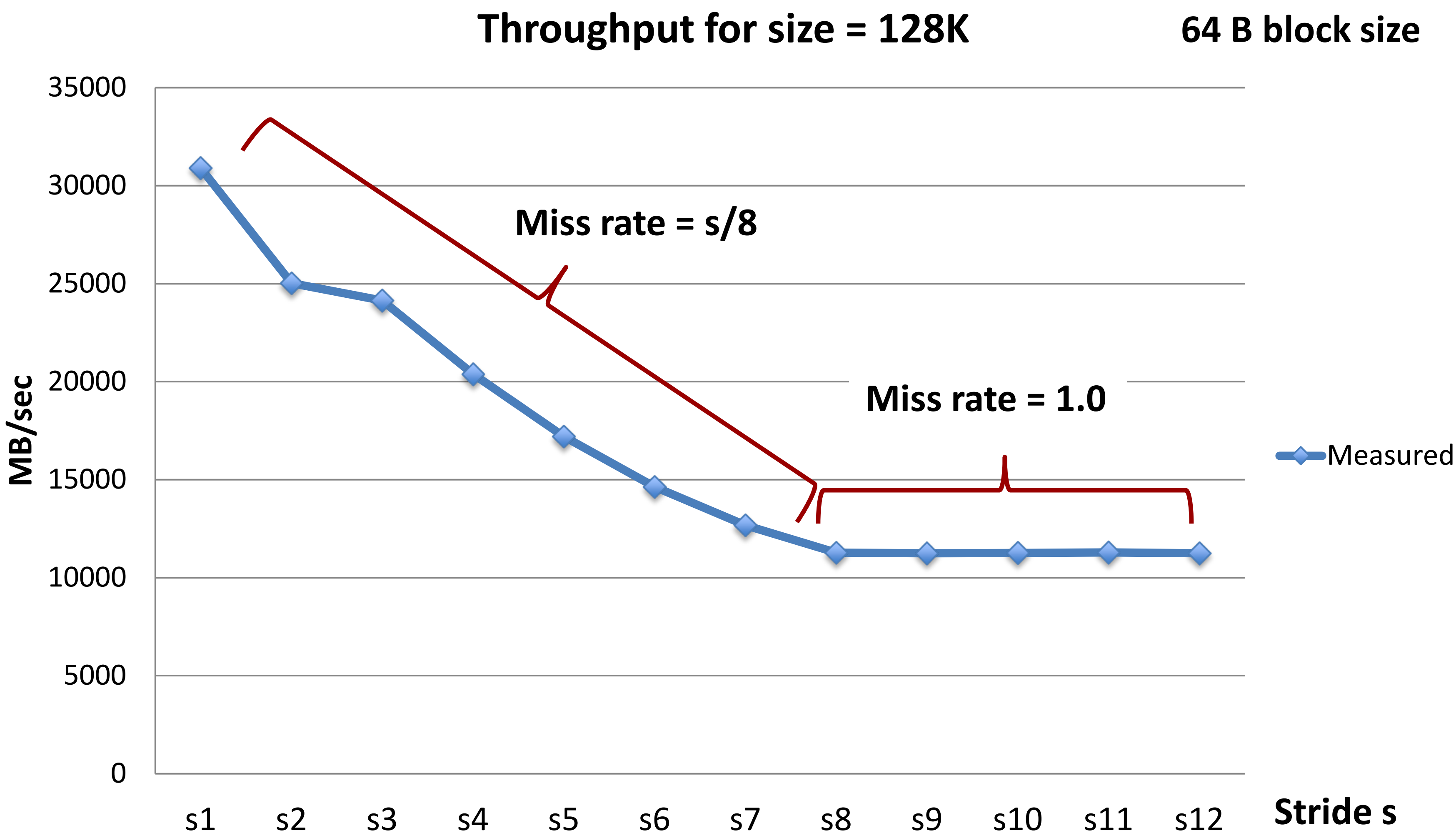
Core i7 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Slice through memory mountain with stride=8

Cache Block Size Effects from Memory Mountain

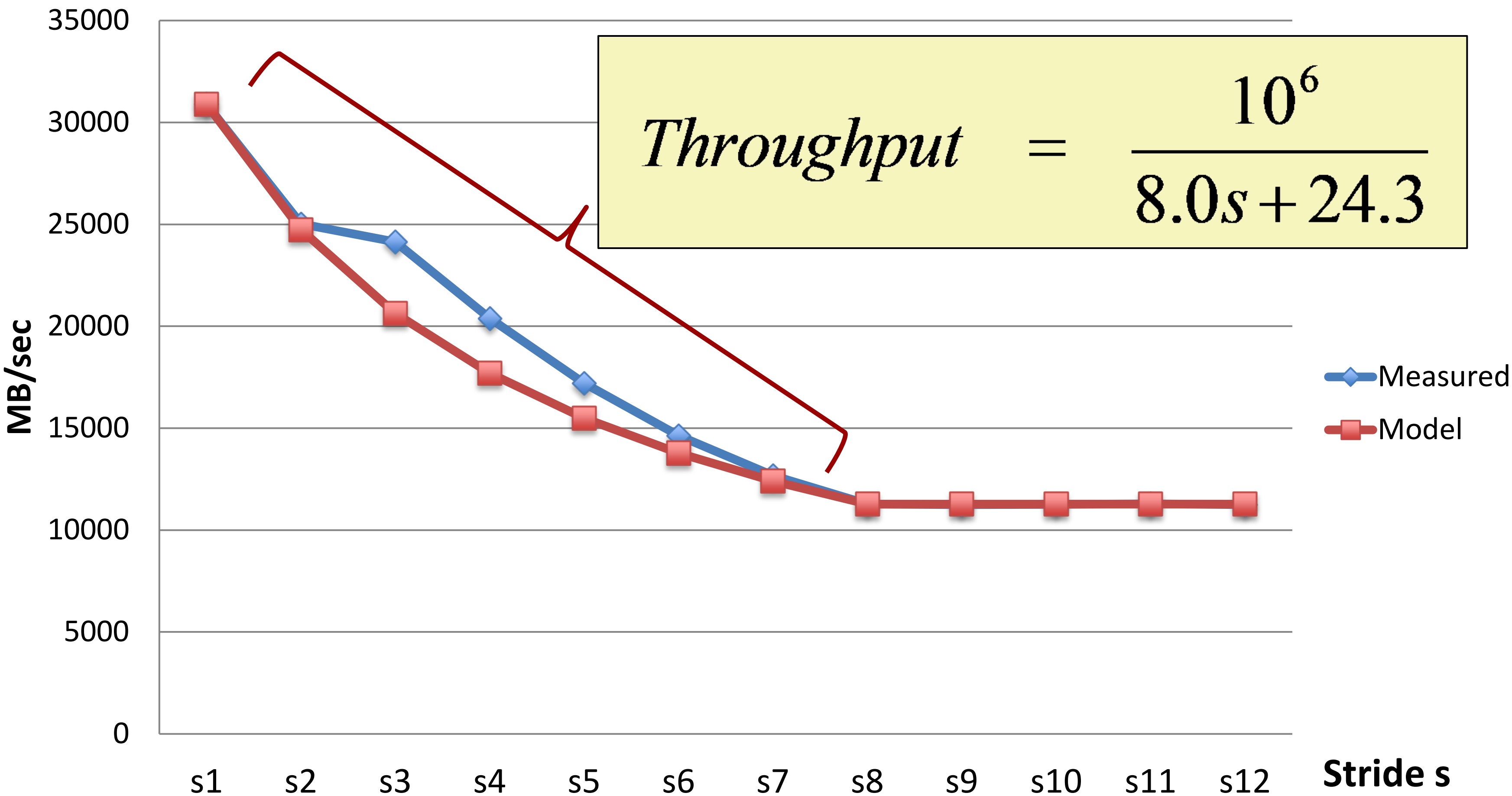
Core i7 Haswell
2.26 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



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

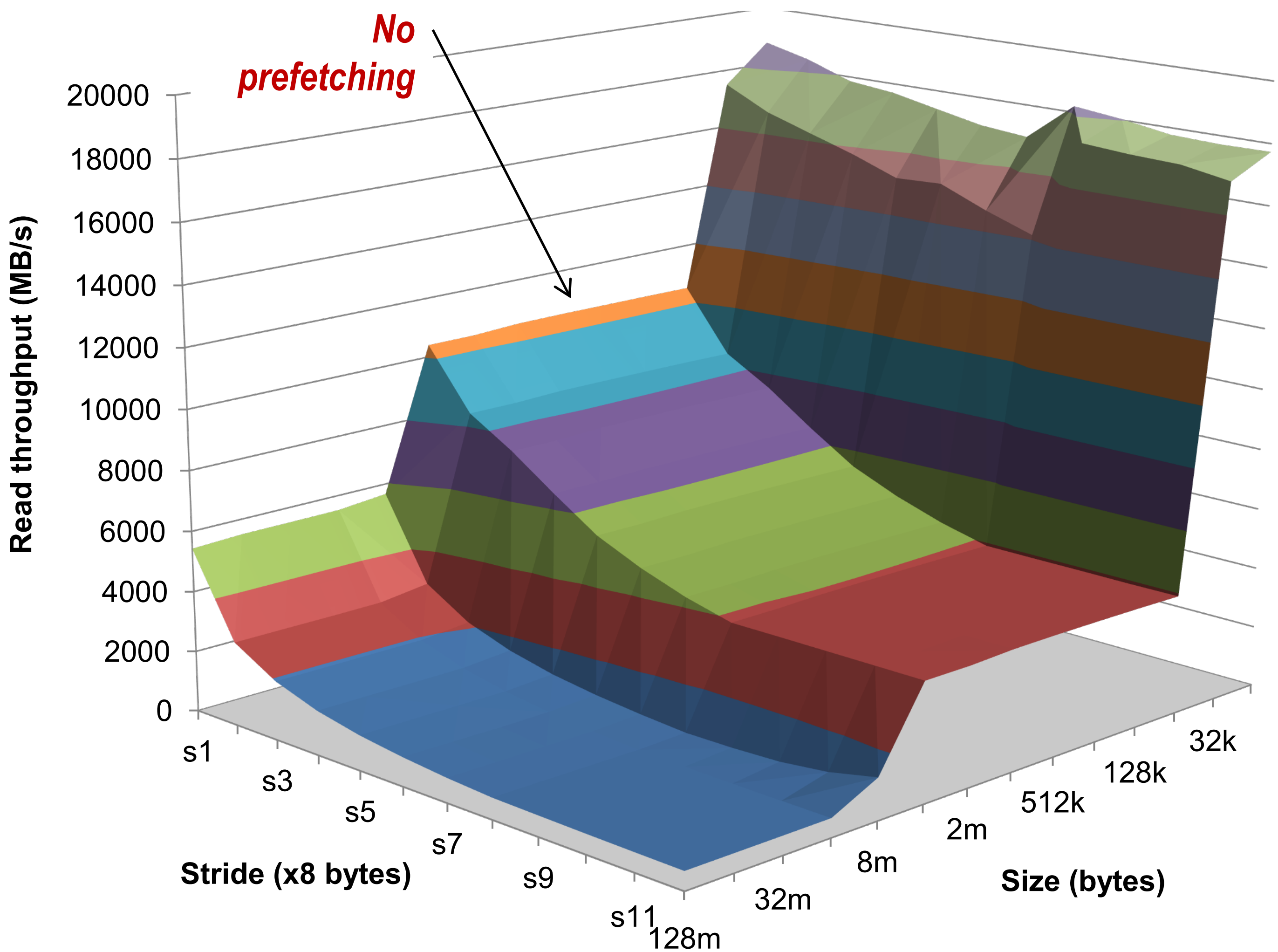
Throughput for size = 128K



$$Throughput = \frac{10^6}{8.0s + 24.3}$$

2008 Memory Mountain

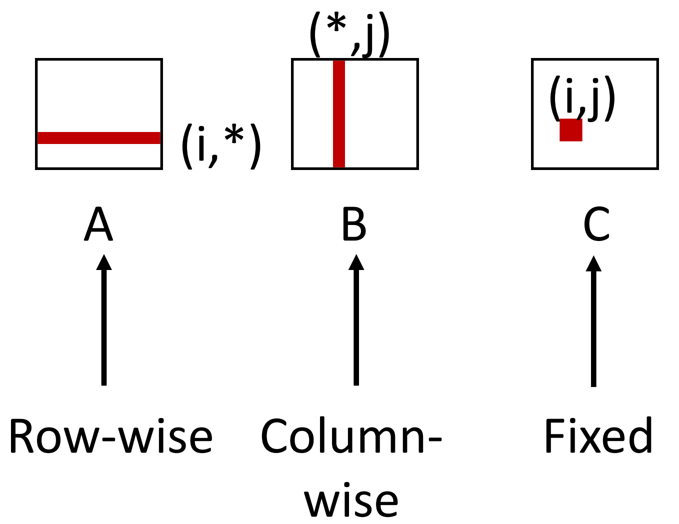
Core 2 Duo
2.4 GHz
32 KB L1 d-cache
6MB L2 cache
64 B block size



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
  }
}                              matmult/mm.c
```

Inner loop:



Misses per inner loop iteration:

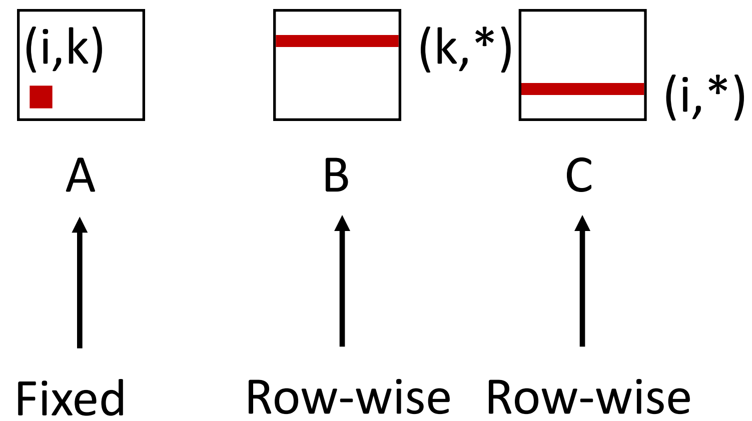
A	B	C
0.25	1.0	0.0

Block size = 32B (four doubles)

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:

A	B	C
0.0	0.25	0.25

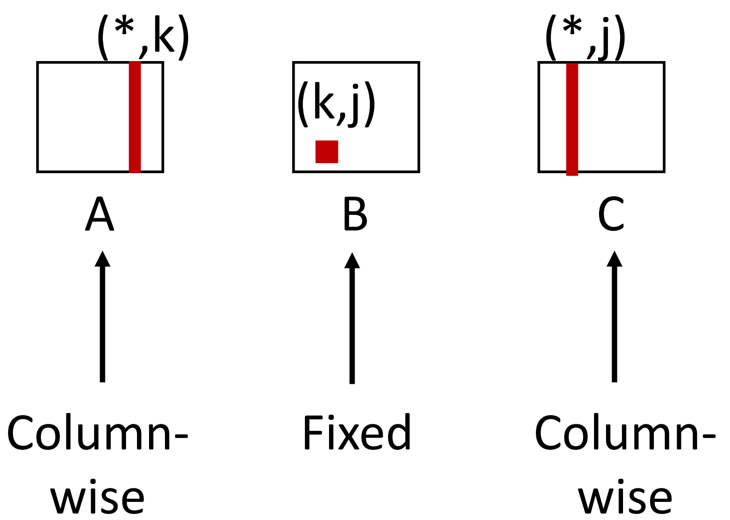
Block size = 32B (four doubles)

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

Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Block size = 32B (four doubles)