Cache Memories

Topics

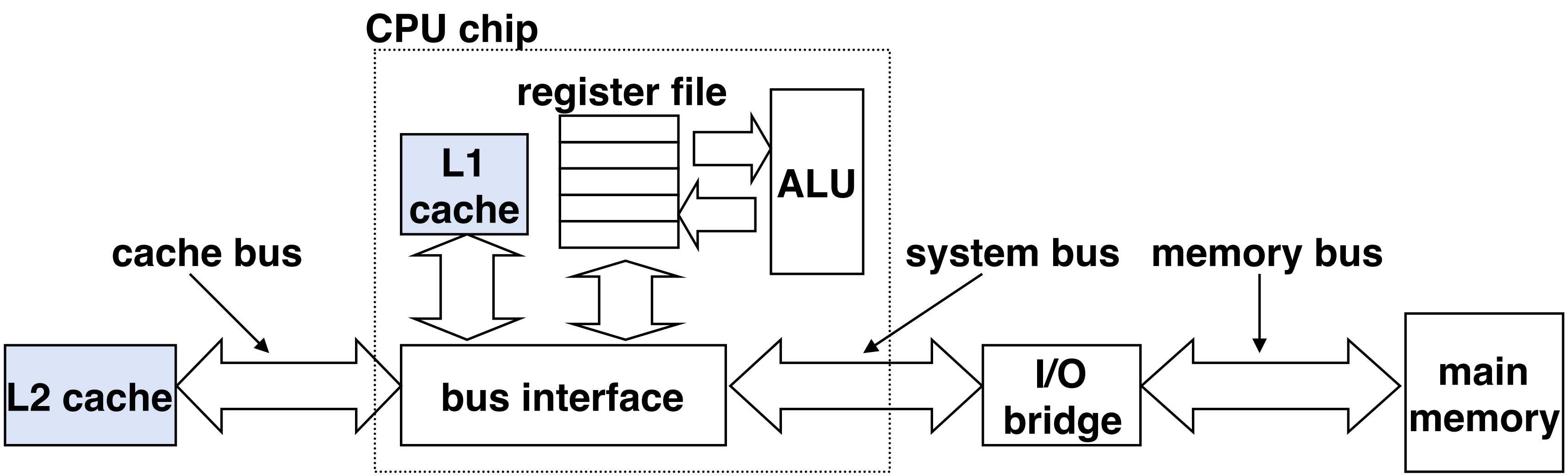
- Generic cache memory organization
- Direct mapped caches
- Set associative caches
- Impact of caches on performance

Next time

- Dynamic memory allocation and memory bugs

Cache memories

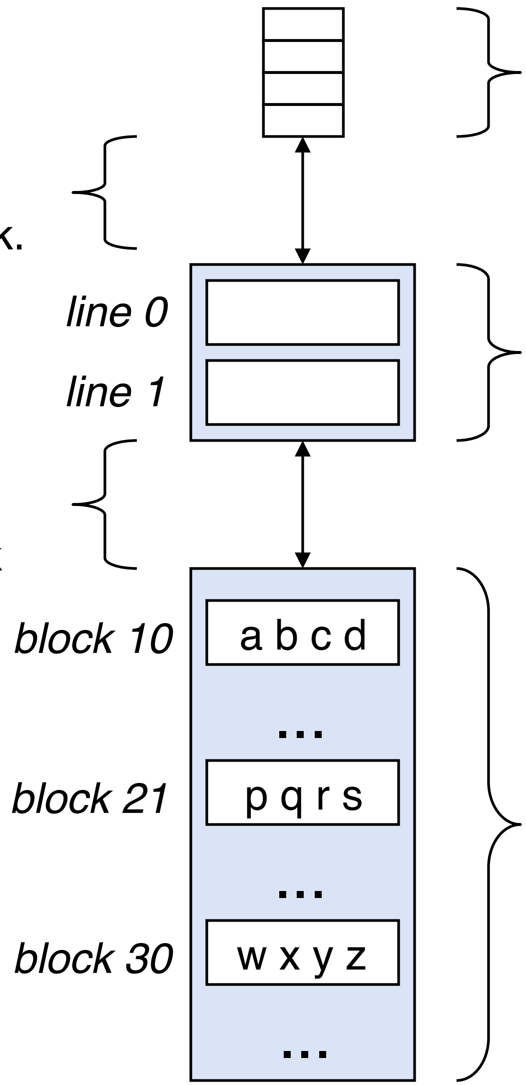
- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  – Hold frequently accessed blocks of main memory
- CPU looks first for data in L1, then in L2, …, then in main memory.
- Typical bus structure:



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.

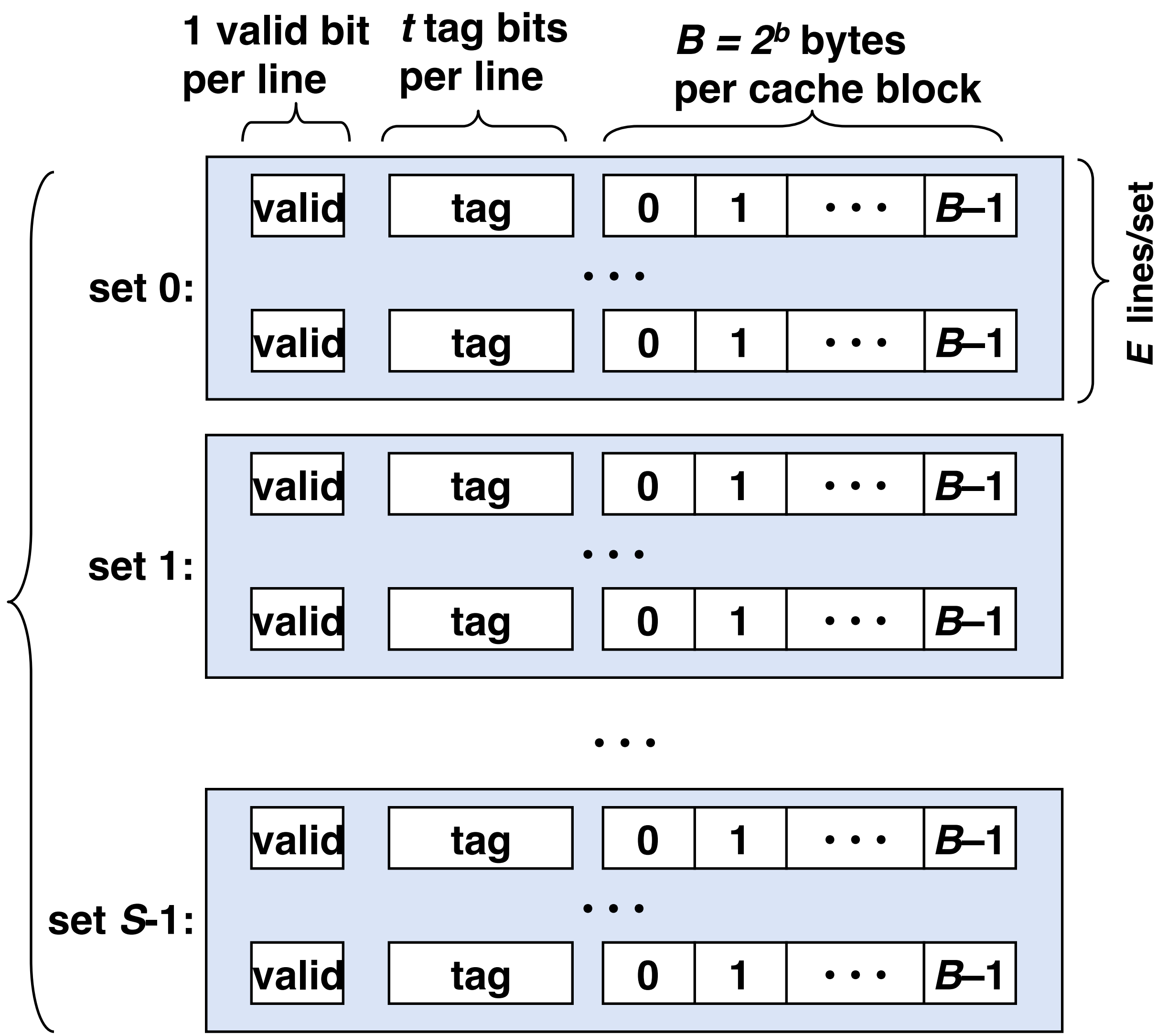
General organization of a cache memory

Memory address: *m* bits

Cache: $S = 2^s$ sets

Set: *E* lines

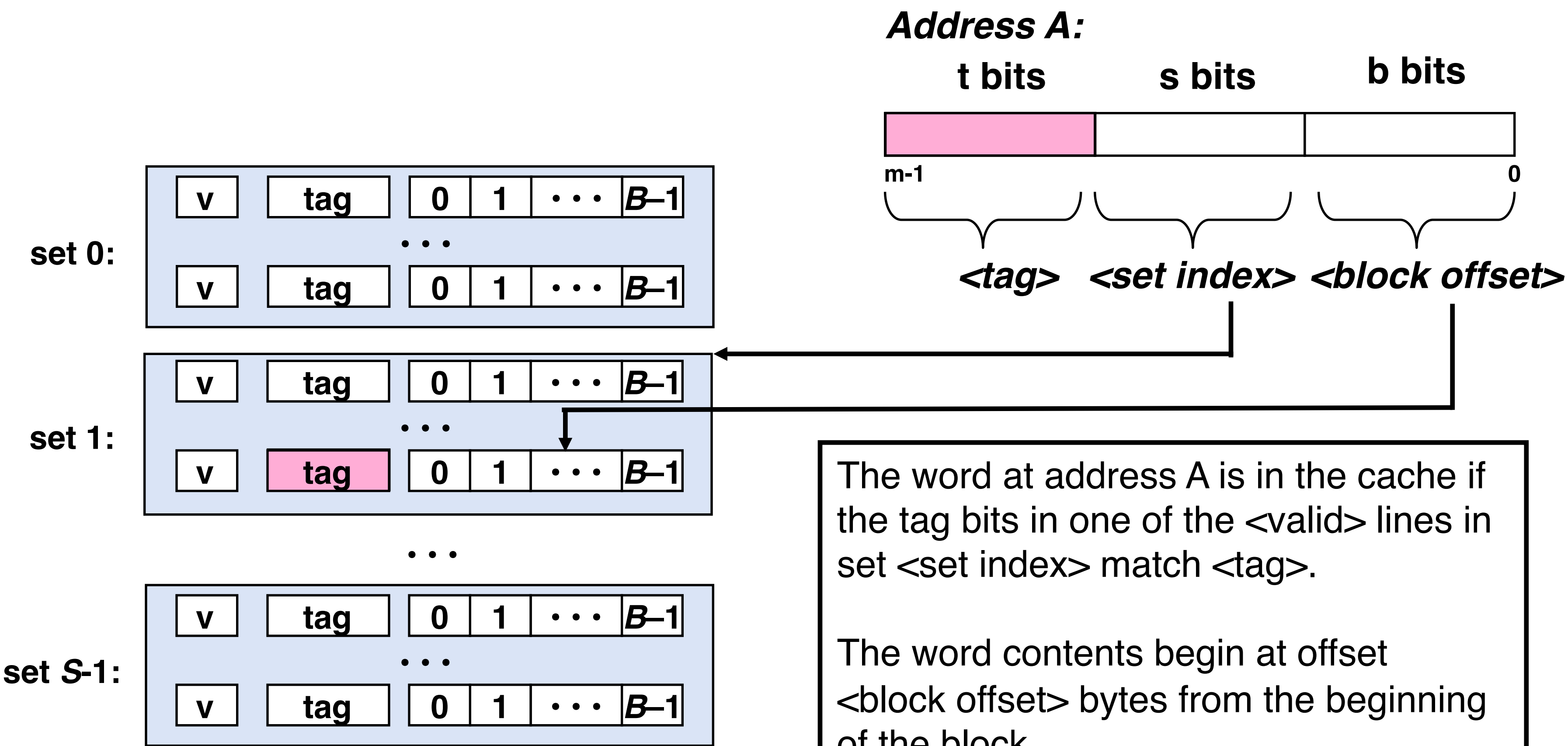
Line holds data block (size *B*)



Cache's organization characterized by (S, E, B, m)

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

Addressing caches

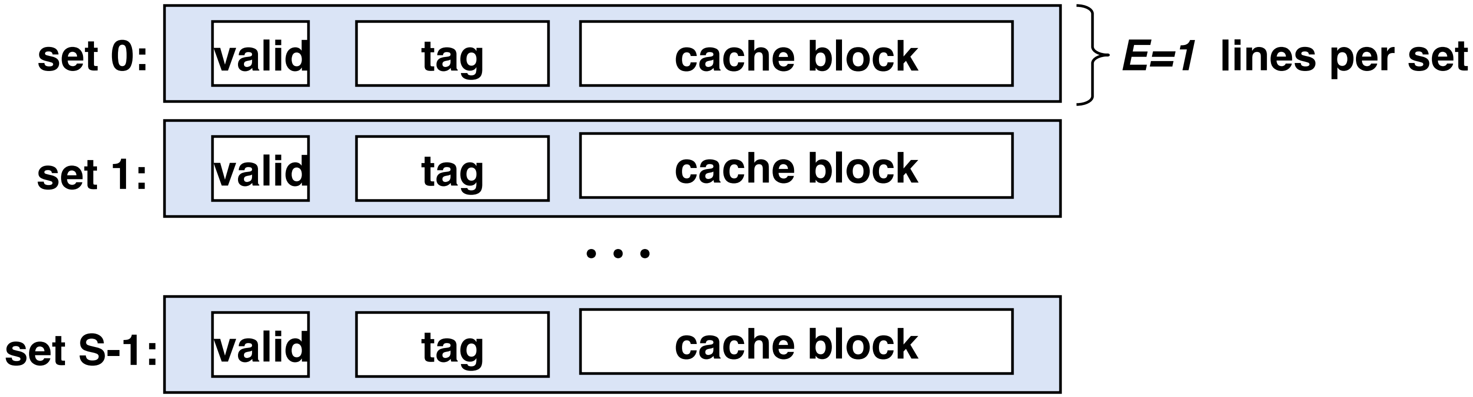


The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

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

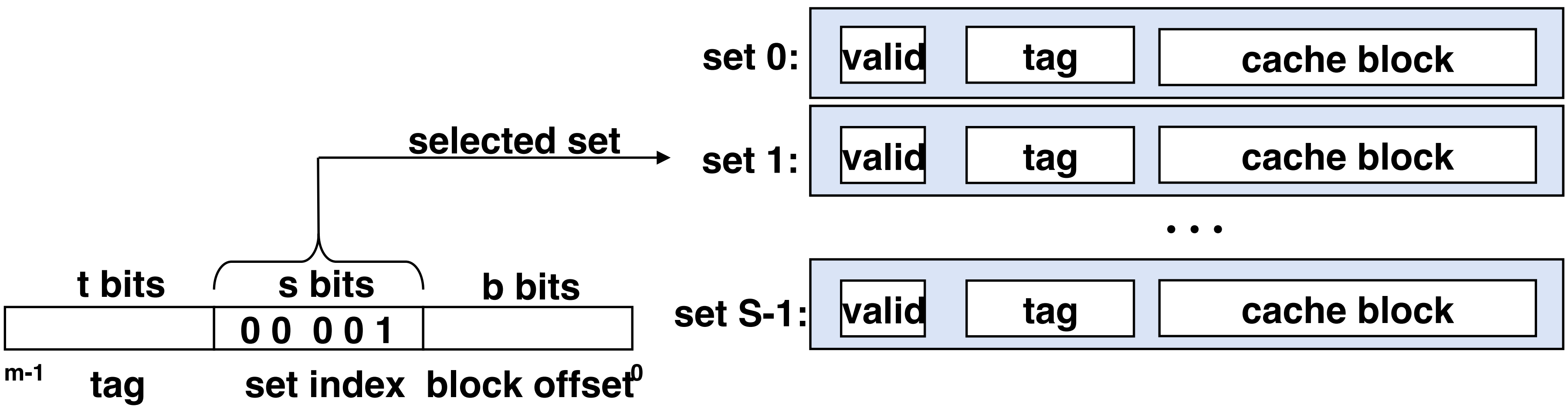
Direct-mapped cache

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



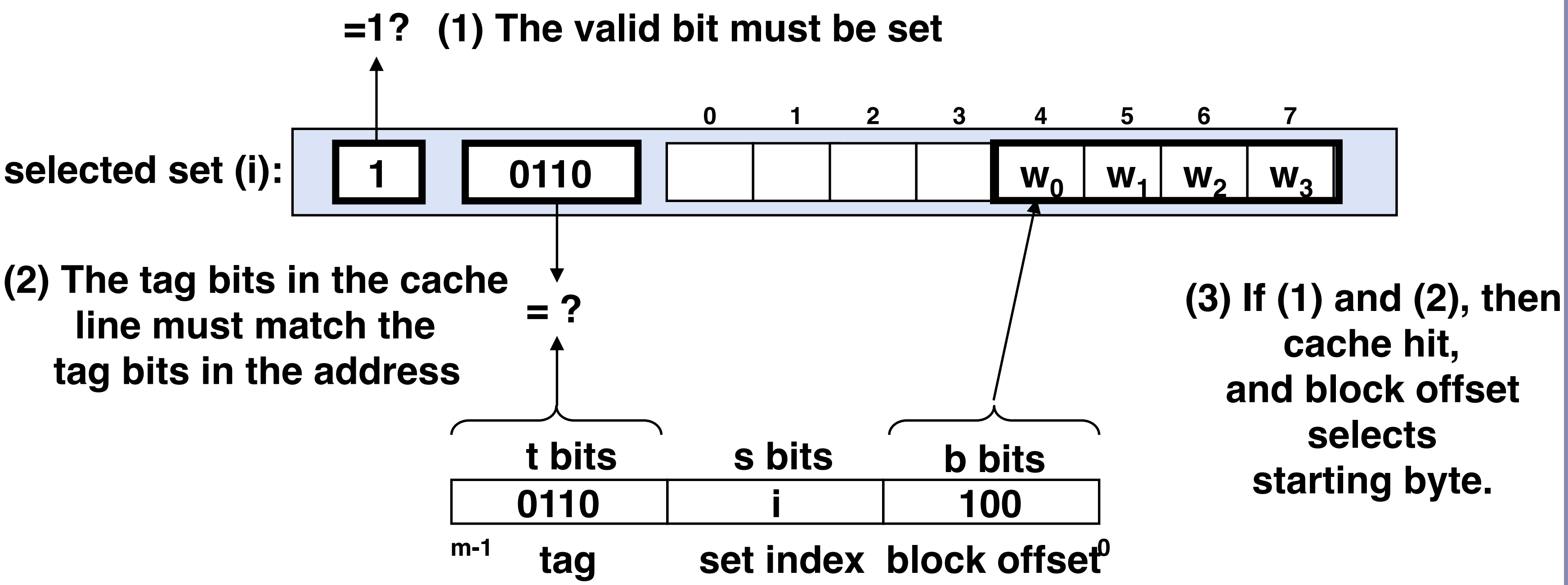
Accessing direct-mapped caches

- 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

Tag + index uniquely identifies each block

Multiple blocks map to the same cache set (0 & 4 to 0, 1 & 5 to 1)

And you can tell them apart by the tag

Direct-mapped cache simulation

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

0 [0000_2] *(miss)*

v	tag	block[0]	block[1]
1	0	m[0]	m[1]

1 [0001_2] *(hit)*

13 [1101_2] *(miss)*

v	tag	block[0]	block[1]
1	0	m[0]	m[1]
1	0	m[12]	m[13]

8 [1000_2] *(miss)*

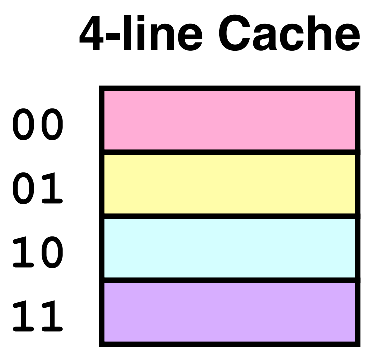
v	tag	block[0]	block[1]
1	0	m[8]	m[9]
1	0	m[12]	m[13]

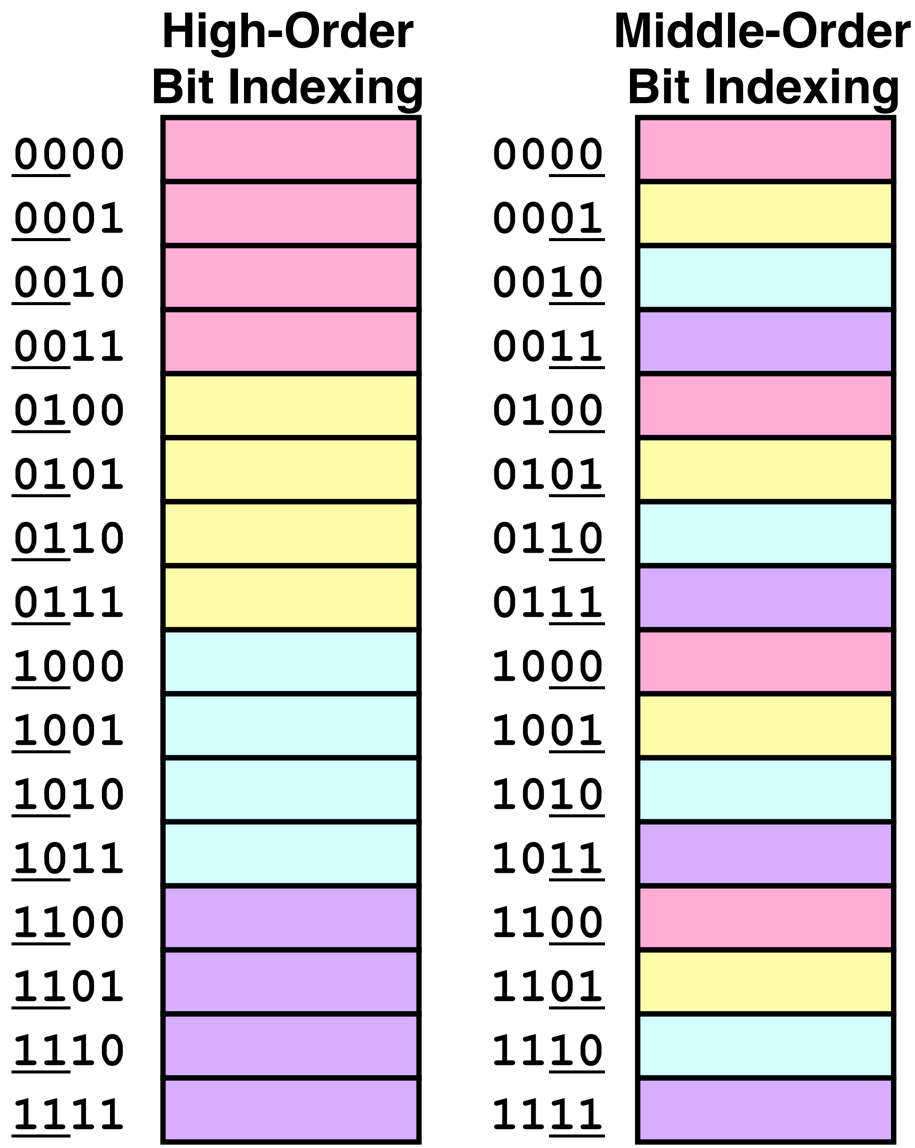
0 [0000_2] *(miss)*

A conflict miss

v	tag	block[0]	block[1]
1	0	m[0]	m[1]
1	0	m[12]	m[13]

Why use middle bits as index?

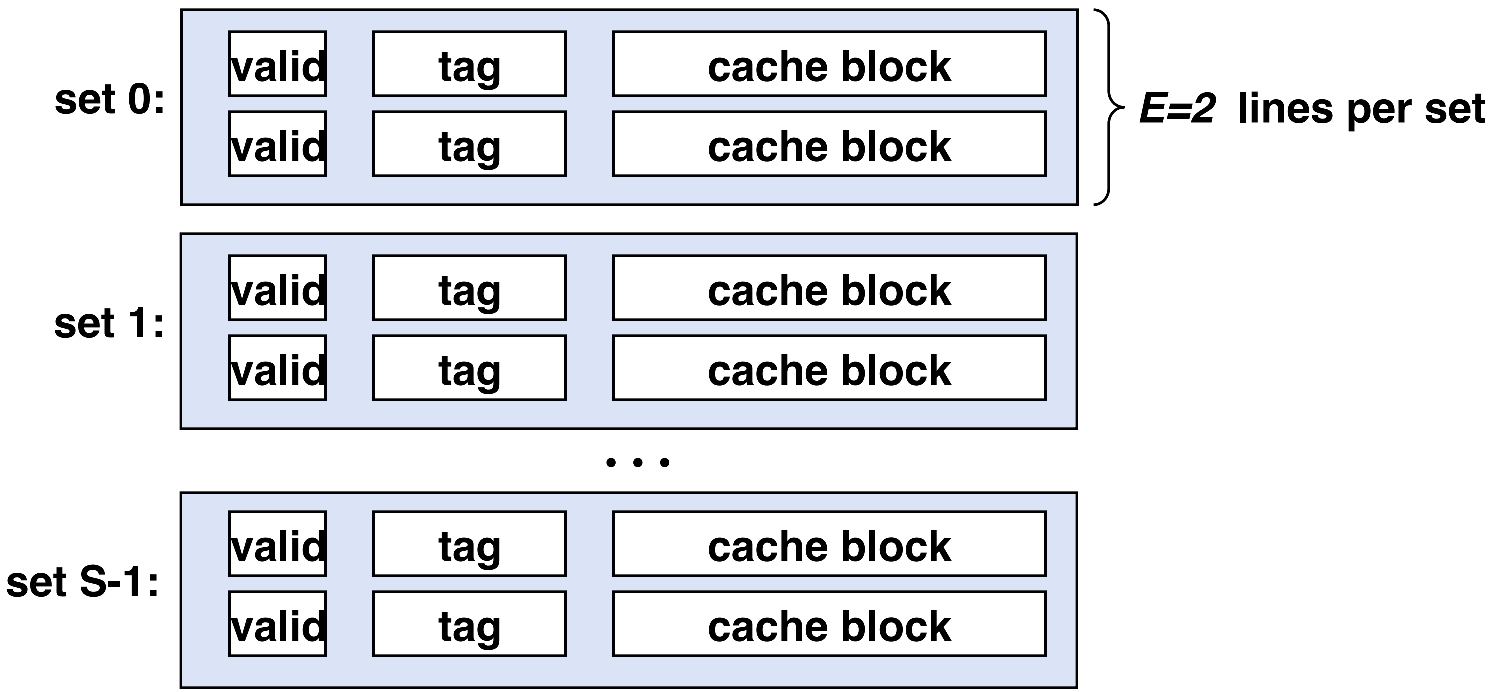




- High-order bit indexing
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality
- Middle-order bit indexing
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time

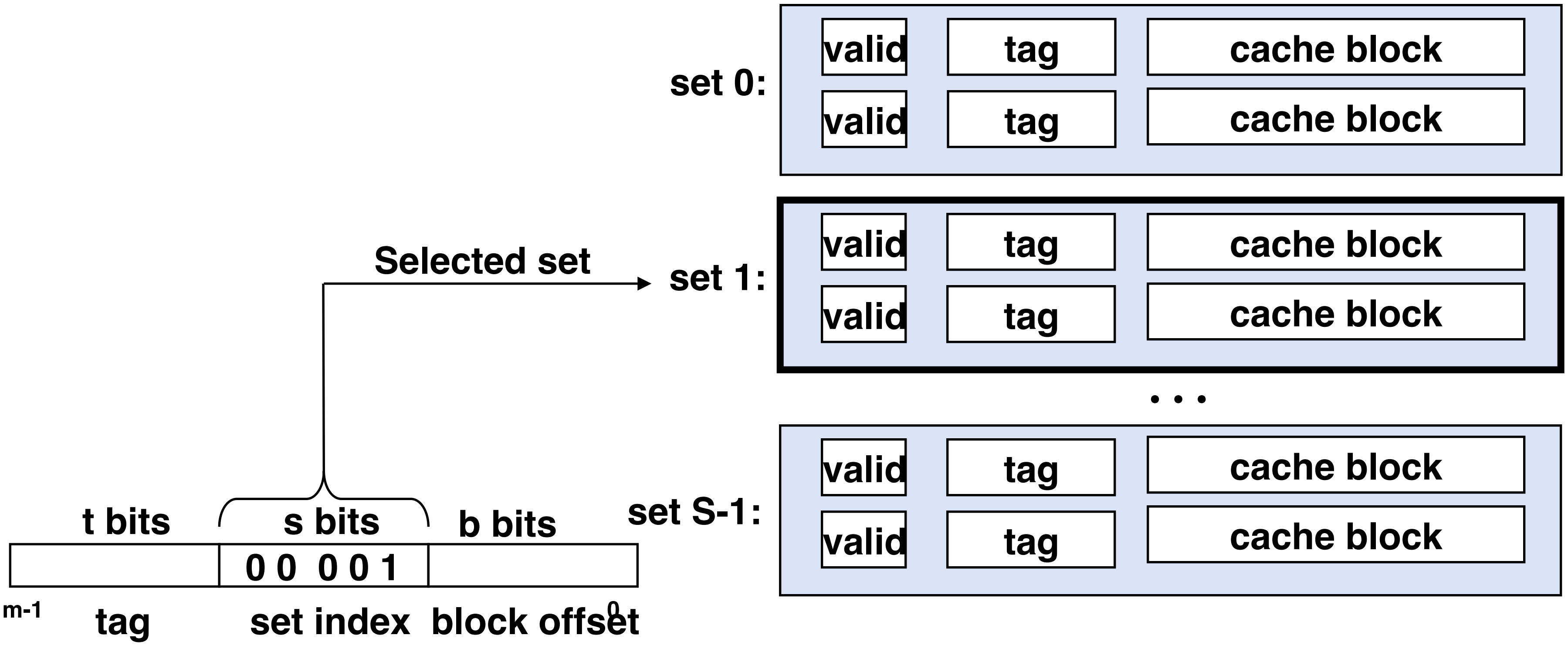
Set associative caches

- In direct mapped caches, since every set as exactly one line – conflict misses
- Set associative cache – >1 line per set ($1 < E < C/B$)
  - *E-way associative*



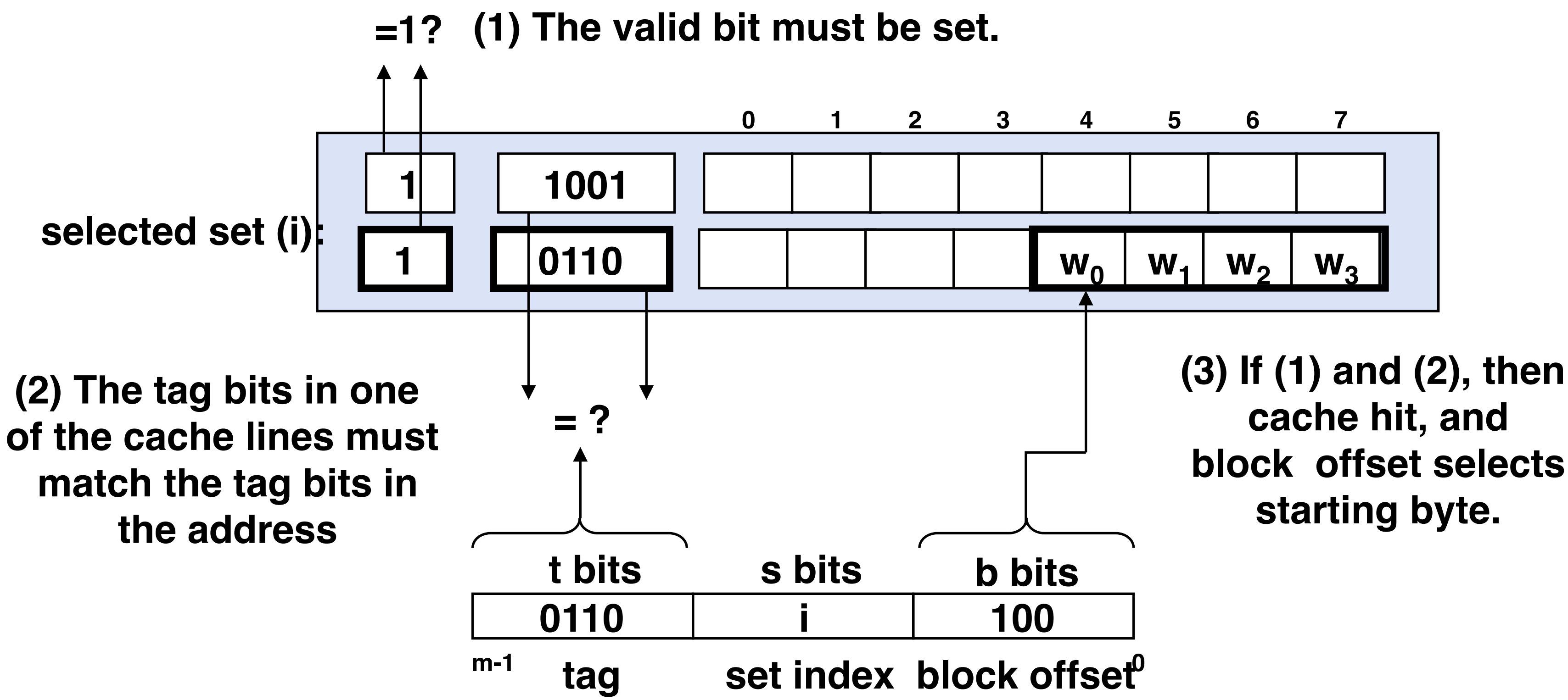
Accessing set associative caches

- Set selection
  - identical to direct-mapped cache



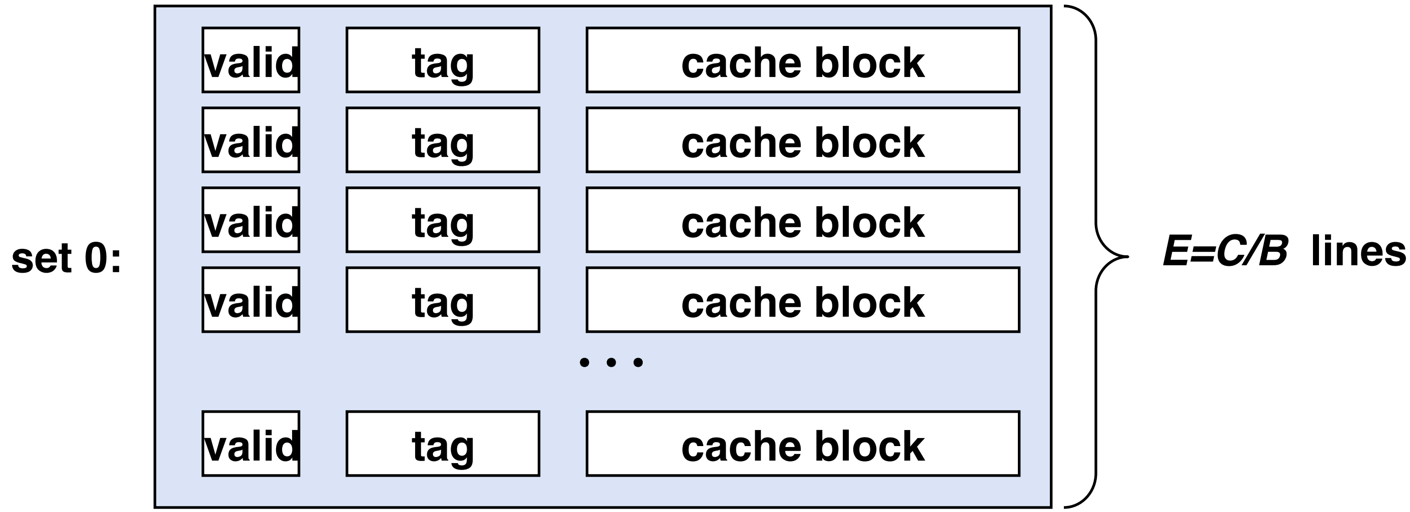
Accessing set associative caches

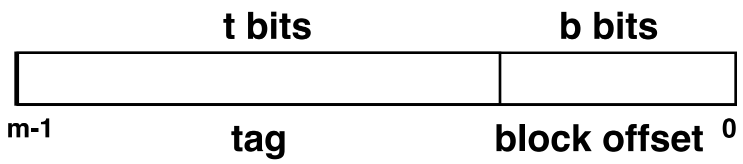
- Line matching and word selection
  - must compare the tag in each valid line in the selected set.



Fully associative caches

- A single set with all the cache lines ($E = C/B$)
  - Set selection is trivial, only one set
  - Line matching and word selection – same as with set associative
  - Pricy so typically use for small caches like TLBs





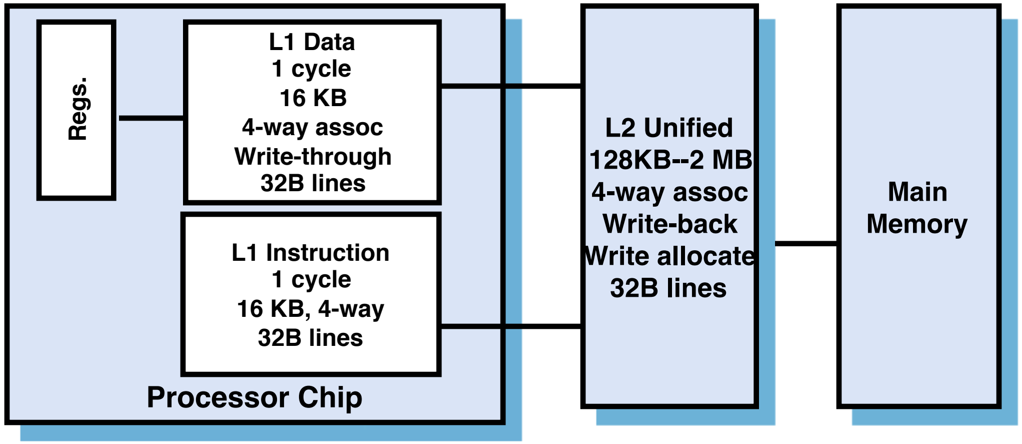
The issues with writes

- So far, all examples have used reads – simple
  - Look for a copy of the desired word, if hit, return
  - Else, fetch block from next level, cache it, return word
- For writes – a bit more complicated
  - If there's a hit, what to do after updating the cache copy?
    - Write it to next level? *Write-through*; simple but expensive
    - Defer update? *Write-back*; write when the block is evicted, faster but more complex (need a dirty bit)

The issues with writes

- For writes – a bit more complicated
  - …
  - If there's a miss, bring it to cache or write through?
    - *Write-allocate* – Bring the block to cache and update; leverage spatial locality but a block transfer per write miss
    - *No-write-allocate* – Write through bypassing the cache
  - Write through caches are typically no-write-allocate
  - *As logic density increases, write-back's complexity is less of an issue and performance is a plus*

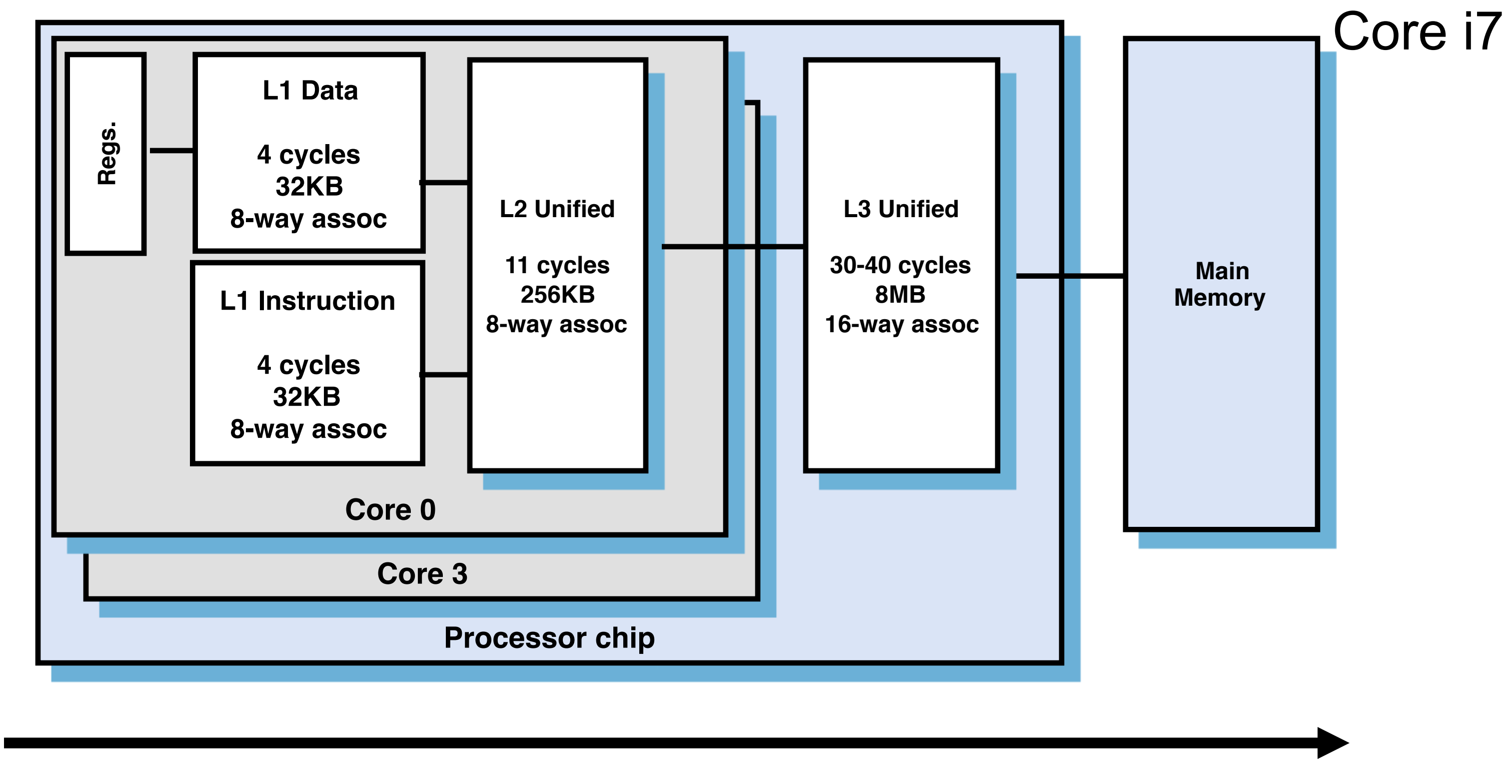
Real Cache Hierarchies



Pentium III Xeon

- Caches can be for anything (unified) or specialized for data/instruction (d-cache & i-cache); why specialized?
  - Processor can read both at the same time
  - i-caches are typically read-only, simpler, and with different access patterns
  - Data and instruction access can’t create conflict with each other

Real Cache Hierarchies



larger,
slower,
cheaper

Size:	32KB	256KB	8MB
E:	8-way	8-way	16-way
Access:	4 cycles	11 cycles	30-40 cycles

Cache performance metrics

- Miss Rate
  - Fraction of memory references not found in cache
  - 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:
    - 1-2 clock cycle for L1, 5-20 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - Typically 50-200 cycles for main memory (increasing)

Cache performance metrics

- Big difference between a hit and a miss
  - 100x if you only have L1 and main memory
- *A 99% hit rate is twice as good as 97% rate?*
  - Consider
    - Cache hit time 1 cycle
    - Miss penalty 100 cycles
  - Average access time
    - 97% hit rate: 0.97 * 1 cycle + 0.03 * (1+100 cycles) = 1 cycle + 0.03 * 100 cycles = 4 cycles
    - 99% hit rate: 0.99 * 1 cycle + 0.01* (1+100 cycles) = 1 cycle + 0.01 * 100 cycles = 2 cycles

Writing cache-friendly code

- Programs with better locality will tend to have lower miss rates and run faster
- Basic approach to cache friendly code
  - Make the common case go fast – core loops in core functions
  - Minimize the number of cache misses in each inner loop – all other things being equal, better miss rates means faster runs
- Example
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

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;
}
```

Miss rate = 1/4 = 25%

```
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 = 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

```
/* 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];

  /* So compiler doesn't optimize
     away the loop */
  sink = result;
}
```

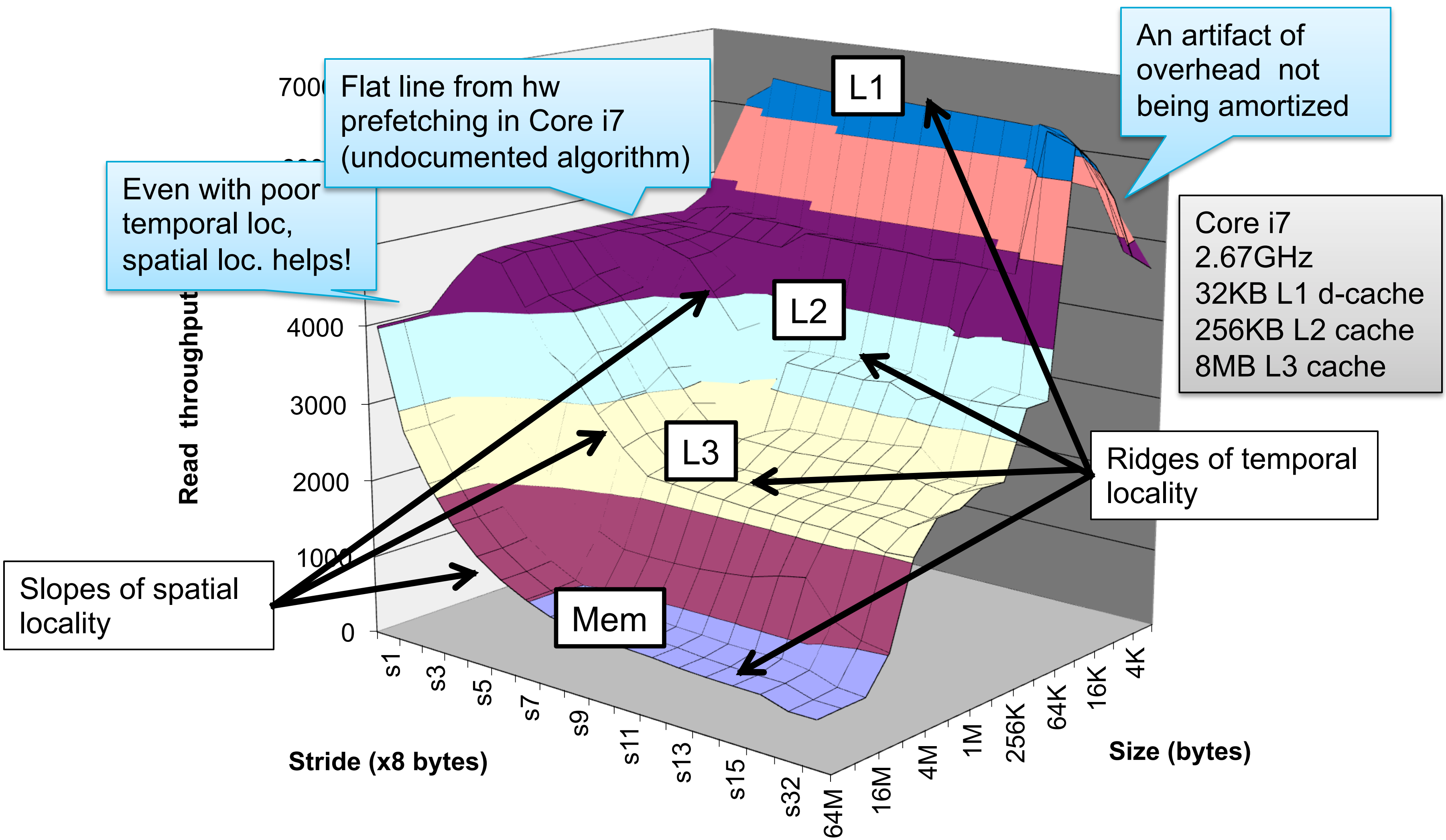
```
/* Run test(elems, stride) and return
   read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
   double cycles;
   int elems = size / sizeof(int);

   /* warm up the cache */
   test(elems, stride);

   /* call test(elems,stride) */
   cycles = fcyc2(test, elems, stride, 0);

   /* convert cycles to MB/s */
   return (size / stride) / (cycles / Mhz);
}
```

The memory mountain for Intel Core i7



Rearranging loops to improve locality

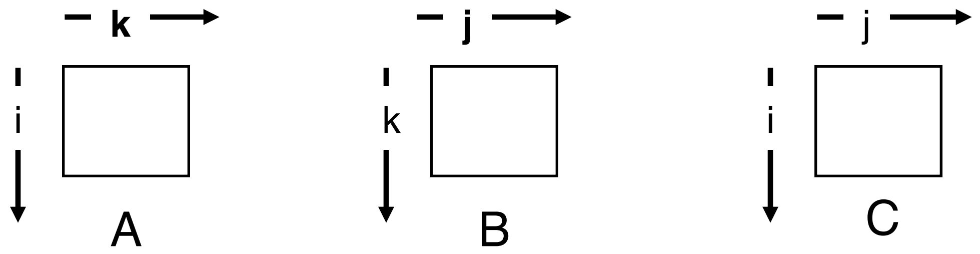
- Matrix multiply
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

```
/* 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*

Miss rate analysis for matrix multiply

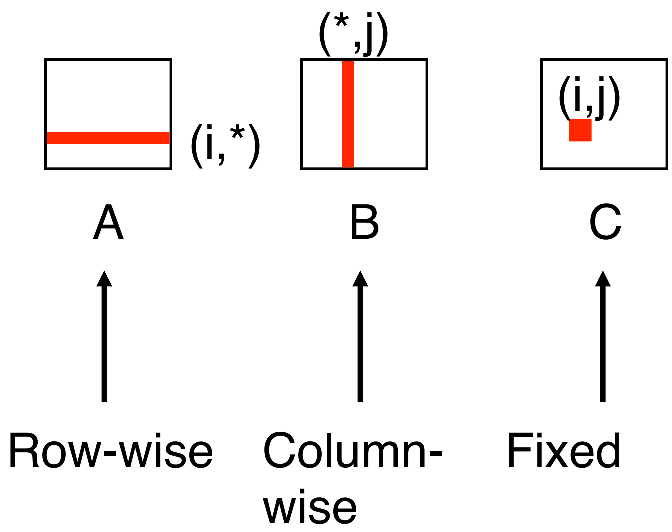
- Assume:
  - Line size = 32B (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - A single matrix row does not fit in L1
  - Compiler stores local variables in registers
- Analysis method:
  - Look at access pattern of inner loop



Matrix multiplication (ijk)

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

Inner loop:



Per iteration

Loads	Stores	A misses	B misses	C misses	Total misses
2	0	0.25	1.00	0.00	1.25

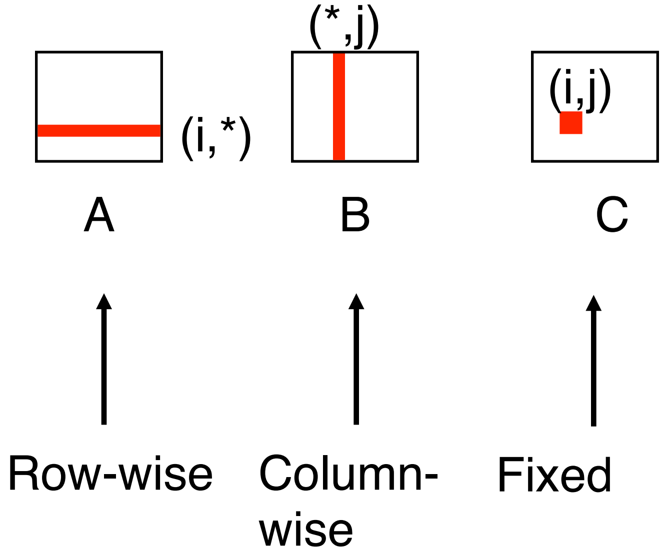
Each cache block holds 4 elements (doublewords)

But it scans B with a stride of *n*

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:



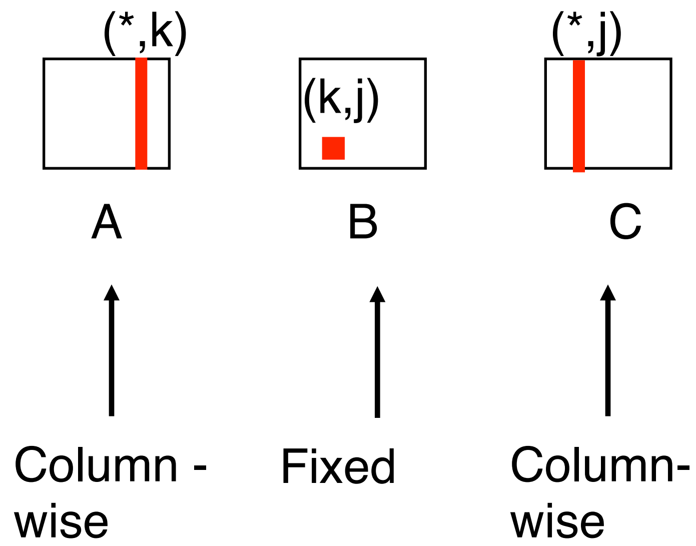
Per iteration

Loads	Stores	A misses	B misses	C misses	Total misses
2	0	0.25	1.00	0.00	1.25

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;
  }
}
```

Inner loop:



Per iteration

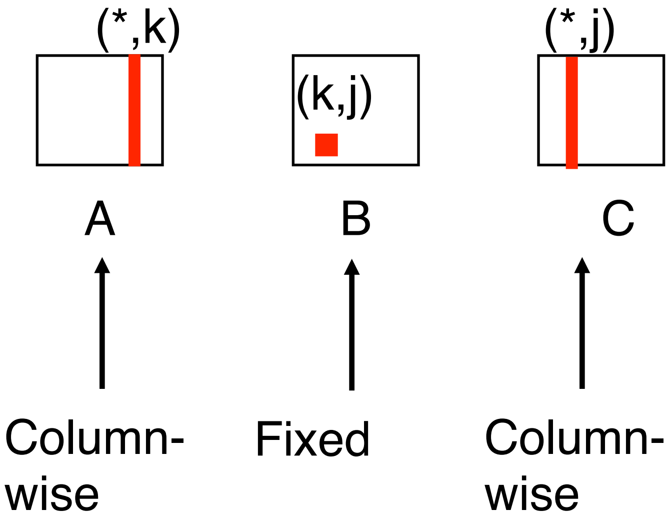
Loads	Stores	A misses	B misses	C misses	Total misses
2	1	1.00	0.00	1.00	2.00

Scan A and C with stride of *n;* a miss on each iteration; that plus 1 more memory op!

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:

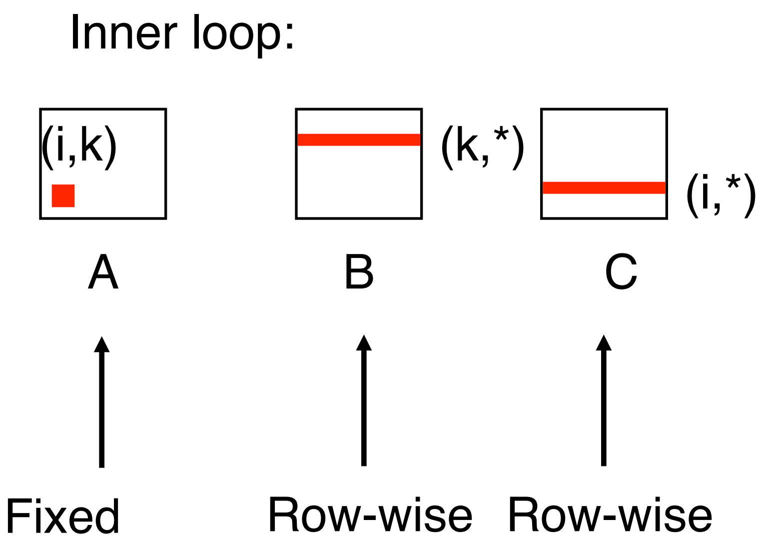


Per iteration

Loads	Stores	A misses	B misses	C misses	Total misses
2	1	1.00	0.00	1.00	2.00

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];
  }
}
```



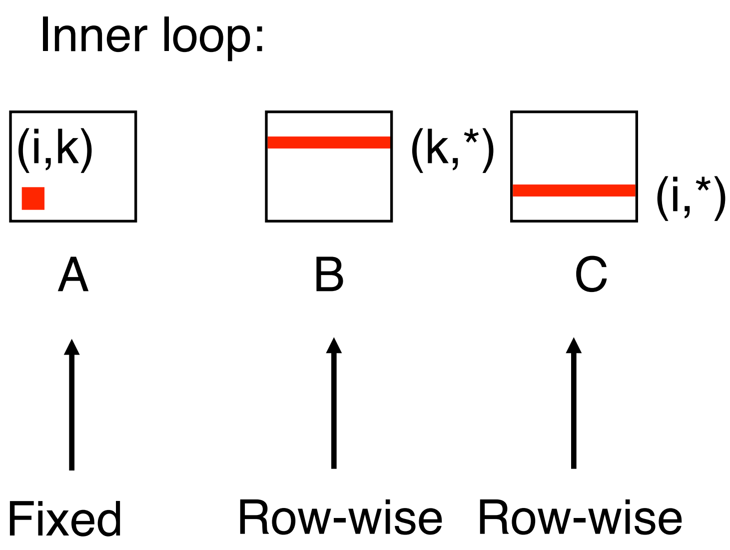
Per iteration

Loads	Stores	A misses	B misses	C misses	Total misses
2	1	0.00	0.25	0.25	0.50

An interesting trade-off; one more memory operation for fewer misses

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];
  }
}
```



Per iteration

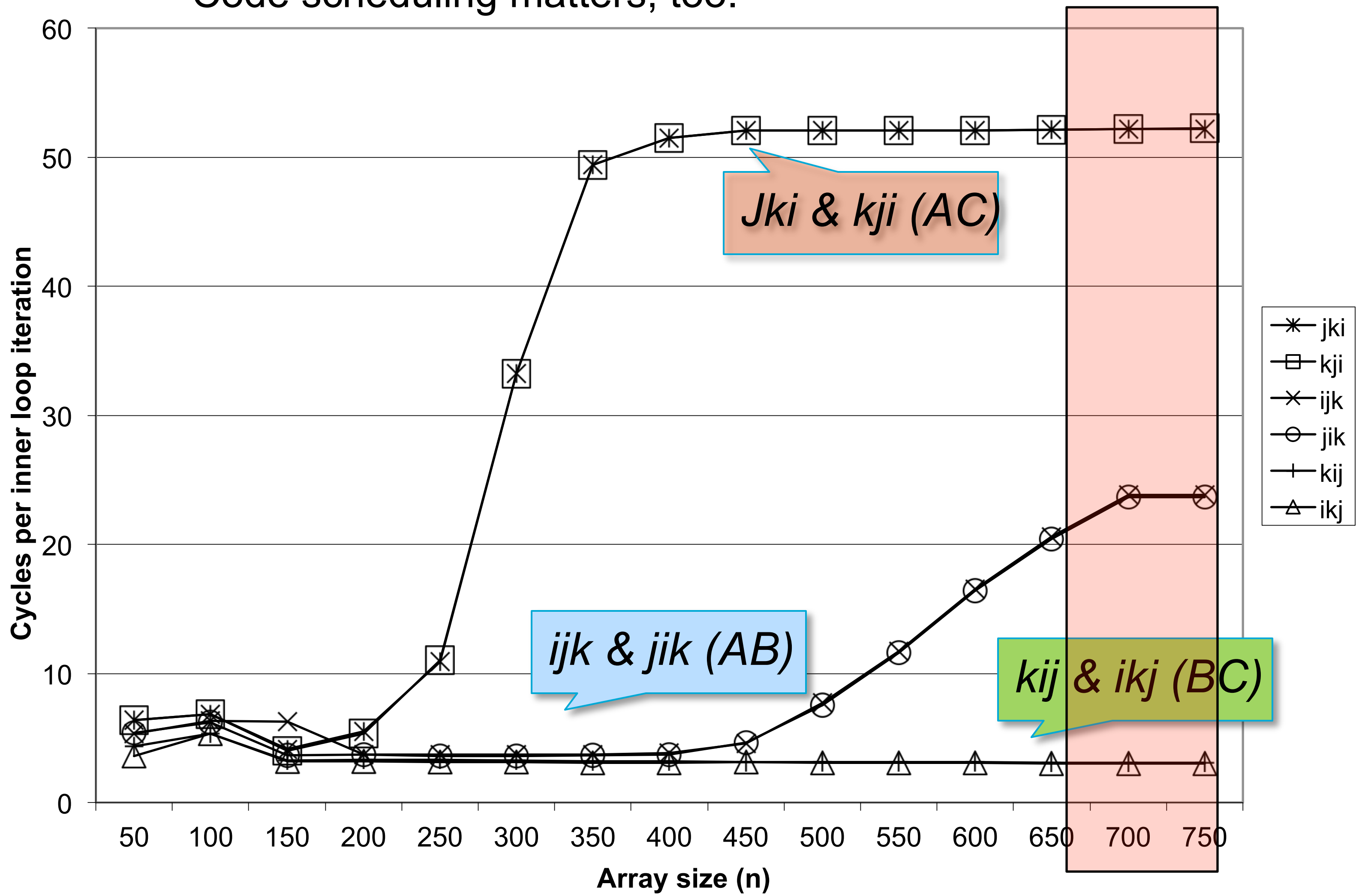
Loads	Stores	A misses	B misses	C misses	Total misses
2	1	0.00	0.25	0.25	0.50

Summary of matrix multiplication

Matrix multiply class	Loads	Stores	A misses	B misses	C misses	Total misses
ijk & jik (AB)	2	0	0.25	1.00	0.00	1.25
jki & kji (AC)	2	1	1.00	0.00	1.00	2.00
kij & ikj (BC)	2	1	0.00	0.25	0.25	0.50

Core i7 matrix multiply performance

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



Concluding observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - You can try to help with blocking, but that’s better left to libraries and compilers
- 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)