

The Hardware/Software Interface

CSE351 Winter 2013

Memory and Caches II

Types of Cache Misses

■ Cold (compulsory) miss

- Occurs on very first access to a block

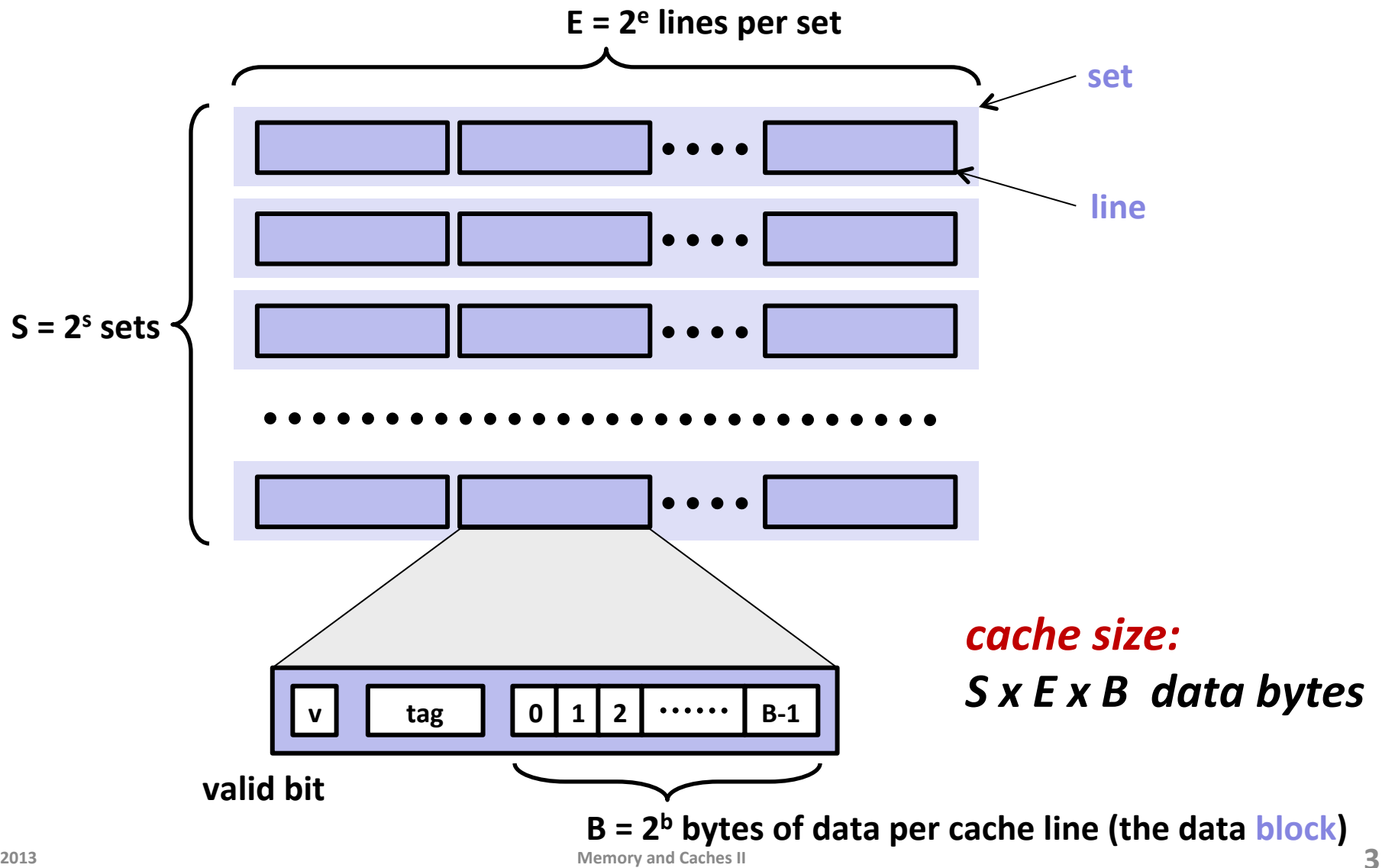
■ Conflict miss

- Occurs when some block is evicted out of the cache, but then that block is referenced again later
- Conflict misses occur when the cache is large enough, but multiple data blocks all map to the same slot
 - e.g., if blocks 0 and 8 map to the same cache slot, then referencing 0, 8, 0, 8, ... would miss every time
 - Conflict misses may be reduced by increasing the associativity of the cache

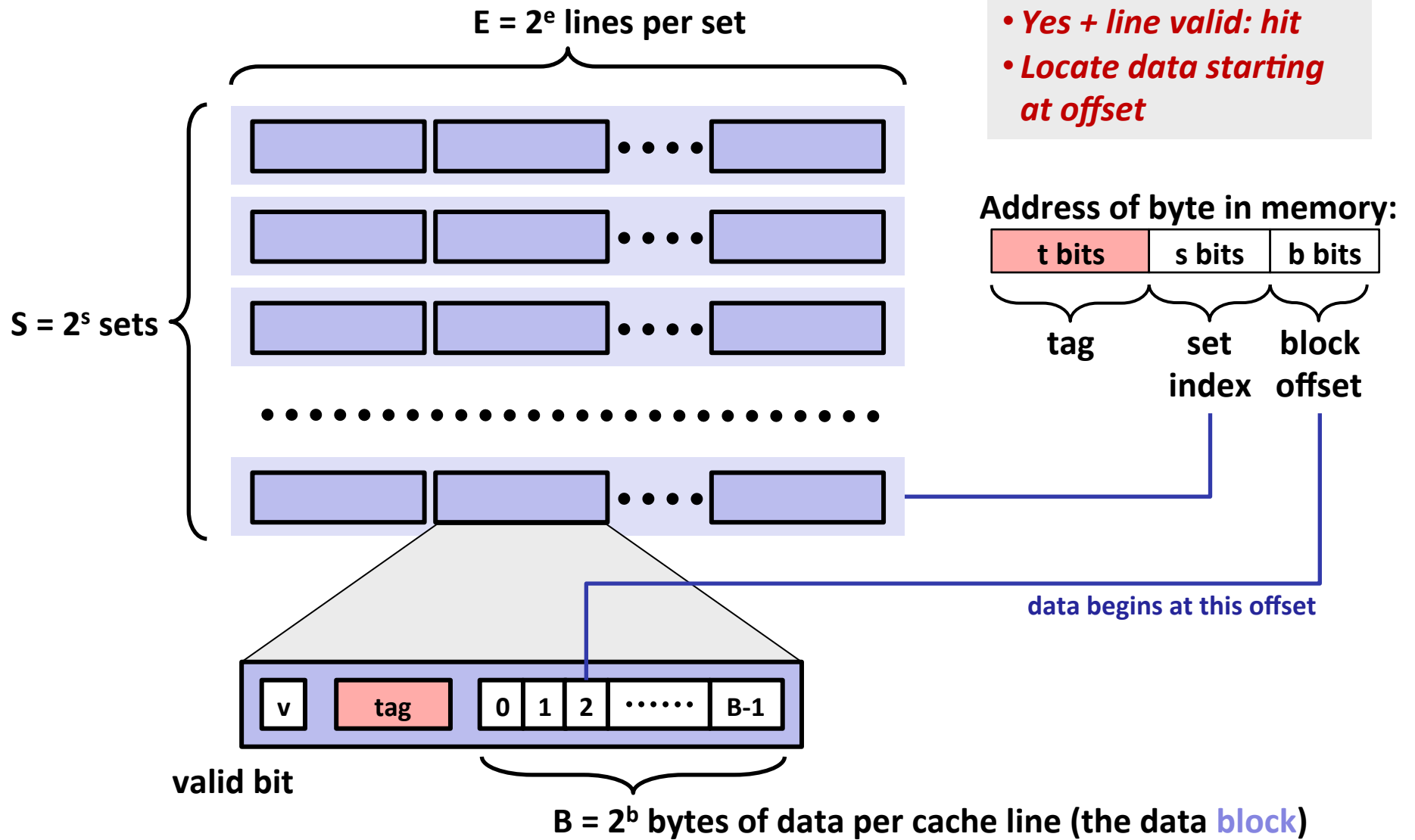
■ Capacity miss

- Occurs when the set of active cache blocks (the working set) is larger than the cache (just won't fit)

General Cache Organization (S, E, B)



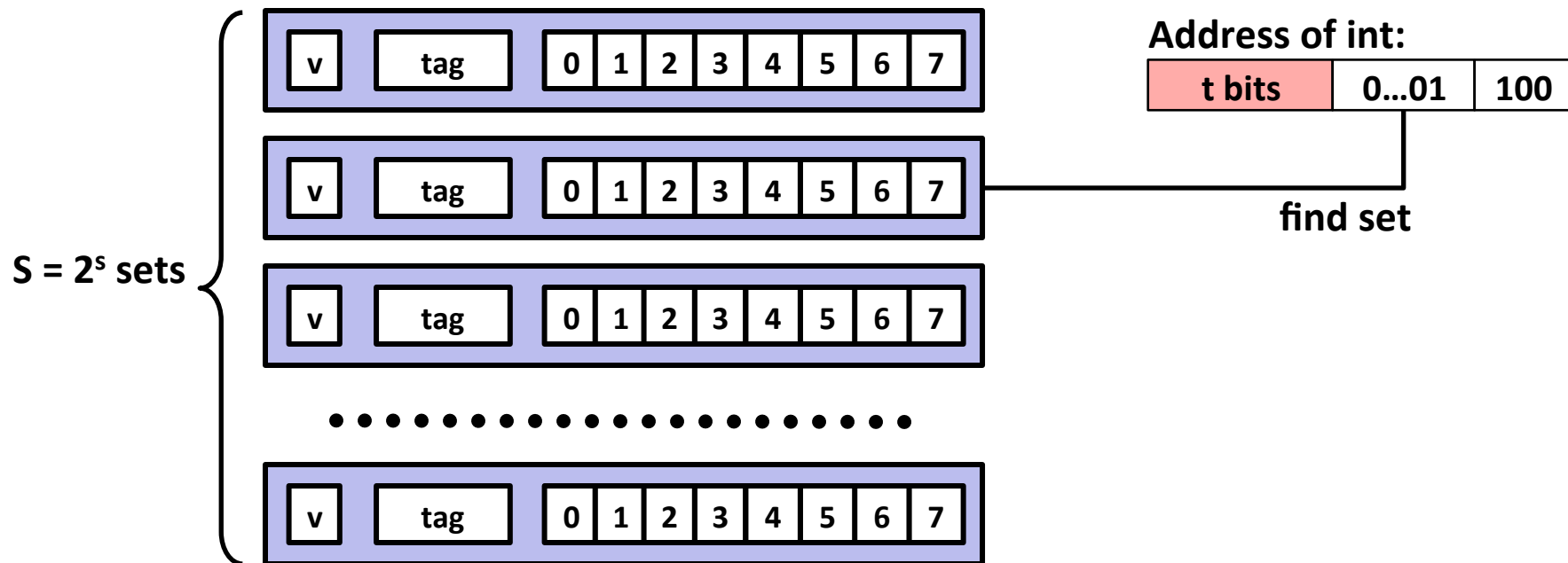
Cache Read



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

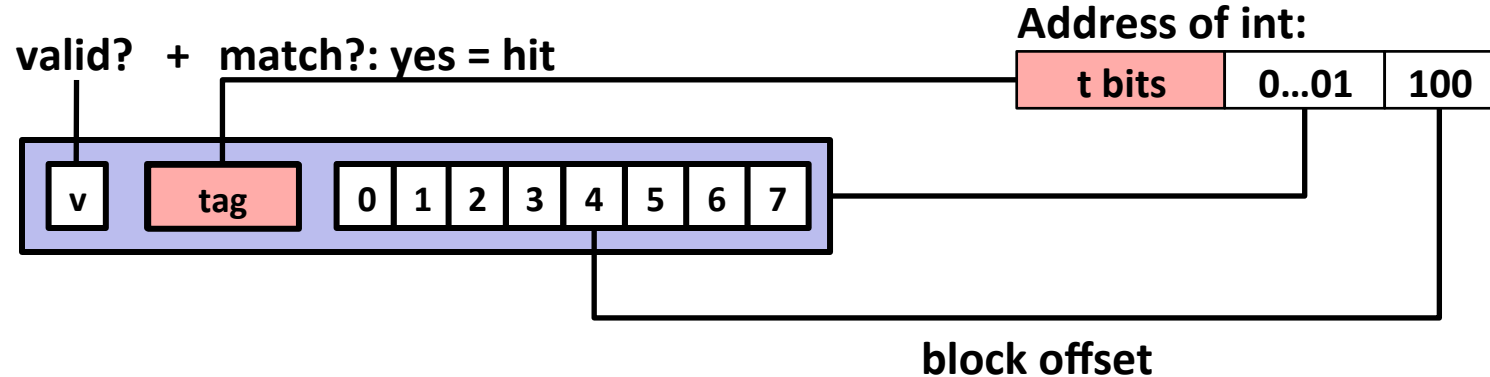
Example: Direct-Mapped Cache (E = 1)

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



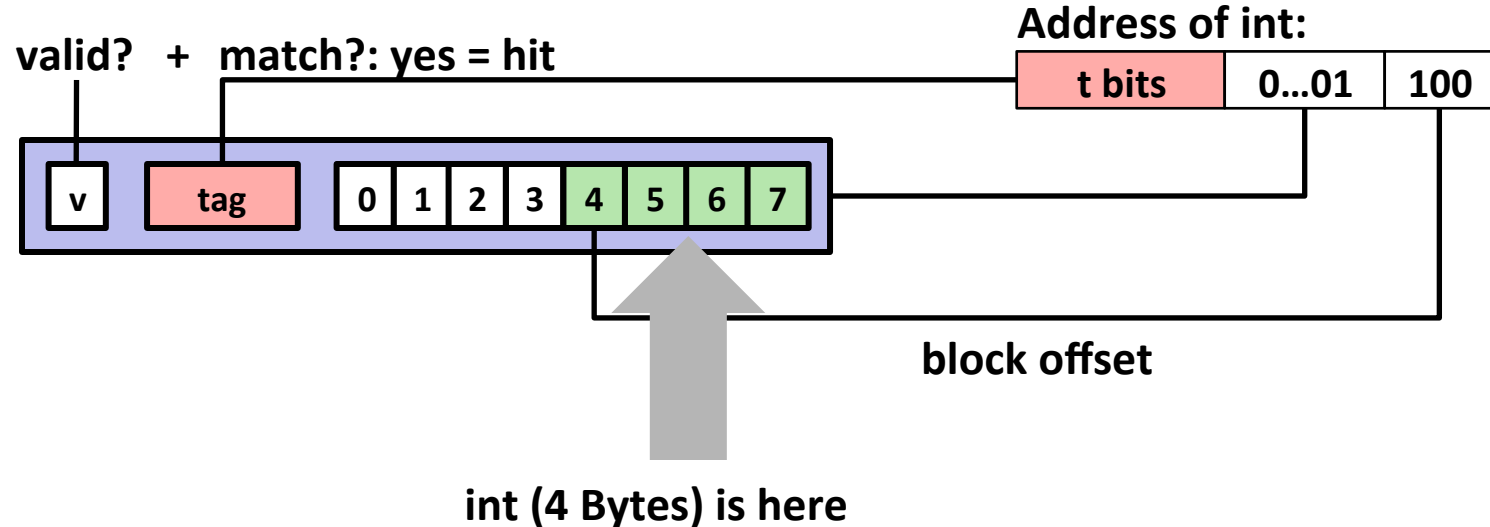
Example: Direct-Mapped Cache (E = 1)

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



Example: Direct-Mapped Cache (E = 1)

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



No match: old line is evicted and replaced

Example (for E = 1)

```
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;

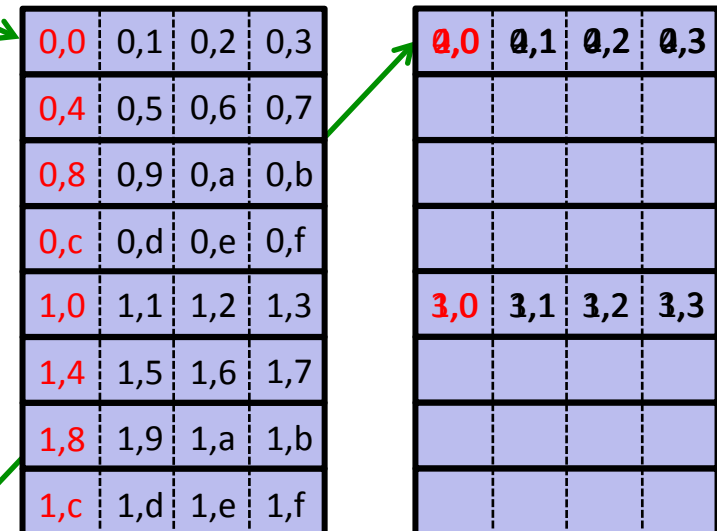
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Assume sum, i, j in registers
Address of an aligned element
of a : $aa \dots ayyyxxxx000$

Assume: cold (empty) cache
3 bits for set, 5 bits for offset

$aa \dots ayyy \quad yxx \quad xx000$

0,0: $aa \dots a000 \quad 000 \quad 00000$



32 B = 4 doubles

4 misses per row of array
 $4 \cdot 16 = 64$ misses

32 B = 4 doubles

every access a miss
 $16 \cdot 16 = 256$ misses

Example (for E = 1)

```
float dotprod(float x[8], float y[8])
{
    float sum = 0;
    int i;

    for (i = 0; i < 8; i++)
        sum += x[i]*y[i];
    return sum;
}
```

In this example, cache blocks are
16 bytes; 8 sets in cache

How many block offset bits?

How many set index bits?

Address bits: ttt...t sss bbbb

$B = 16 = 2^b$: $b=4$ offset bits

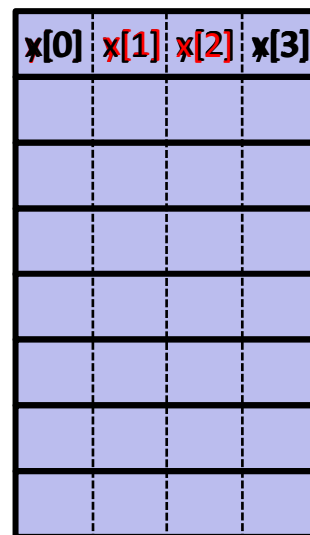
$S = 8 = 2^s$: $s=3$ index bits

0: 000...0 000 0000

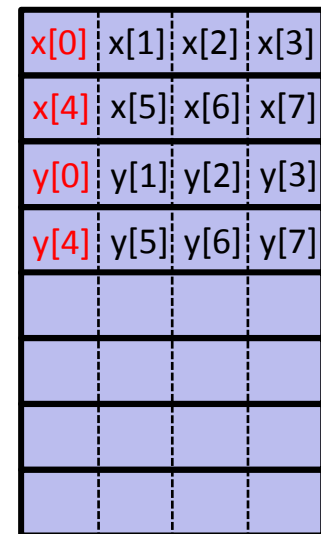
128: 000...1 000 0000

160: 000...1 010 0000

if x and y have aligned
starting addresses,
e.g., $\&x[0] = 0$, $\&y[0] = 128$



if x and y have unaligned
starting addresses,
e.g., $\&x[0] = 0$, $\&y[0] = 160$

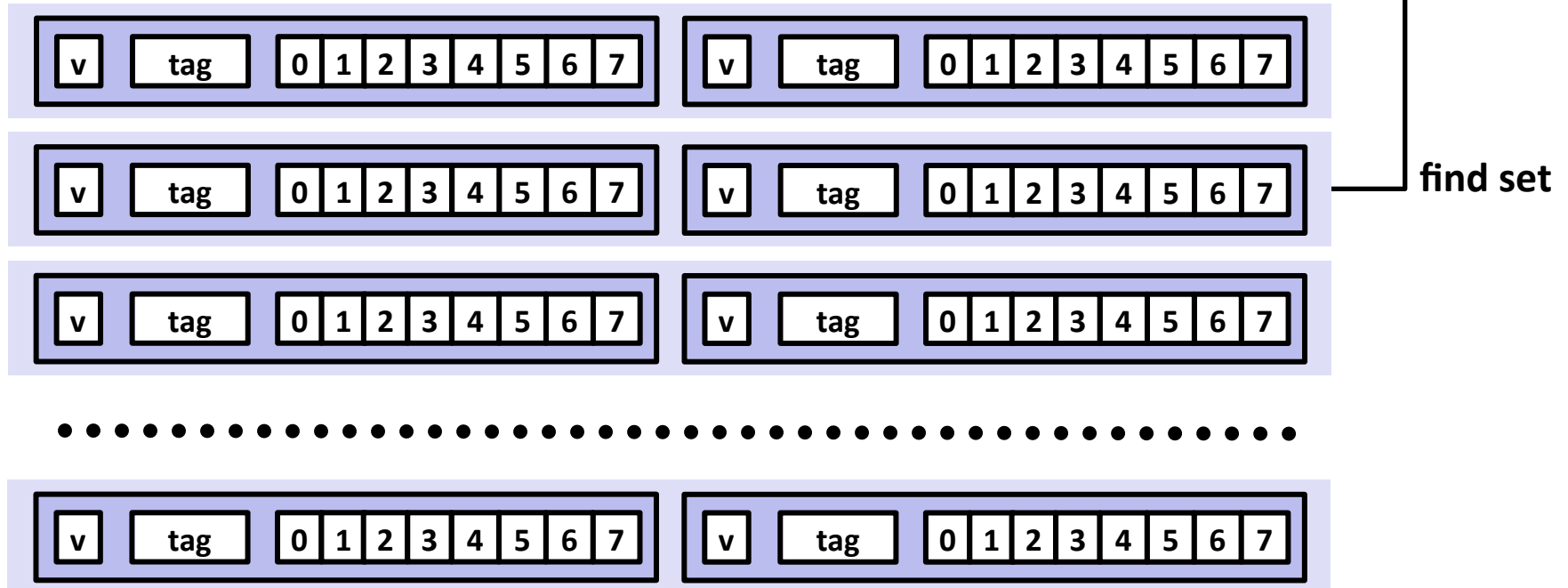
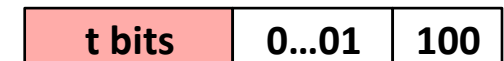


E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

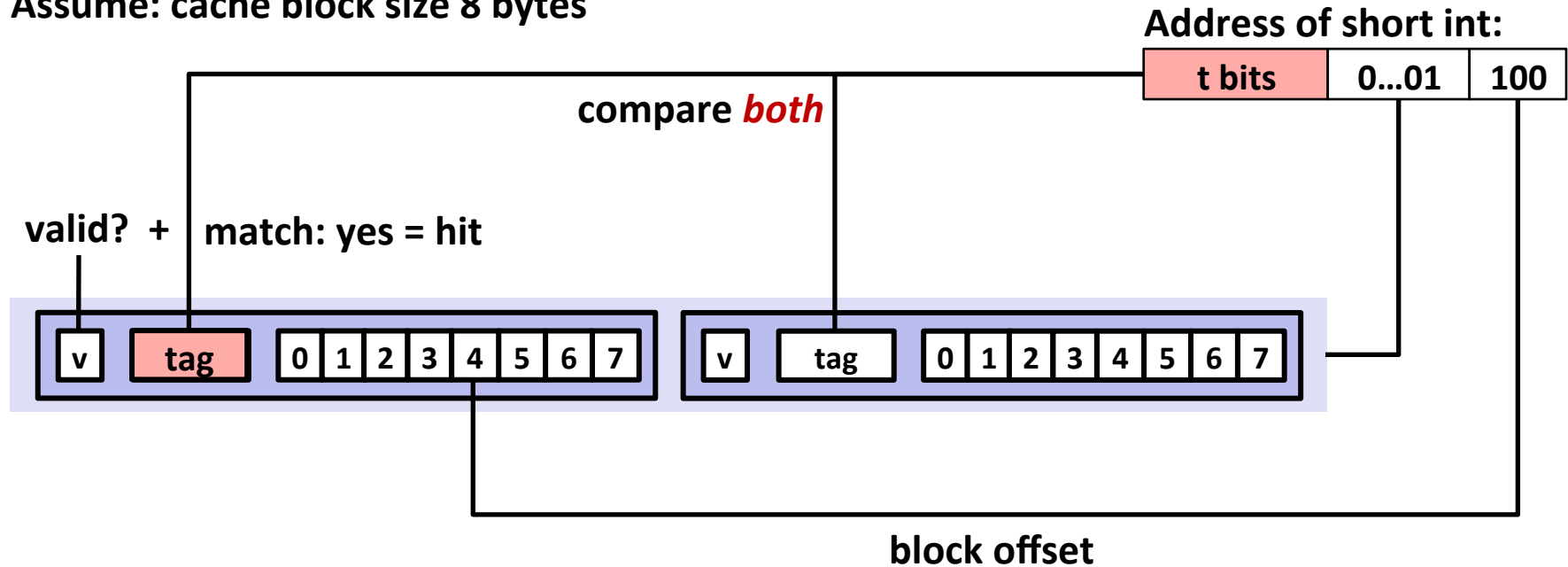
Address of short int:



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

E = 2: Two lines per set

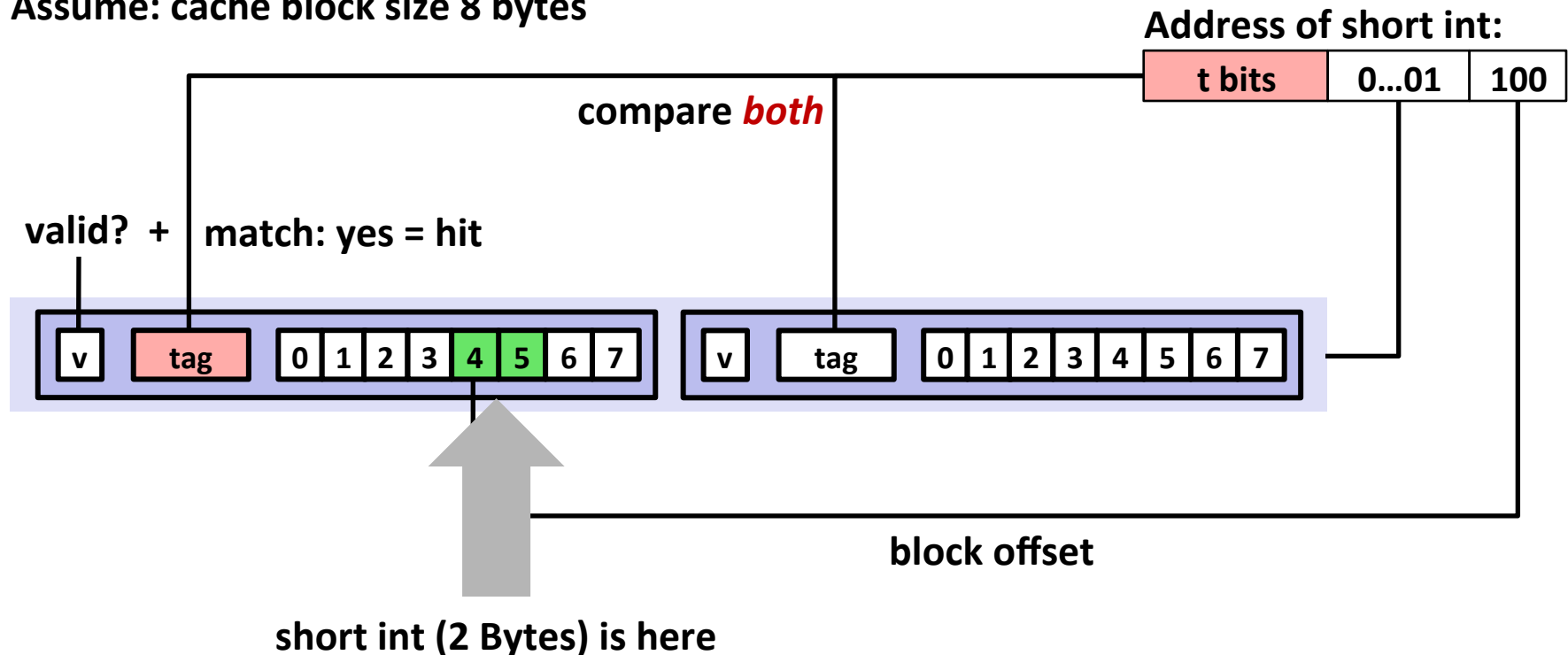
Assume: cache block size 8 bytes



E-way Set-Associative Cache (Here: 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), ...

Example (for E = 2)

```
float dotprod(float x[8], float y[8])
{
    float sum = 0;
    int i;

    for (i = 0; i < 8; i++)
        sum += x[i]*y[i];
    return sum;
}
```

If x and y have aligned starting addresses, e.g. $\&x[0] = 0$, $\&y[0] = 128$, can still fit both because two lines in each set

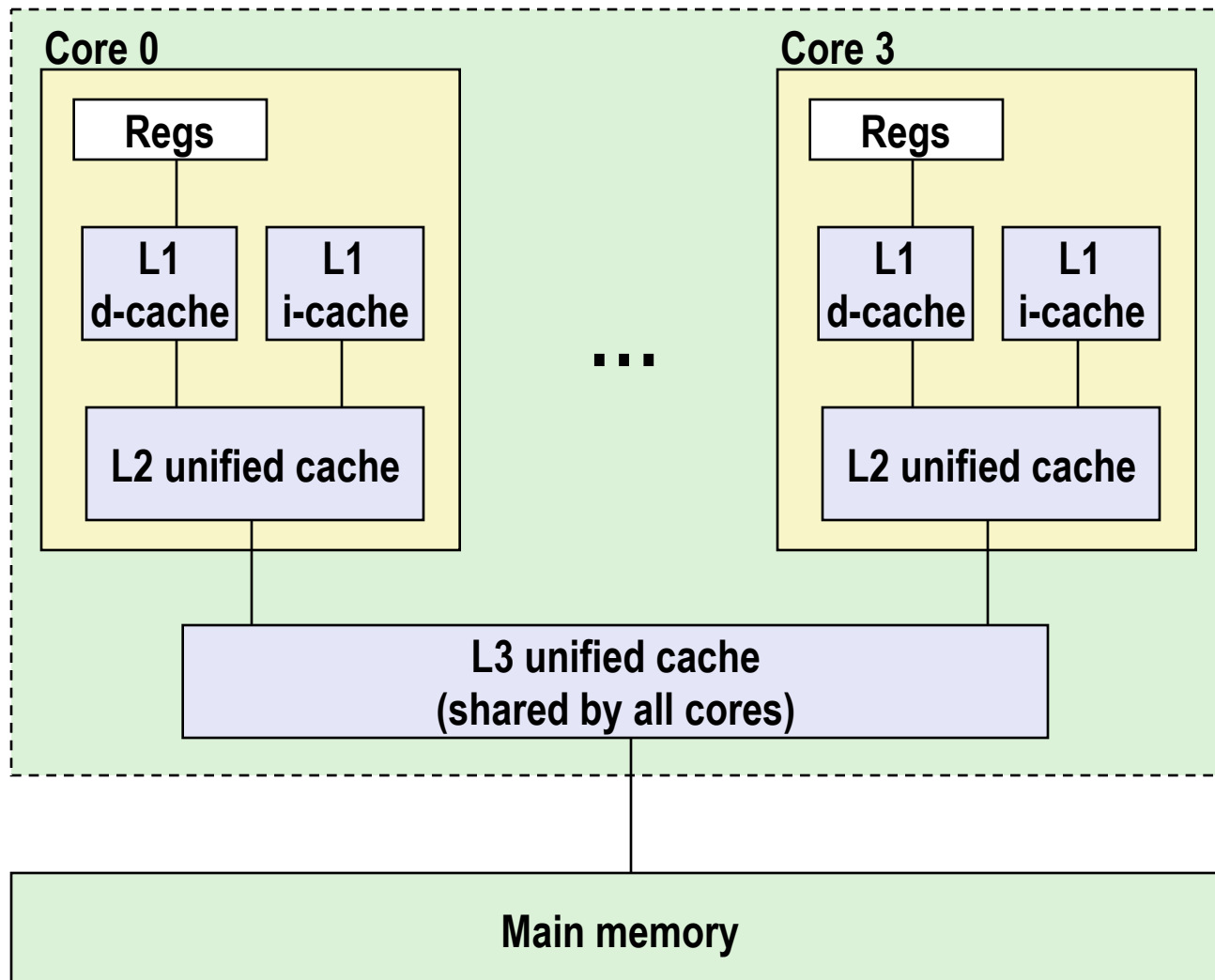
x[0]	x[1]	x[2]	x[3]	y[0]	y[1]	y[2]	y[3]
x[4]	x[5]	x[6]	x[7]	y[4]	y[5]	y[6]	y[7]

Fully Set-Associative Caches ($S = 1$)

- **Fully-associative caches have all lines in one single set, $S = 1$**
 - $E = C / B$, where C is total cache size
 - Since, $S = (C / B) / E$, therefore, $S = 1$
- **Direct-mapped caches have $E = 1$**
 - $S = (C / B) / E = C / B$
- **Tag matching is more expensive in associative caches**
 - Fully-associative cache needs C / B tag comparators: one for every line!
 - Direct-mapped cache needs just 1 tag comparator
 - In general, an E -way set-associative cache needs E tag comparators
- **Tag size, assuming m address bits ($m = 32$ for IA32):**
 - $m - \log_2 S - \log_2 B$

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: 11 cycles

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

Block size: 64 bytes for
all caches.

What about writes?

- **Multiple copies of data exist:**
 - L1, L2, possibly L3, main memory
- **What to do on a write-hit?**
 - **Write-through** (write immediately to memory)
 - **Write-back** (defer write to memory until line is evicted)
 - Need a *dirty bit* to indicate if line is different from memory or not
- **What to do on a write-miss?**
 - **Write-allocate** (load into cache, update line in cache)
 - Good if more writes to the location follow
 - **No-write-allocate** (just write immediately to memory)
- **Typical caches:**
 - Write-back + Write-allocate, usually
 - Write-through + No-write-allocate, occasionally

Software Caches are More Flexible

■ Examples

- File system buffer caches, web browser caches, etc.

■ Some design differences

- Almost always fully-associative
 - so, no placement restrictions
 - index structures like hash tables are common (for placement)
- Often use complex replacement policies
 - misses are very expensive when disk or network involved
 - worth thousands of cycles to avoid them
- Not necessarily constrained to single “block” transfers
 - may fetch or write-back in larger units, opportunistically

Optimizations for the Memory Hierarchy

- **Write code that has locality**
 - Spatial: access data contiguously
 - Temporal: make sure access to the same data is not too far apart in time
- **How to achieve?**
 - Proper choice of algorithm
 - Loop transformations

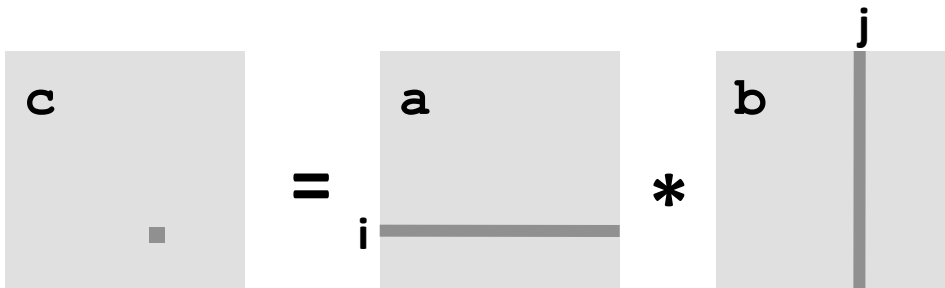
Example: Matrix Multiplication

```

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];
}

```



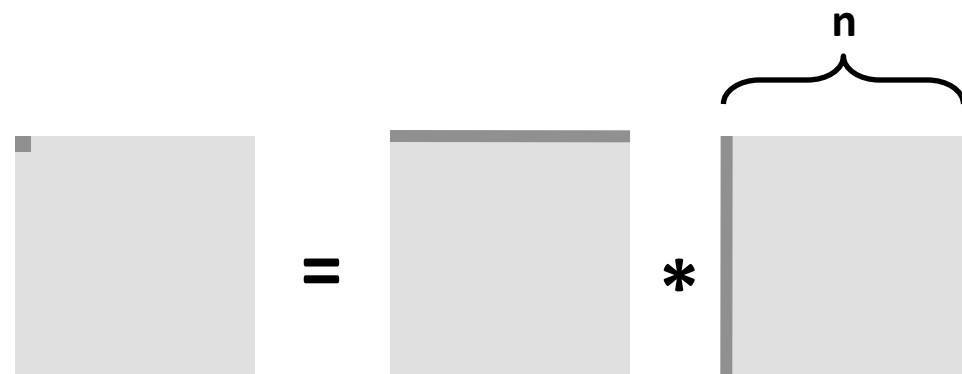
Cache Miss Analysis

■ Assume:

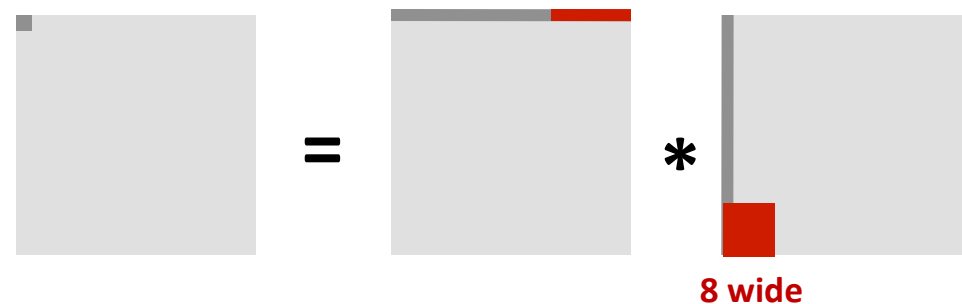
- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size $C \ll n$ (much smaller than n)

■ First iteration:

- $n/8 + n = 9n/8$ misses
(omitting matrix c)



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



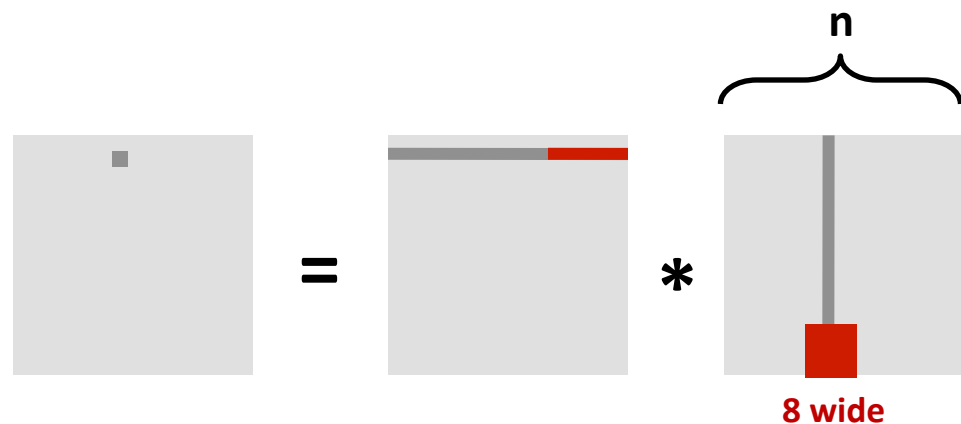
Cache Miss Analysis

■ Assume:

- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size $C \ll n$ (much smaller than n)

■ Other iterations:

- Again:
 $n/8 + n = 9n/8$ misses
 (omitting matrix c)



■ Total misses:

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

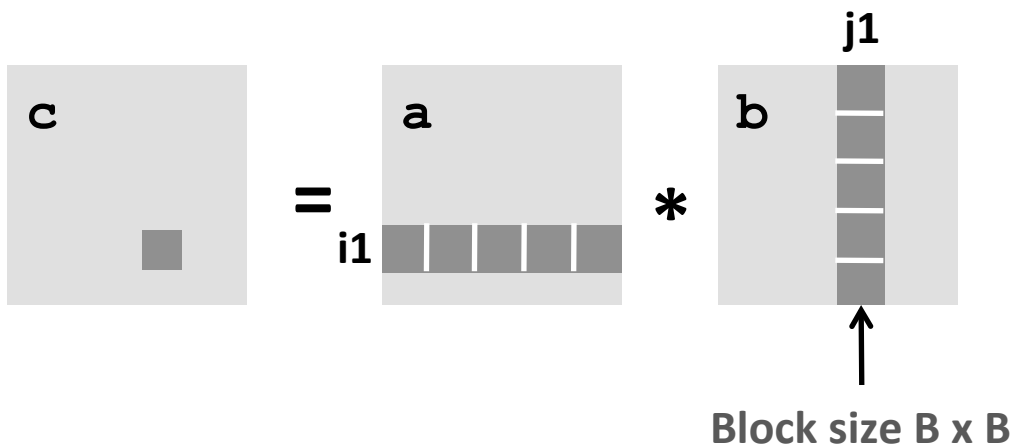
Blocked Matrix Multiplication

```

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];
}

```



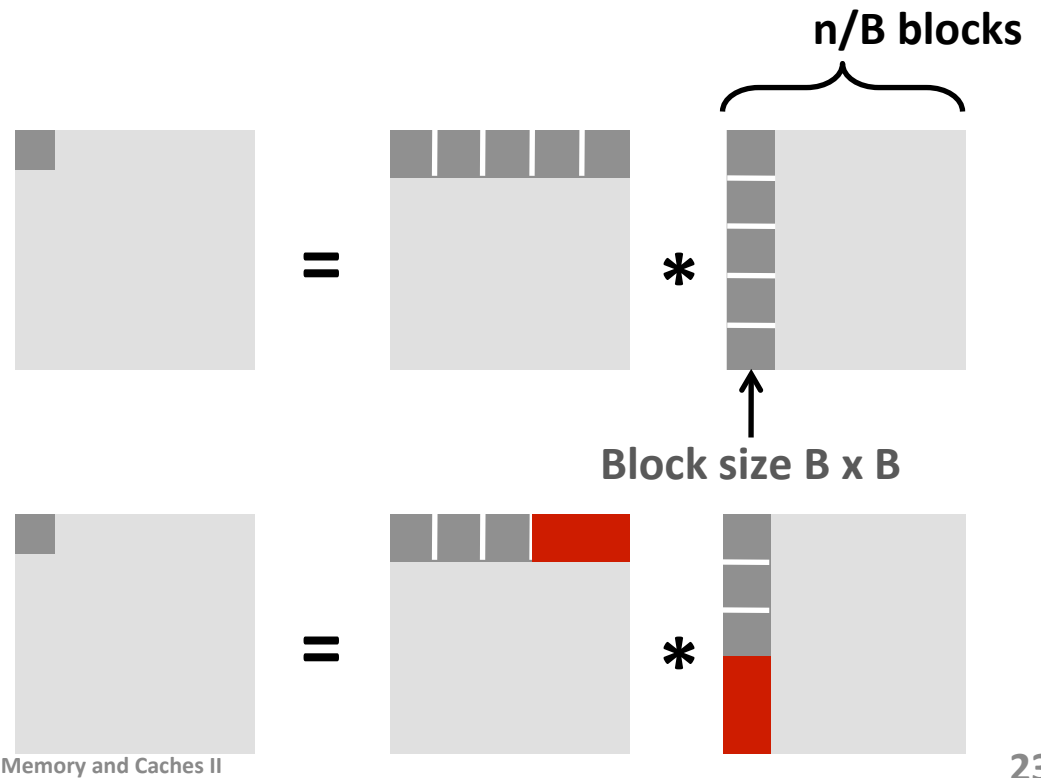
Cache Miss Analysis

■ Assume:

- Cache block = 64 bytes = 8 doubles
- Cache size $C \ll n$ (much smaller than n)
- Three blocks  fit into cache: $3B^2 < C$


■ First (block) iteration:

- $B^2/8$ misses for each block
- $2n/B * B^2/8 = nB/4$
(omitting matrix c)



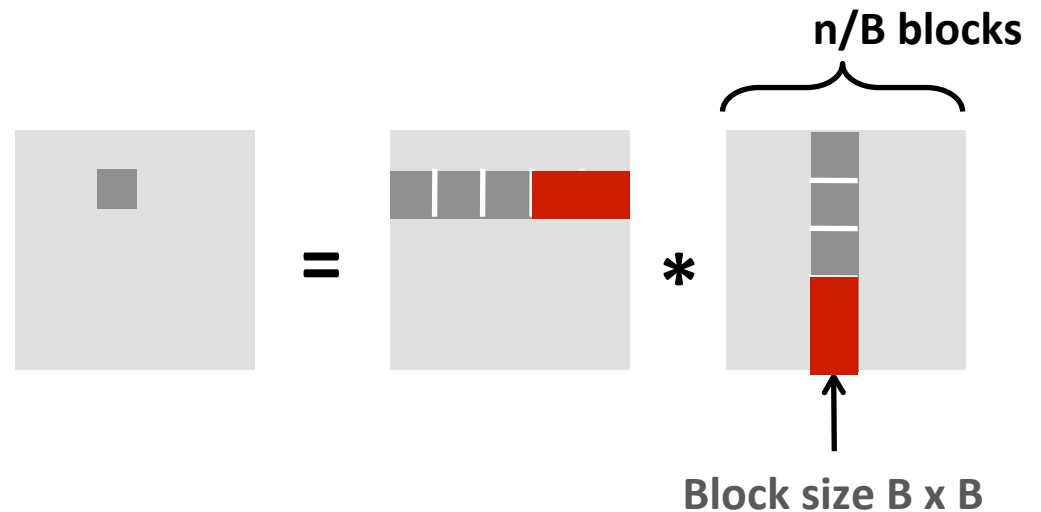
Cache Miss Analysis

■ Assume:

- Cache block = 64 bytes = 8 doubles
- Cache size $C \ll n$ (much smaller than n)
- Three blocks  fit into cache: $3B^2 < C$

■ Other (block) iterations:

- Same as first iteration
- $2n/B * B^2/8 = nB/4$



■ Total misses:

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

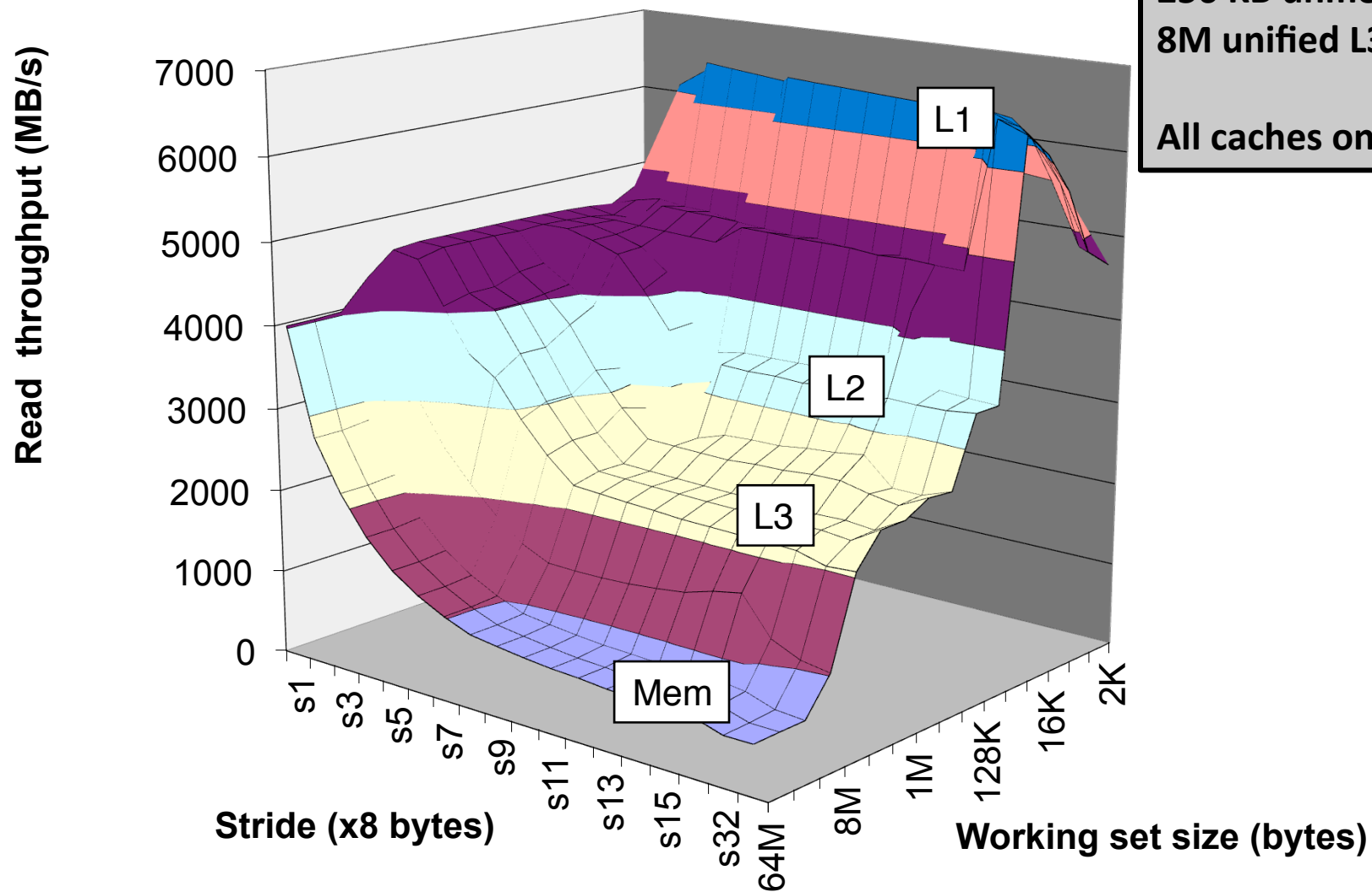
Summary

- No blocking: $(9/8) * n^3$
- Blocking: $1/(4B) * n^3$
- If $B = 8$ difference is $4 * 8 * 9 / 8 = 36x$
- If $B = 16$ difference is $4 * 16 * 9 / 8 = 72x$
- Suggests largest possible block size B , but limit $3B^2 < C!$
- Reason for dramatic difference:
 - Matrix multiplication has inherent temporal locality:
 - Input data: $3n^2$, computation $2n^3$
 - Every array element used $O(n)$ times!
 - But program has to be written properly

Cache-Friendly Code

- **Programmer 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)
 - Focus on inner loop code

The Memory Mountain



Intel Core i7
 32 KB L1 i-cache
 32 KB L1 d-cache
 256 KB unified L2 cache
 8M unified L3 cache

All caches on-chip