Cache Memories

Topics

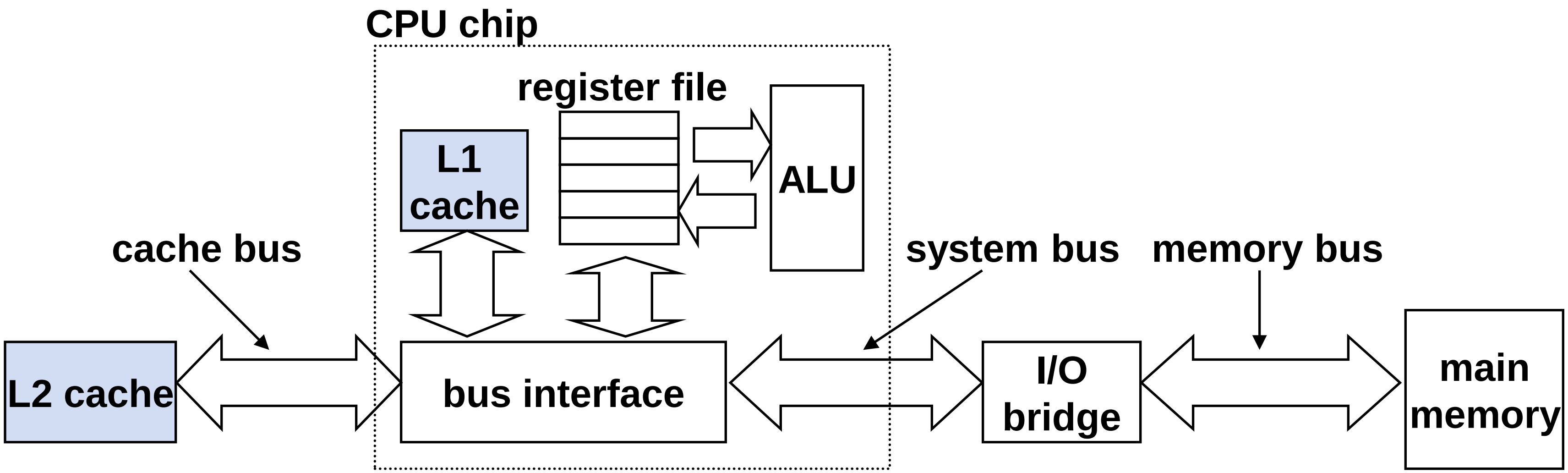
- Generic cache memory organization
- Direct mapped caches
- Set associative caches
- Impact of caches on performance

Next time

- Linking

Cache memories

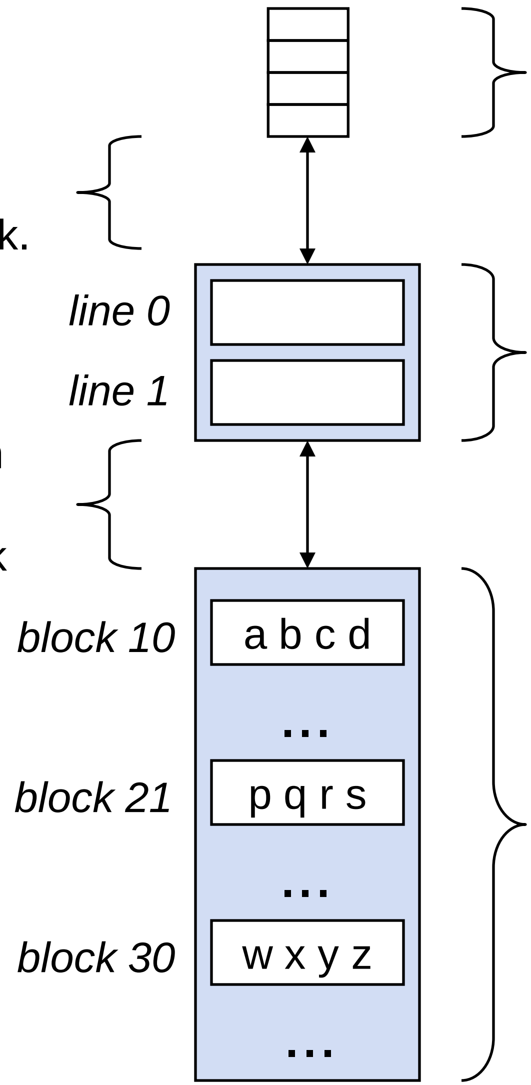
- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in L1, then in L2, …, then in main memory.
- Typical bus structure:



Inserting an L1 cache

The transfer unit between the CPU register file and the cache is a 4-byte block.

The transfer unit between the cache and main memory is a 4-word block (16 bytes).



The tiny, very fast CPU **register file** has room for four 4-byte words.

The small fast **L1 cache** has room for two 4-word blocks.

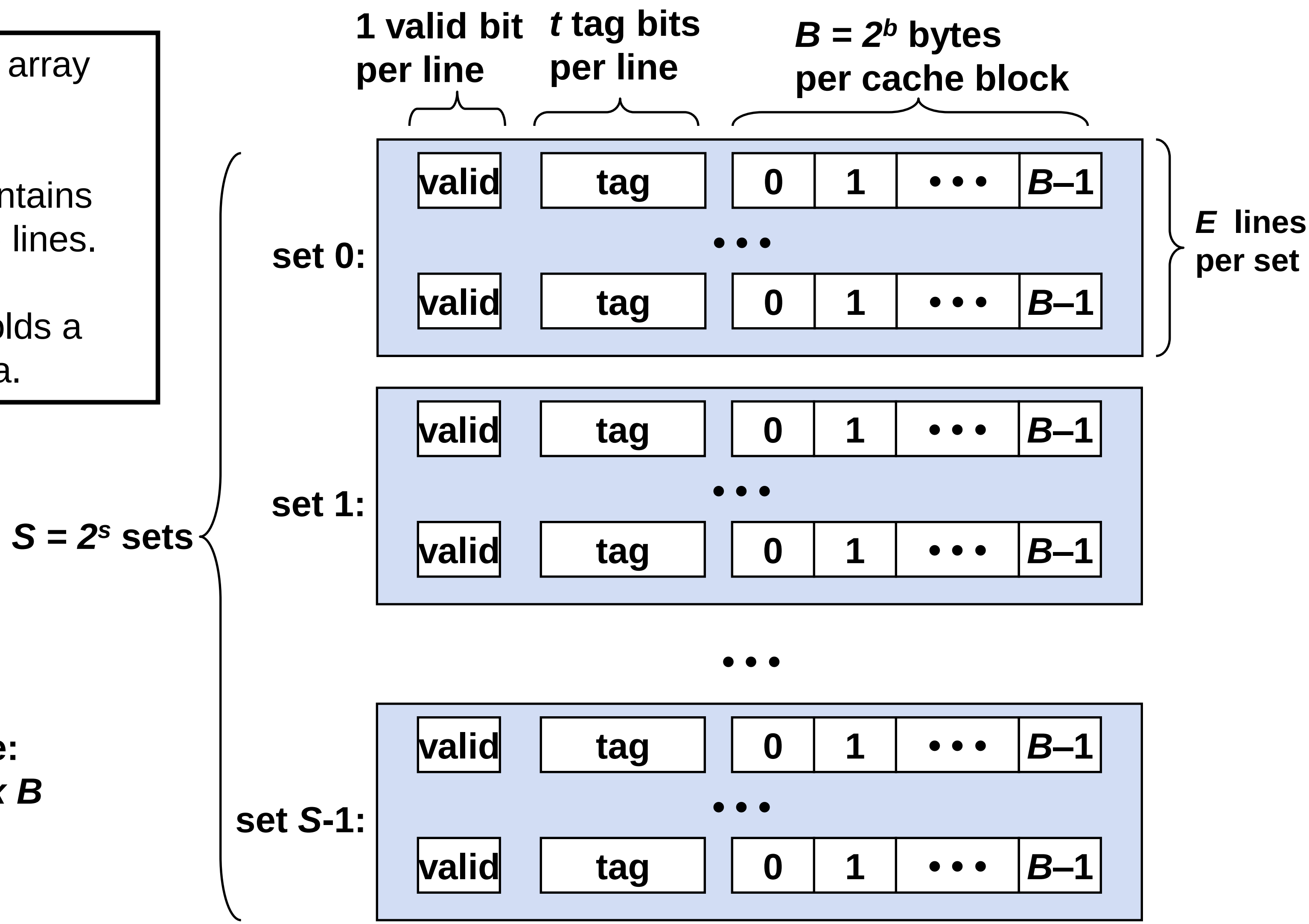
The big slow **main memory** has room for many 4-word blocks.

General org of a cache memory

Cache is an array of sets.

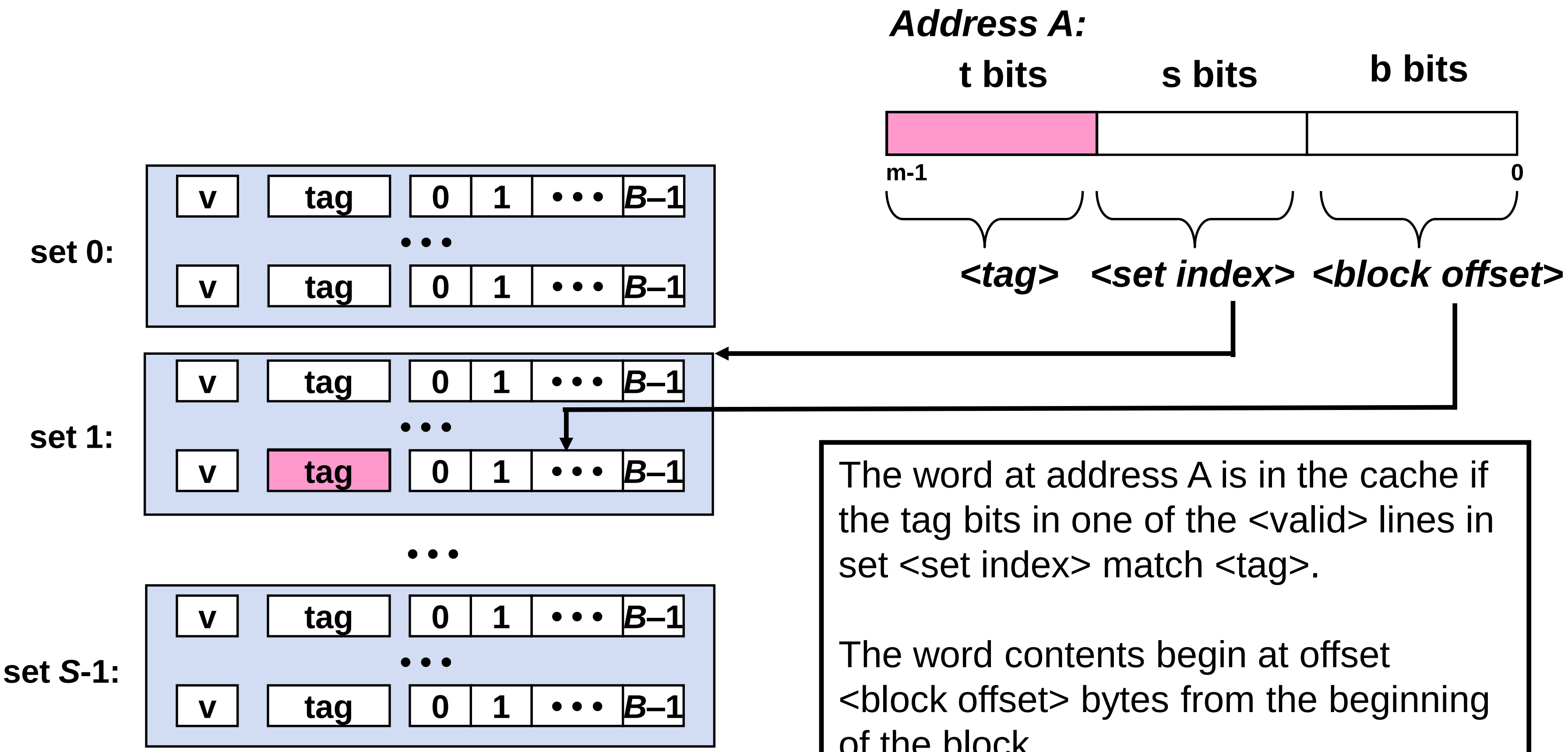
Each set contains one or more lines.

Each line holds a block of data.



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

Addressing caches

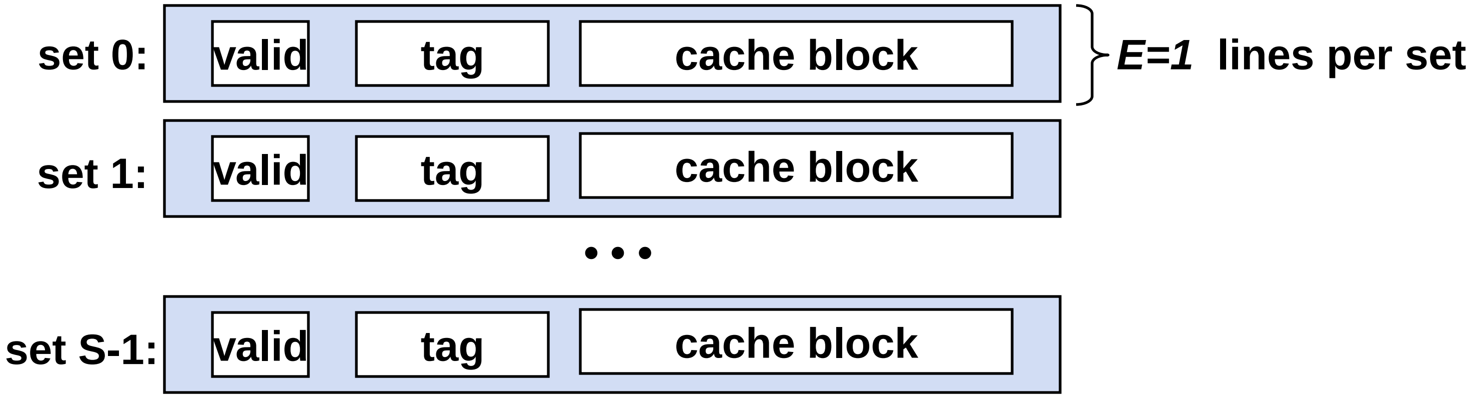


The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.

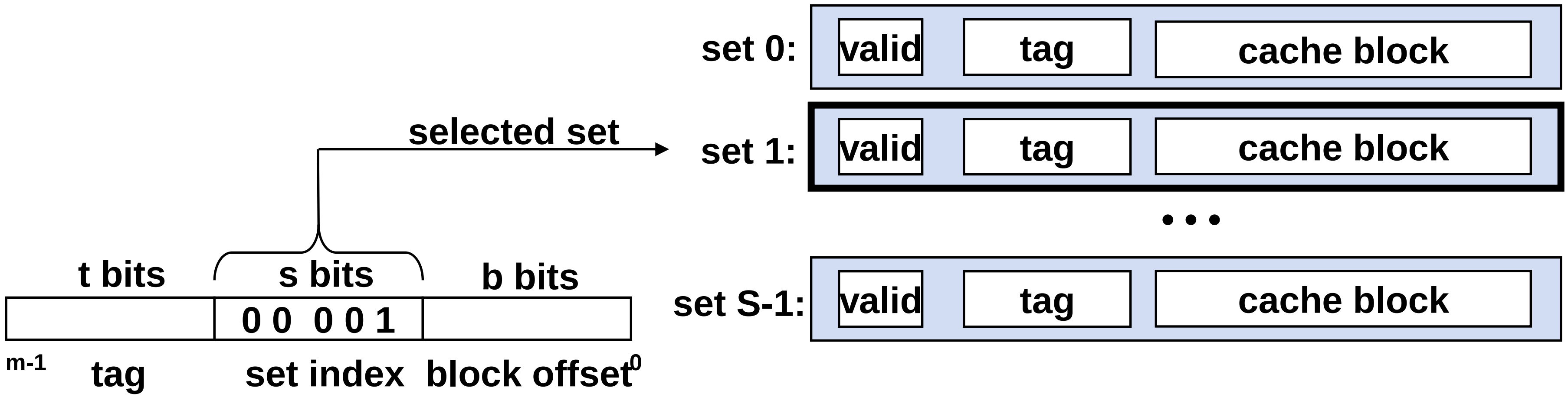
Direct-mapped cache

- Simplest kind of cache
- Characterized by exactly one line per set.



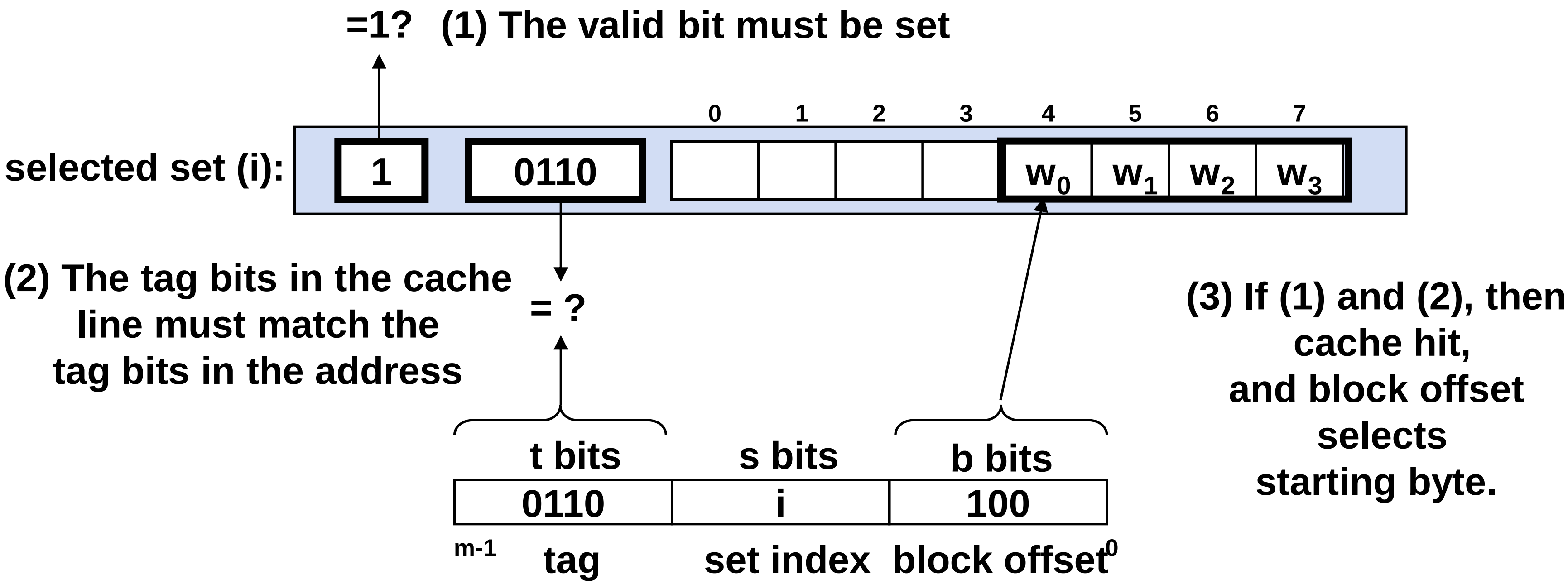
Accessing direct-mapped caches

- Set selection
  - Use the set index bits to determine the set of interest.



Accessing direct-mapped caches

- Line matching and word selection
  - *Line matching*: Find a valid line in the selected set with a matching tag
  - *Word selection*: Then extract the word



Direct-mapped cache simulation

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

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

Address	Tag	Index	Offset	Block #
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Direct-mapped cache simulation

0 [0000_2] *(miss)*

1 [0001_2] *(hit)*

(1)

v	tag	data
1	0	M[0-1]

13 [1101_2] *(miss)*

(3)

v	tag	data
1	0	M[0-1]
1	1	M[12-13]

8 [1000_2] *(miss)*

(4)

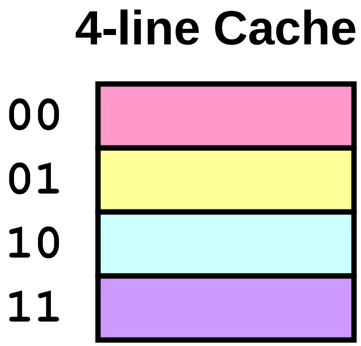
v	tag	data
1	1	M[8-9]
1	1	M[12-13]

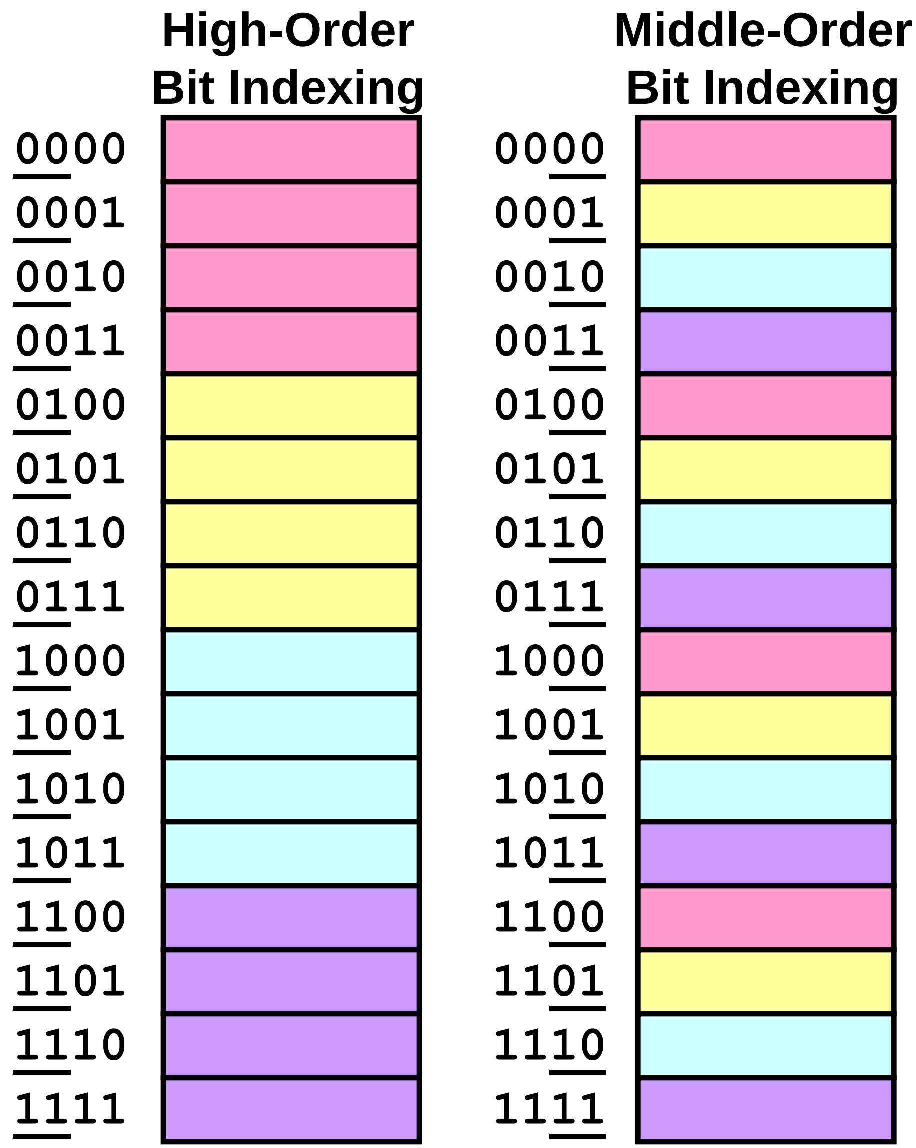
0 [0000_2] *(miss)*

(5)

v	tag	data
1	0	M[0-1]
1	1	M[12-13]

Why use middle bits as index?

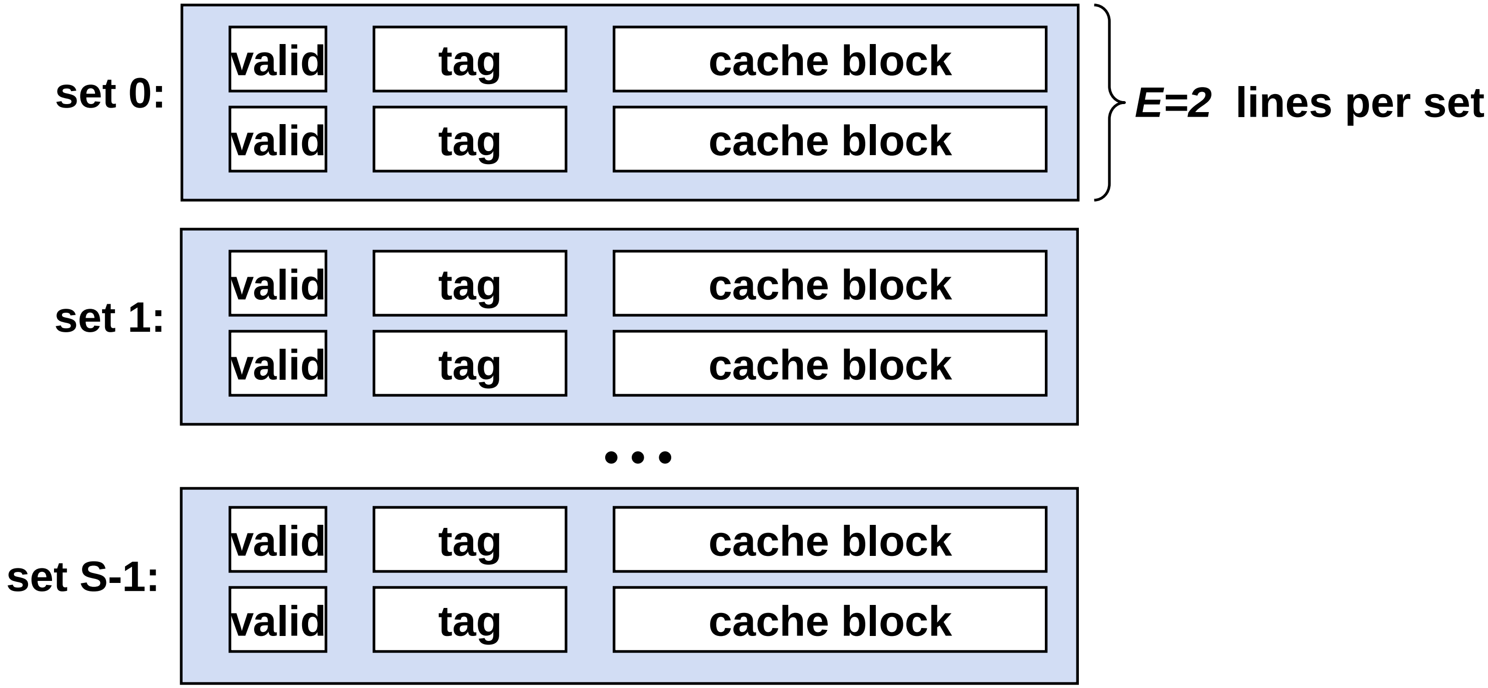




- High-order bit indexing
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality
- Middle-order bit indexing
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time

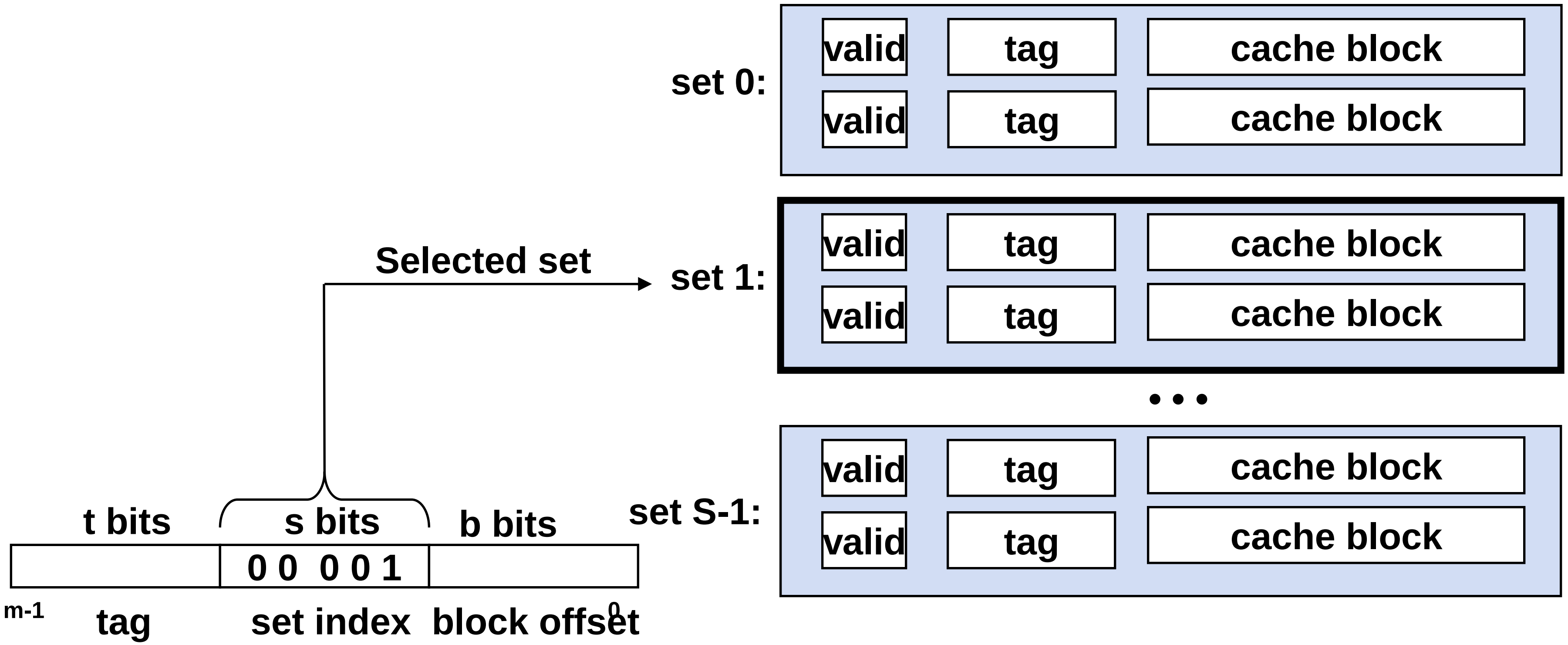
Set associative caches

- Characterized by more than one line per set



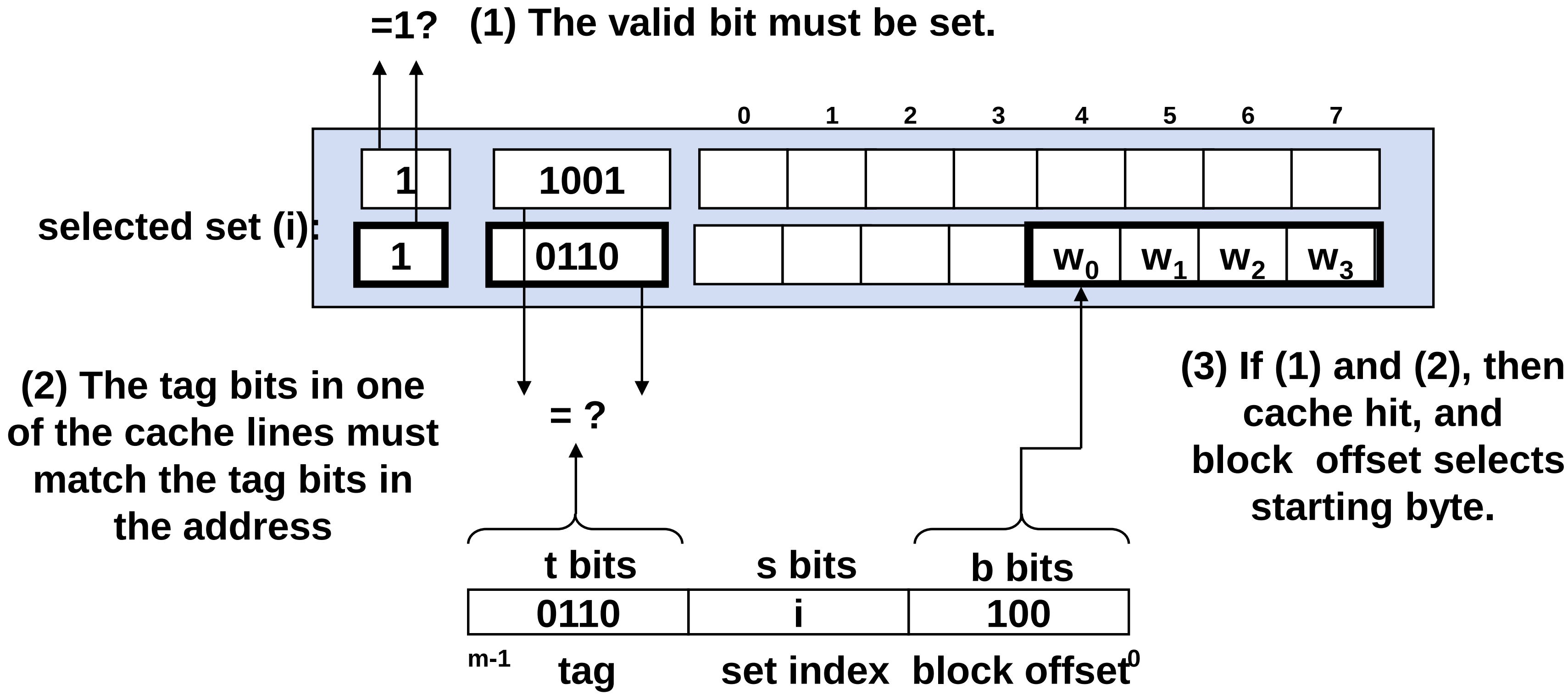
Accessing set associative caches

- Set selection
  – identical to direct-mapped cache



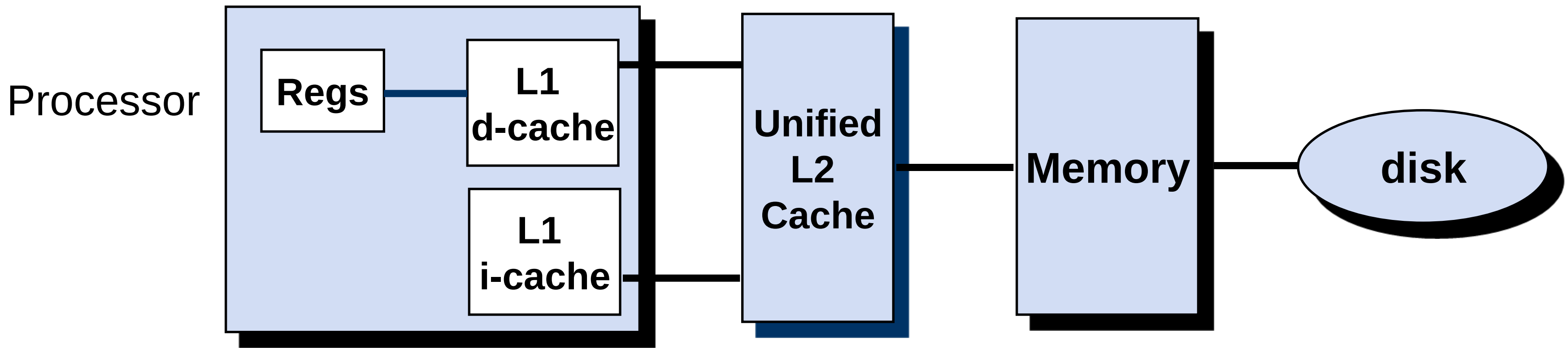
Accessing set associative caches

- Line matching and word selection
  - must compare the tag in each valid line in the selected set.



Multi-level caches

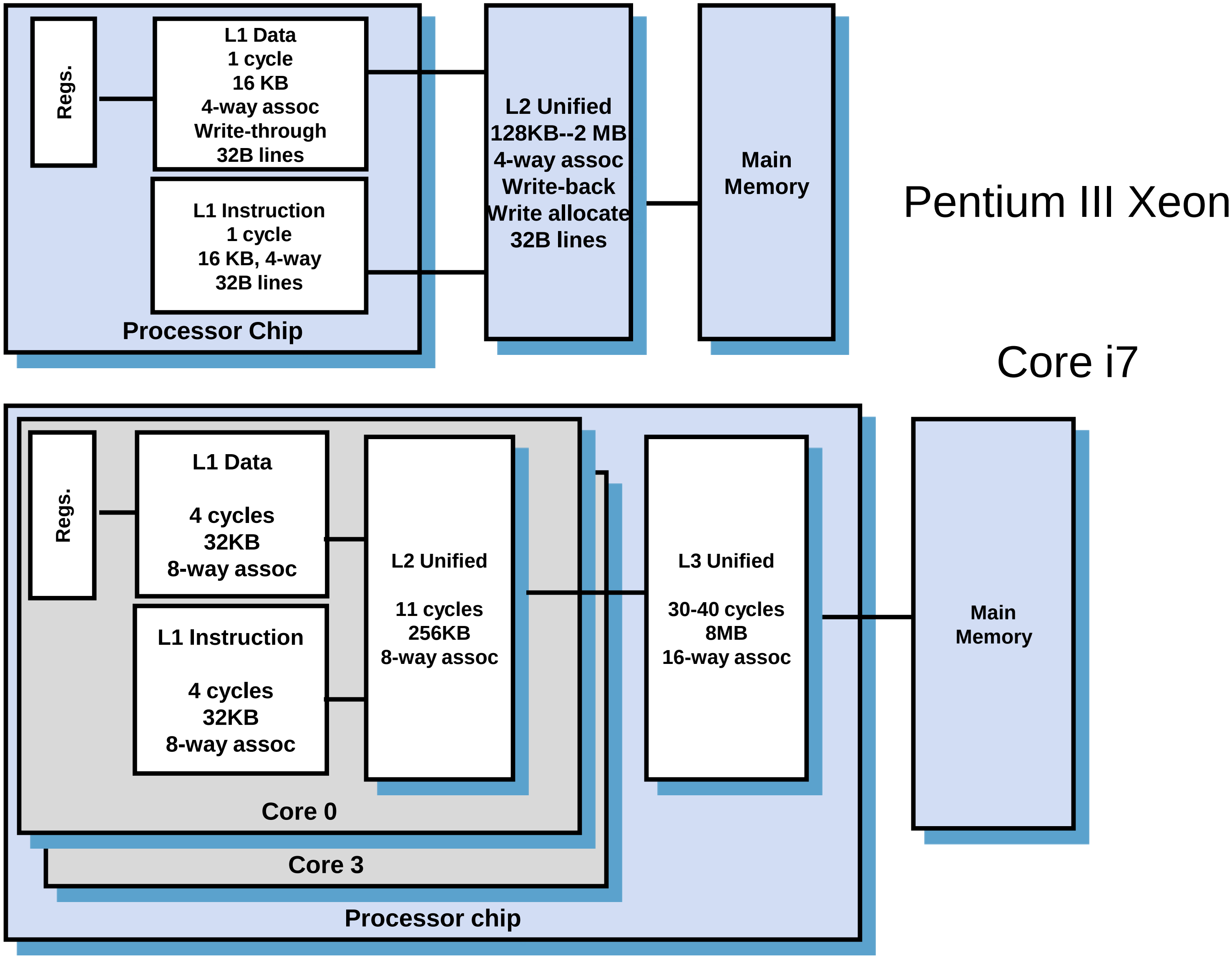
- Options: separate data and instruction caches, or a unified cache



size:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	30 GB
speed:	3 ns	3 ns	6 ns	60 ns	8 ms
$/Mbyte:			$100/MB	$1.50/MB	$0.05/MB
line size:	8 B	32 B	32 B	8 KB	

larger, slower, cheaper →

Pentium III and Core i7 Cache Hierarchy



Cache performance metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses/references)
  - Typical numbers:
    - 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:
    - 1 clock cycle for L1
    - 3-8 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - Typically 25-100 cycles for main memory

Writing cache friendly code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
  - cold cache, 4-byte words, 4-word cache blocks

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

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

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

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

Miss rate = 1/4 = 25%

Miss rate = 100%

The memory mountain

- Read throughput (read bandwidth)
  – Number of bytes read from memory per sec (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

```
/* 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 */
}
```

Memory mountain main routine

- Smaller values of size – smaller working set size, better temporal locality
- Smaller values of stride – better spatial locality

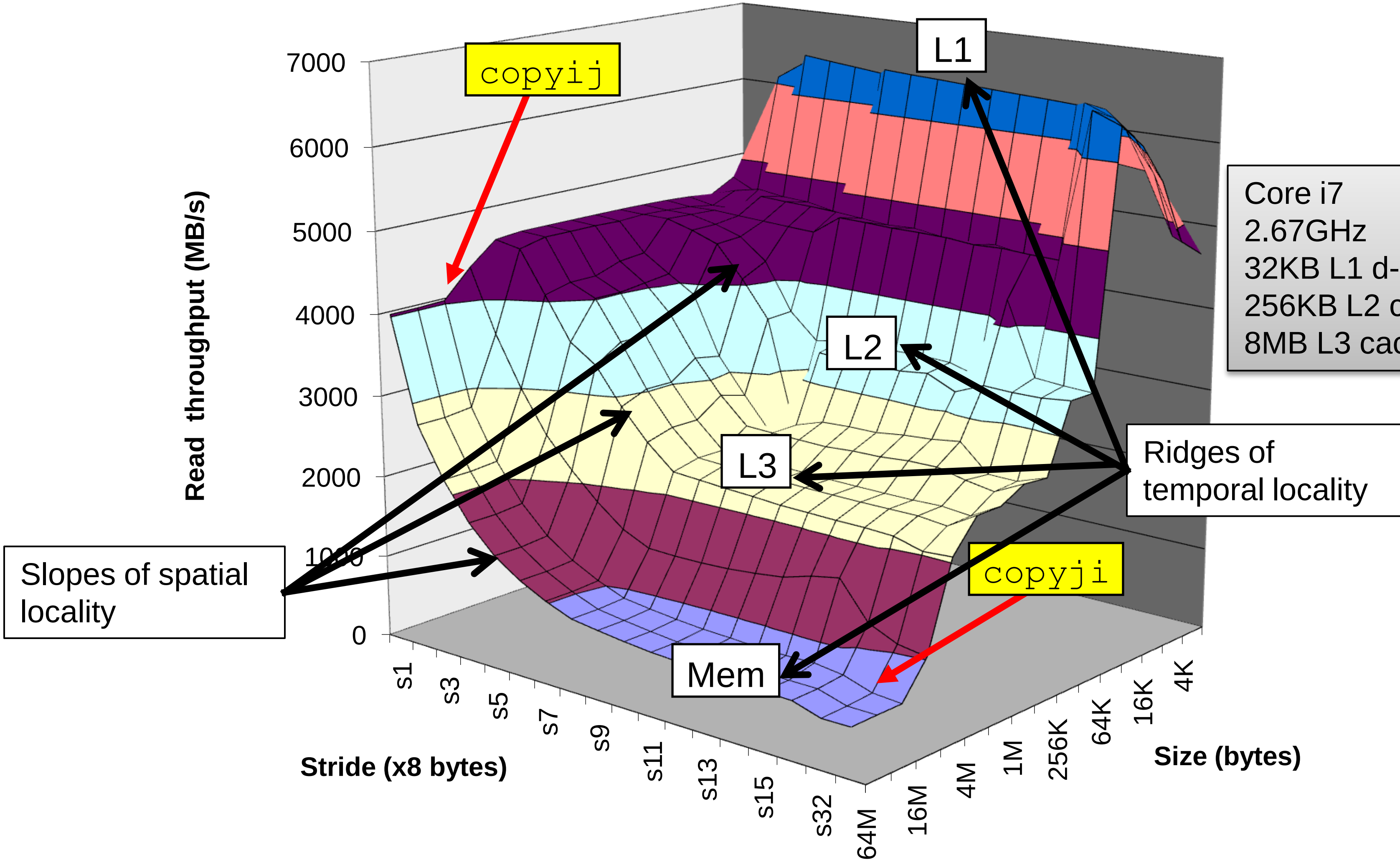
```
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS];         /* The array we'll be traversing */

int main()
{
    int size;        /* Working set size (in bytes) */
    int stride;      /* Stride (in array elements) */
    double Mhz;      /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0);              /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
         for (stride = 1; stride <= MAXSTRIDE; stride++)
             printf("%.1f\t", run(size, stride, Mhz));
         printf("\n");
    }
    exit(0);
}
```

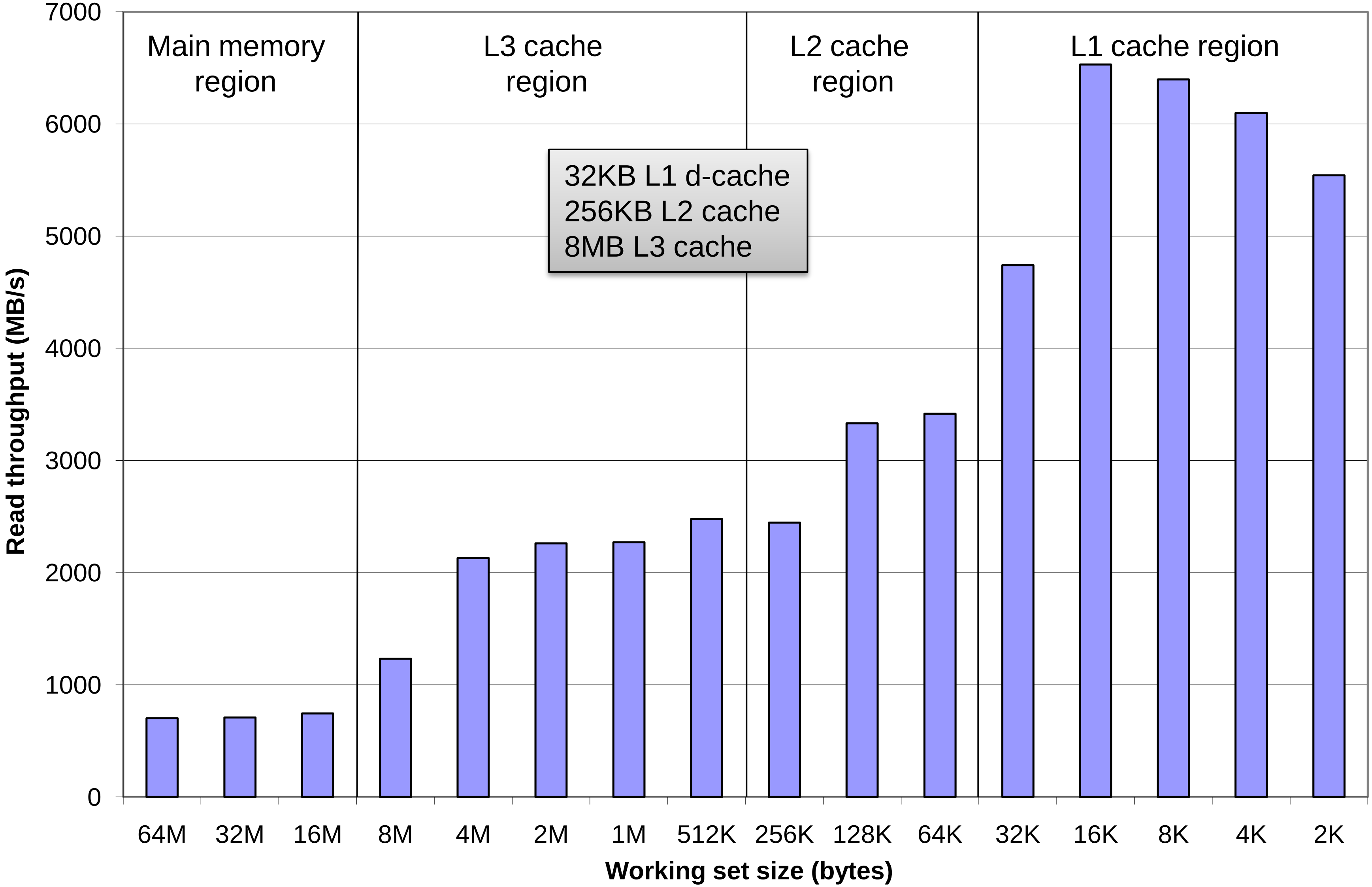
The memory mountain



Core i7
2.67GHz
32KB L1 d-cache
256KB L2 cache
8MB L3 cache

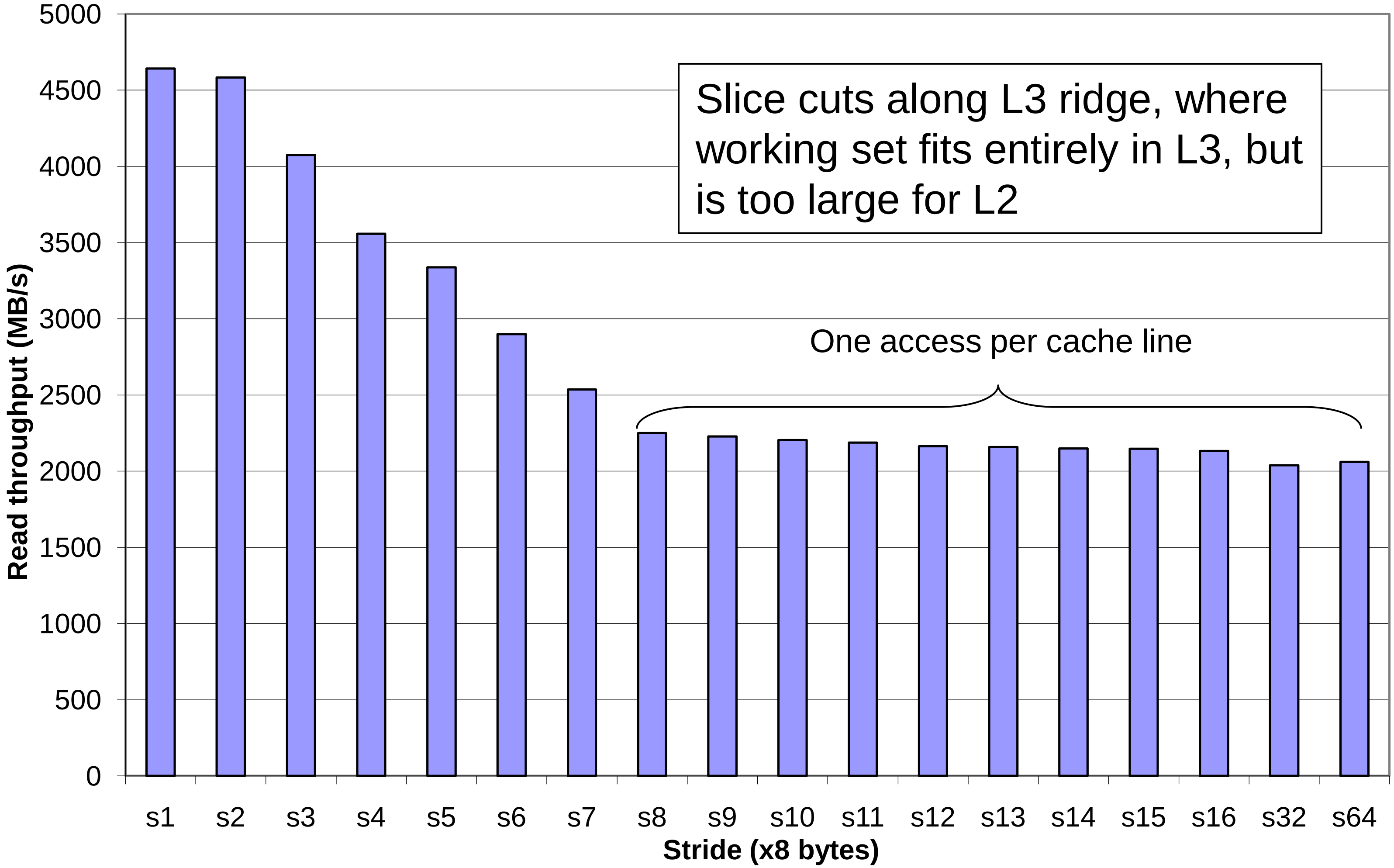
Ridges of temporal locality

- Slice through the memory mountain with stride=16



A slope of spatial locality

- Slice through memory mountain with size=4MB



Matrix multiplication example

- Major cache effects to consider
  - Total cache size
    - Exploit temporal locality and keep the working set small (e.g., by using blocking)
  - Block size
    - Exploit spatial locality

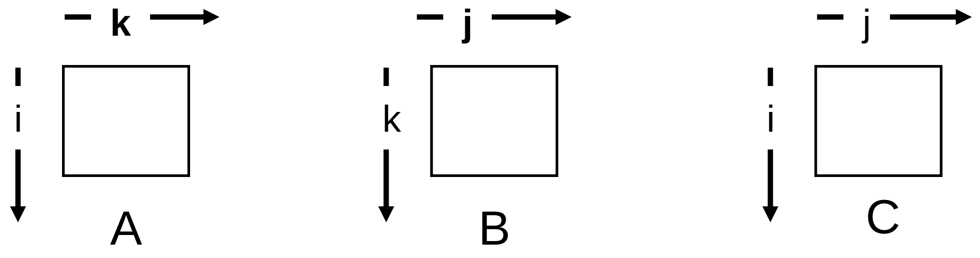
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

*Variable **sum** held in register*

- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

Miss rate analysis for matrix multiply

- Assume:
  - Line size = 32B (big enough for 4 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

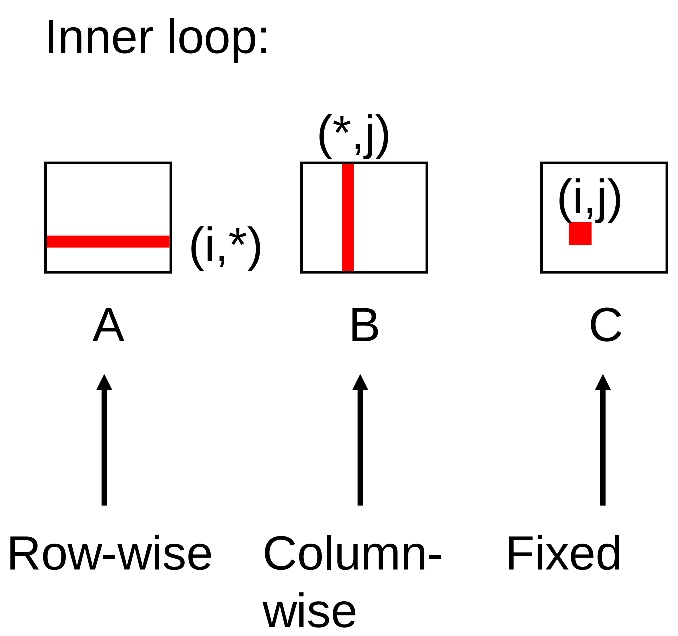


Layout of C arrays in memory (review)

- 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) > 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;
  }
}
```



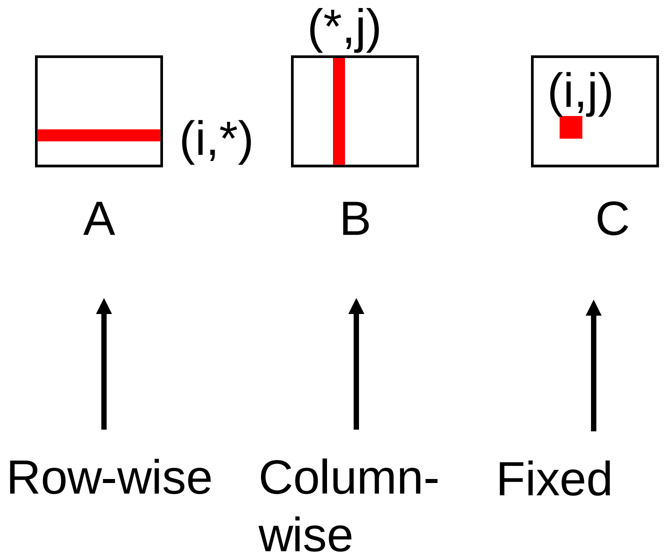
- Misses per inner loop iteration:

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
  }
}
```

Inner loop:

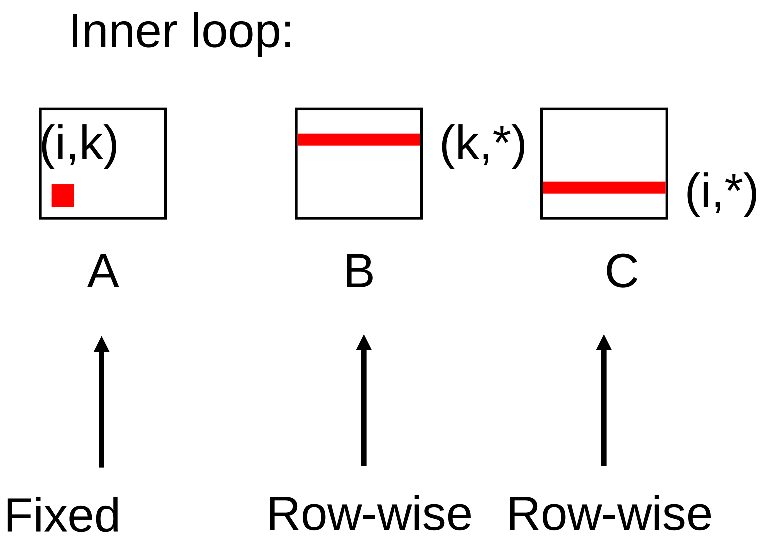


- Misses per inner loop iteration:

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];
  }
}
```

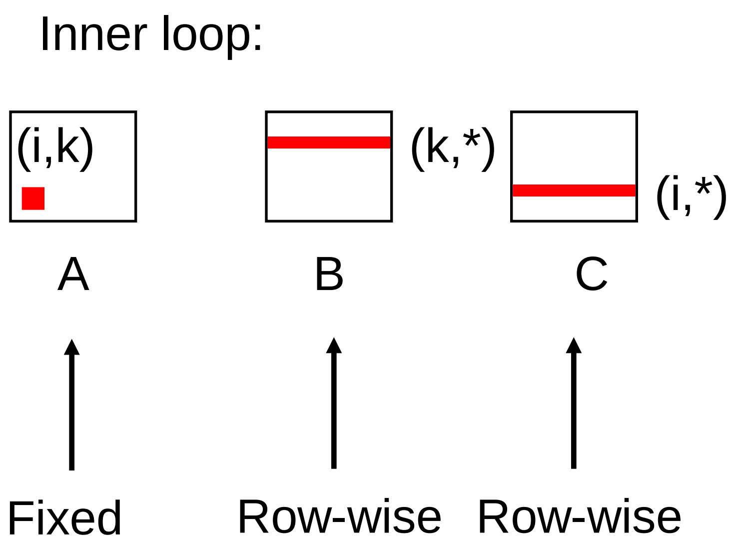


- Misses per inner loop iteration:

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];
  }
}
```

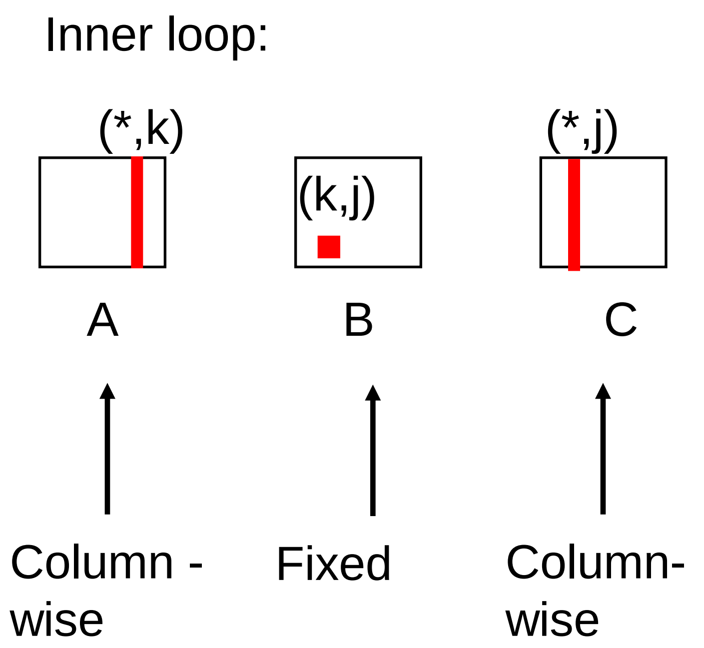


- Misses per inner loop iteration:

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;
  }
}
```



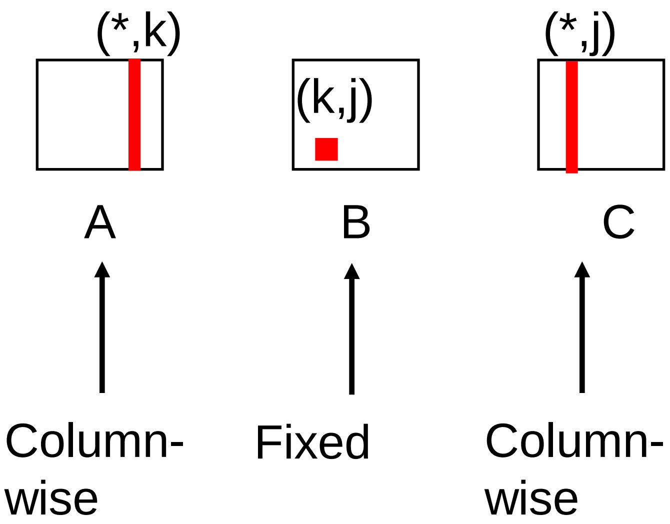
- <u>Misses per inner loop iteration:</u>

<u>A</u>	<u>B</u>	<u>C</u>
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;
  }
}
```

Inner loop:



- Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of matrix multiplication

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

```
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;
   }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

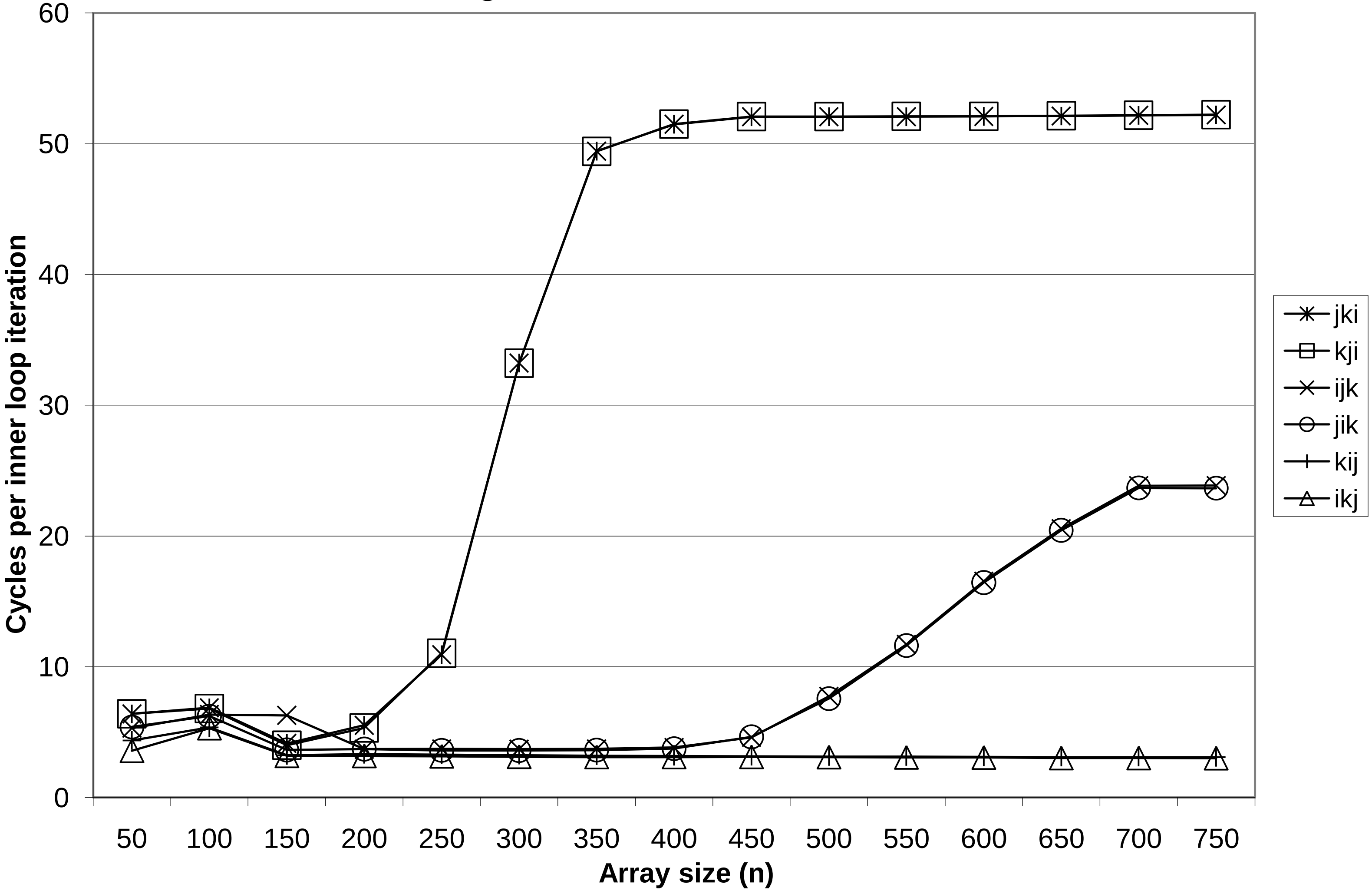
```
for (k=0; k<n; k++) {
   for (i=0; i<n; i++) {
      r = a[i][k];
      for (j=0; j<n; j++)
         c[i][j] += r * b[k][j];
   }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
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;
   }
}
```

Core i7 matrix multiply performance

- Miss rates are helpful but not perfect predictors.
  - Code scheduling matters, too.



Improving temporal locality by blocking

- Example: Blocked matrix multiplication
  - "block" (in this context) does not mean "cache block".
  - Instead, it mean a sub-block within the matrix.
  - Example: N = 8; sub-block size = 4

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \text{X} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

<u>Key idea:</u> Sub-blocks (i.e., $\mathbf{A_{xy}}$) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Blocked matrix multiply (bijk)

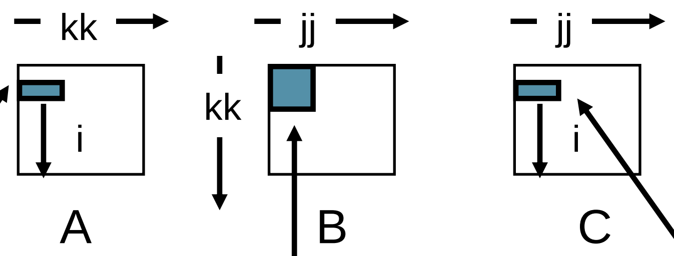
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
      }
    }
  }
}
```

Blocked matrix multiply analysis

- Innermost loop pair multiplies a *1 X bsize* sliver of *A* by a *bsize X bsize* block of *B* and accumulates into *1 X bsize* sliver of *C*
- Loop over *i* steps through *n* row slivers of *A* & *C*, using same *B*

```
for (i=0; i<n; i++) {
  for (j=jj; j < min(jj+bsize,n); j++) {
    sum = 0.0
    for (k=kk; k < min(kk+bsize,n); k++) {
      sum += a[i][k] * b[k][j];
    }
    c[i][j] += sum;
  }
```

Innermost Loop Pair



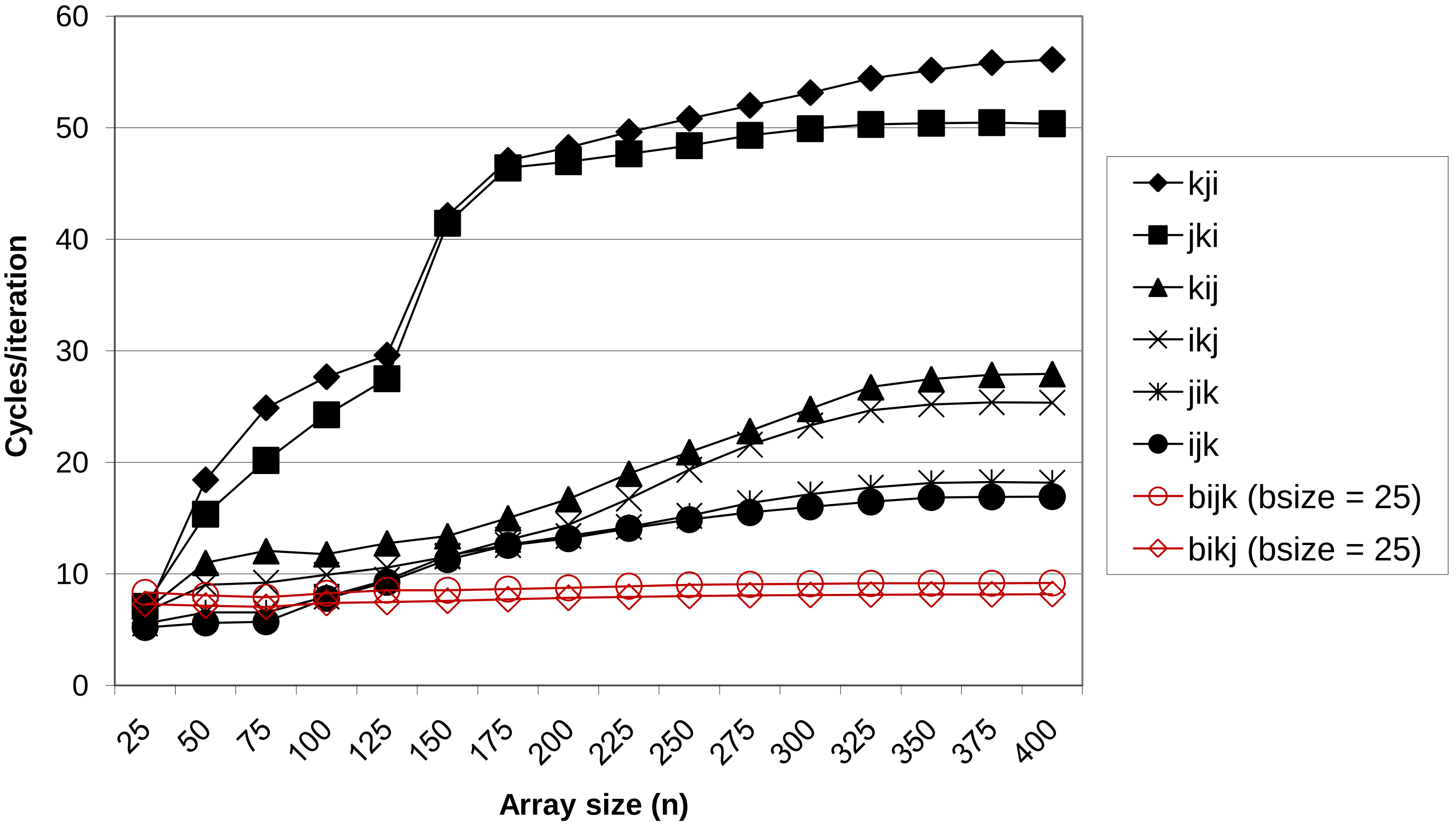
row sliver accessed ***bsize*** times

block reused ***n*** times in succession

Update successive elements of sliver

Pentium blocked matrix mult performance

- Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)
  - relatively insensitive to array size.



Concluding observations

- 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)