Cache Memories

Topics

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

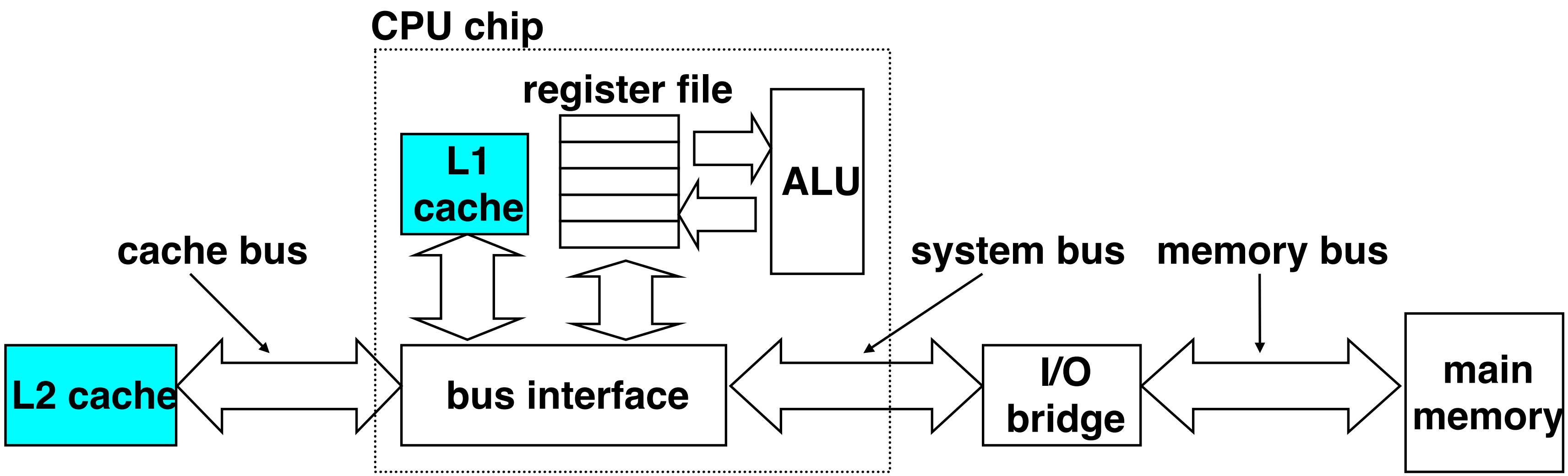
Next time

- Linking

Chris Riesbeck, Spring 2010
Original: Fabian Bustamante

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:



Measuring Cache Effects

- Memory mountain test code
  - Measures read throughput as a function of spatial and temporal locality.
  - Read throughput (read bandwidth) = Number of bytes read from memory per second (MB/s)
  - Graph throughput over changes in stride and working set size (number of repeatedly referenced locations)
  - Compact way to characterize memory system performance.

Memory mountain main routine

```
/* 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);
}
```

Memory mountain test function

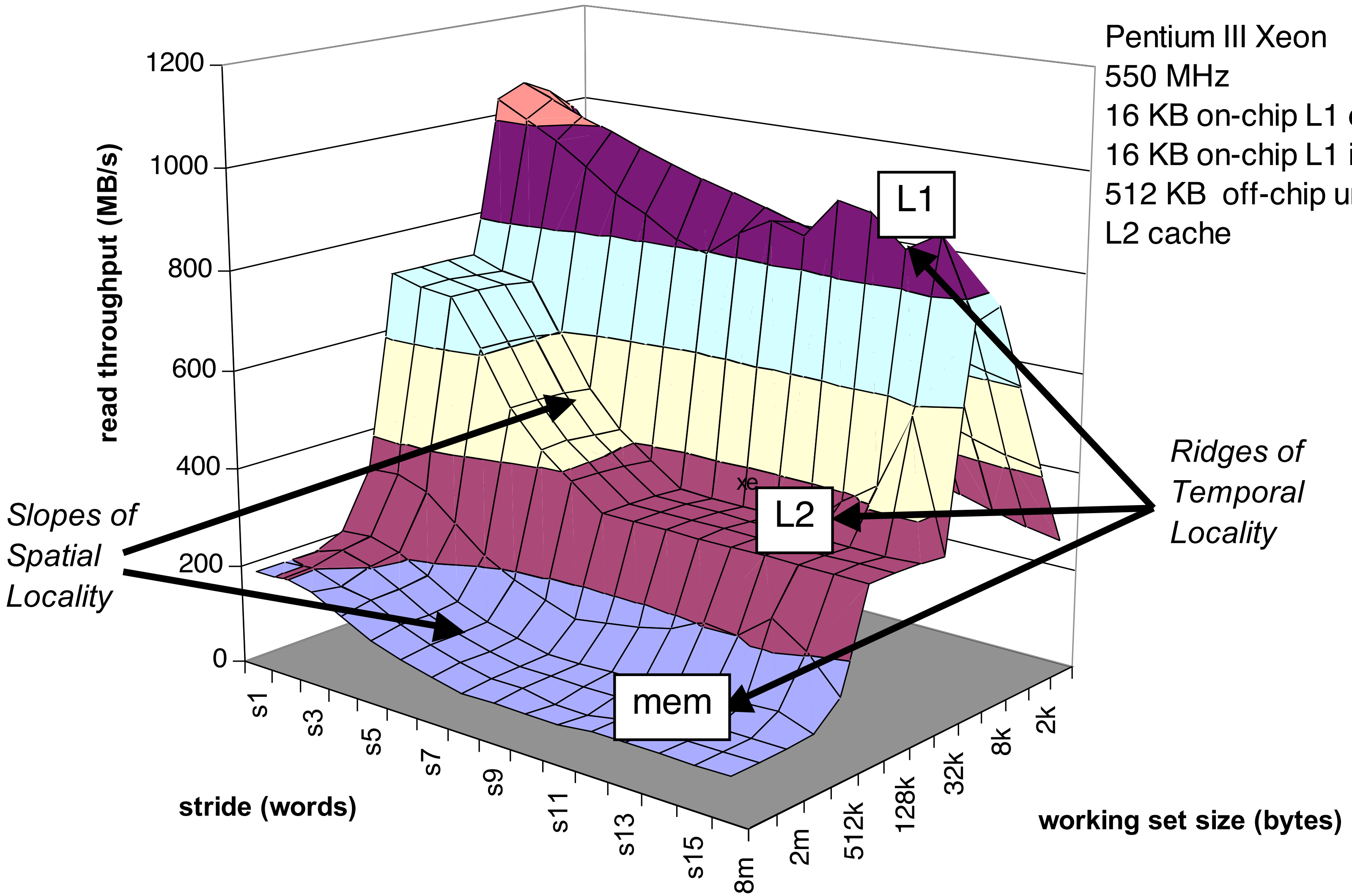
```
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    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 */
}
```

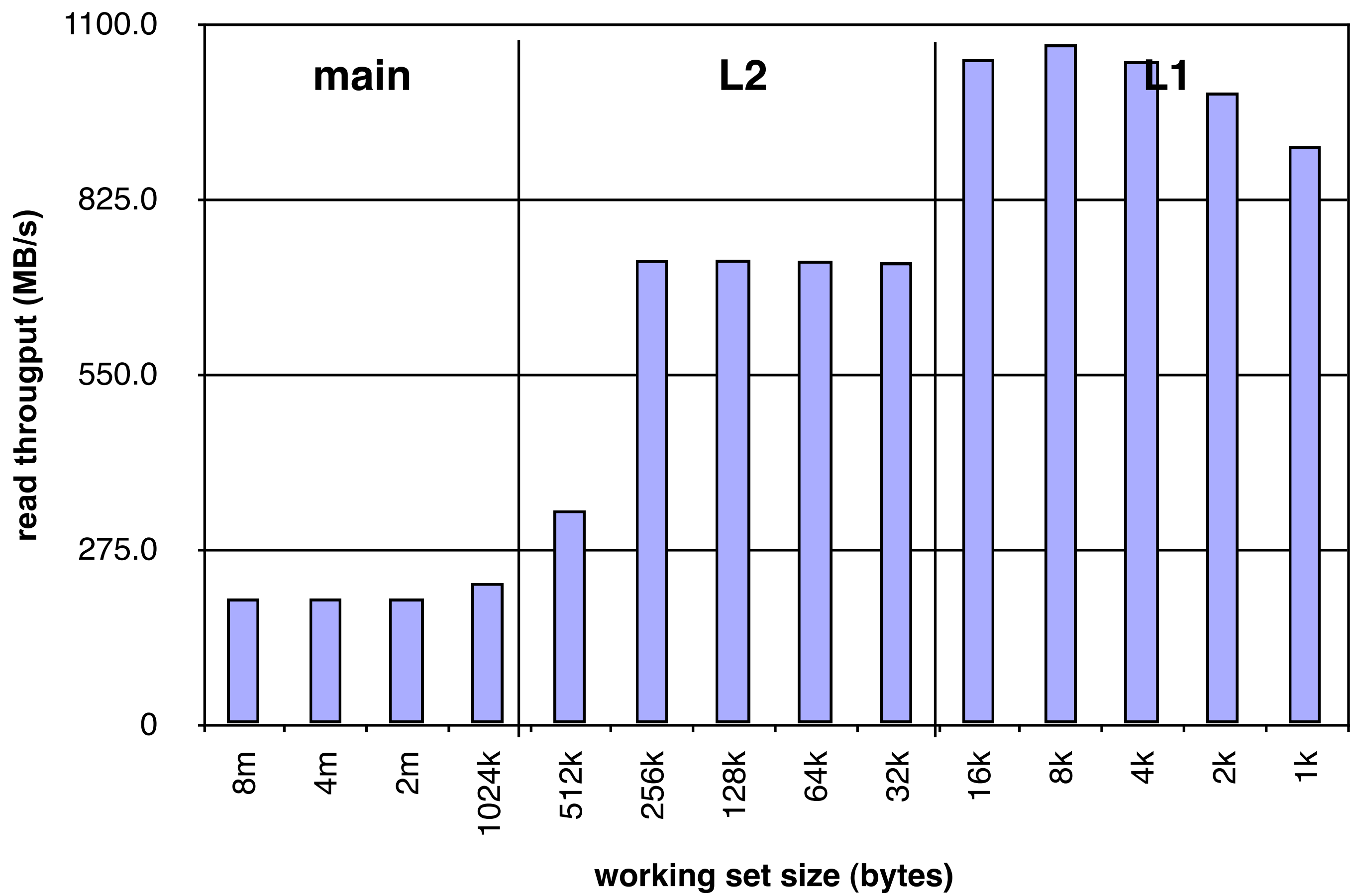
The memory mountain



Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified
L2 cache

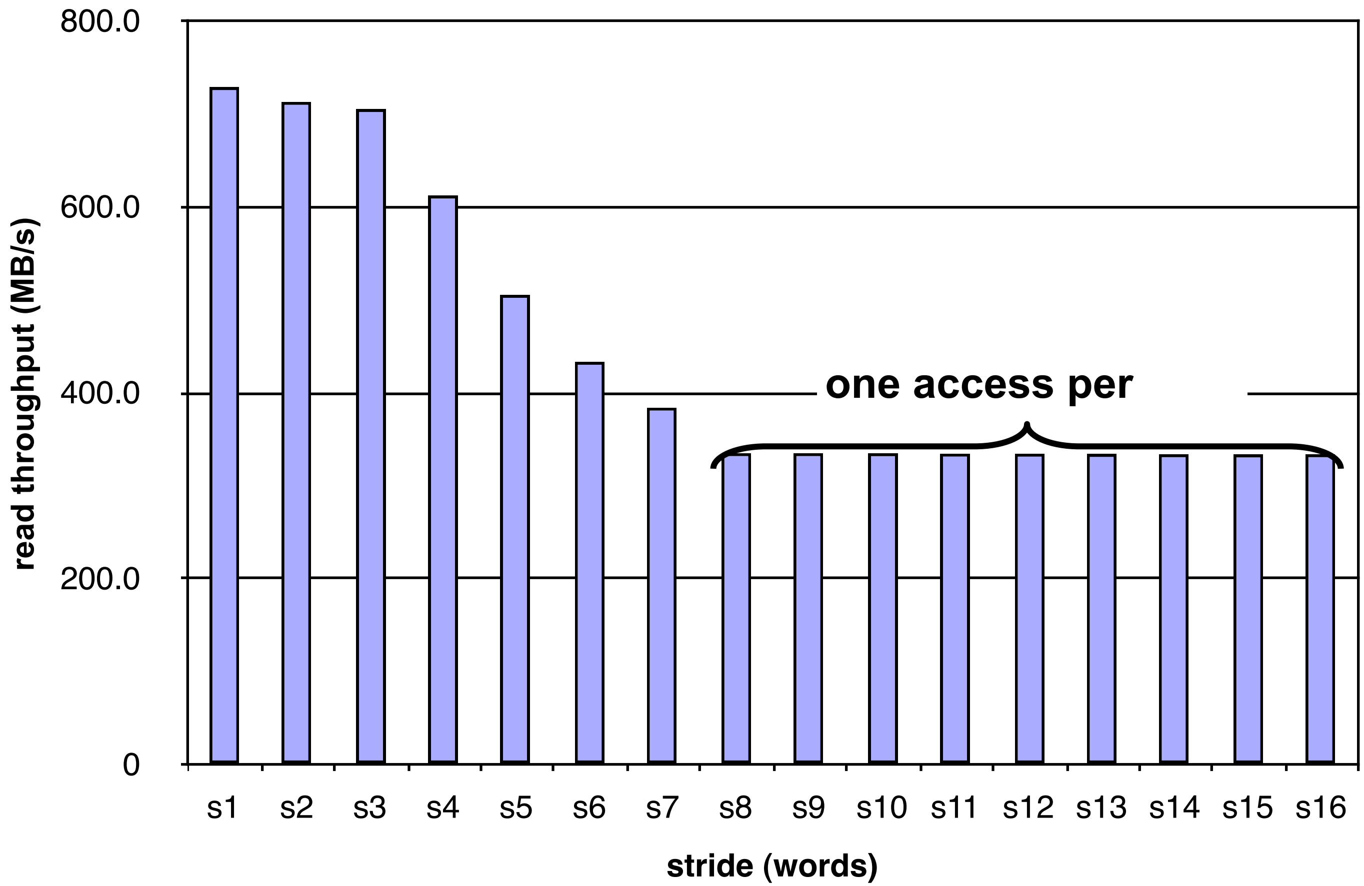
Ridges of temporal locality

- Slice through the memory mountain with stride=1
  - illuminates read throughputs of different caches and memory



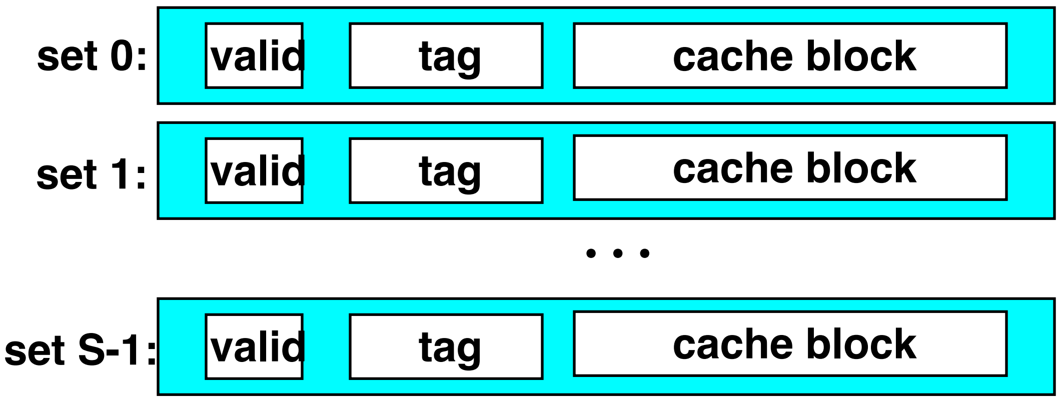
A slope of spatial locality

- Slice through memory mountain with size=256KB
  – shows cache block size.



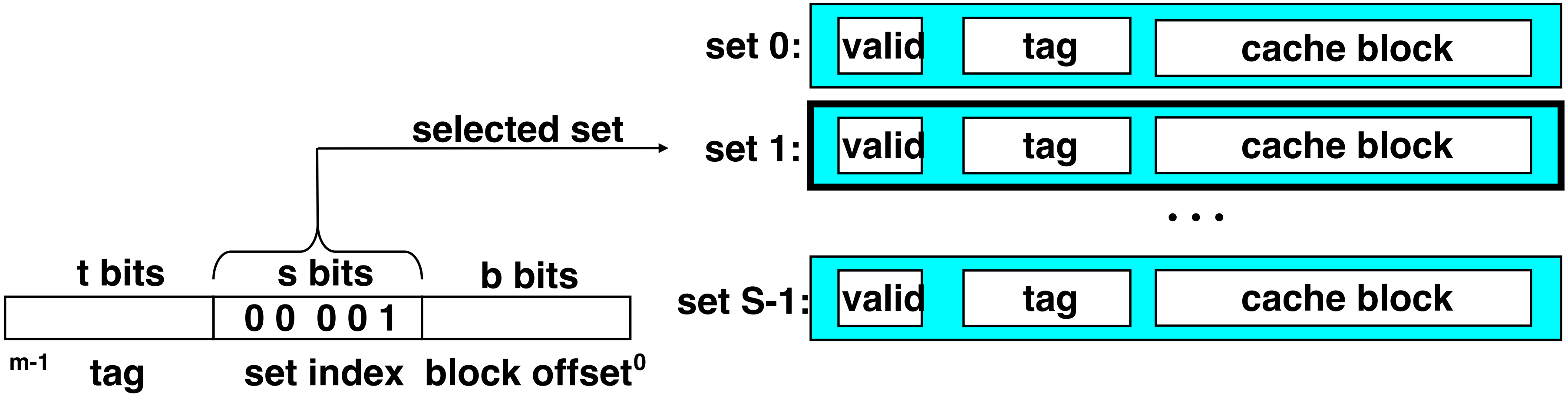
Direct-mapped cache

- Simplest kind of cache
- Cache divided in S sets of N-byte blocks
  - $N = 2^b$, $S = 2^s$
  - Typically, N = 32 or 64 (our examples use 4 bytes)
  - Blocks capture spatial locality
- Valid bit = 1 if data in stored in set i
- Tag field identifies which address is currently stored



Accessing direct-mapped caches

- Low b bits determine block offset
- Middle s bits of address determine index set
- Store remaining t bits in tag



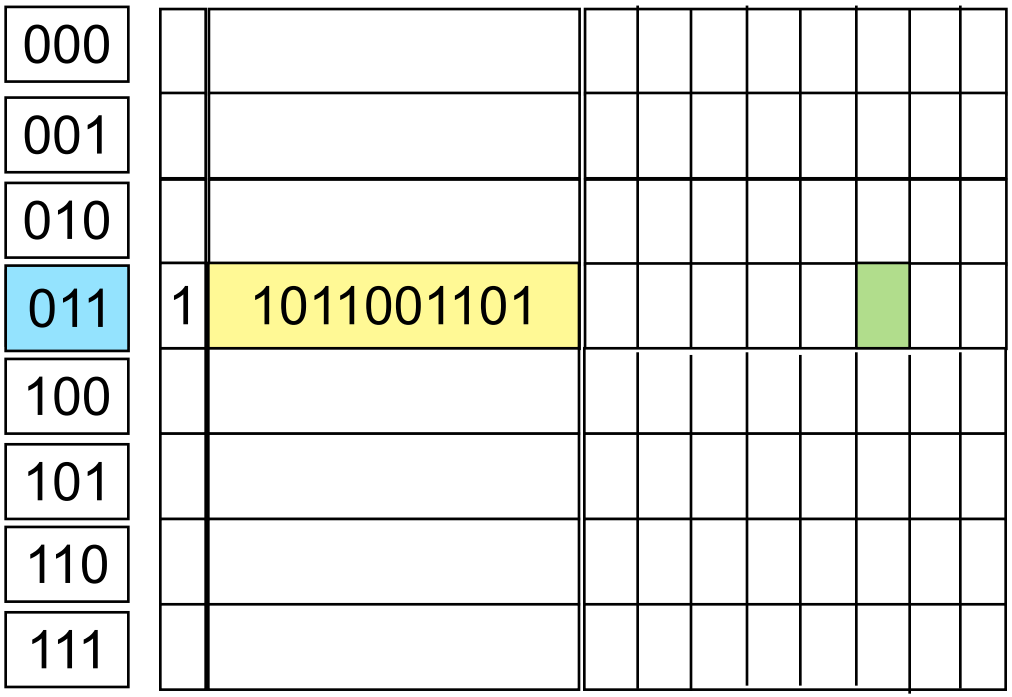
Accessing direct-mapped caches

Example: 16 bit addresses, 8 sets, 8 byte block in each set

tag=10	s=3	b=3
xxxxxxxxxx	xxx	xxx

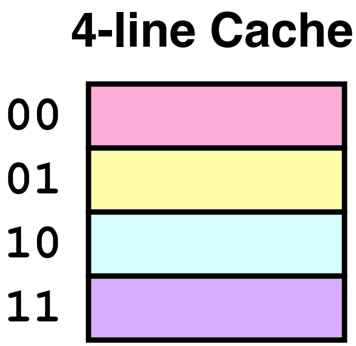
To store: 1011 0011 0101 1101

1011001101	011	101

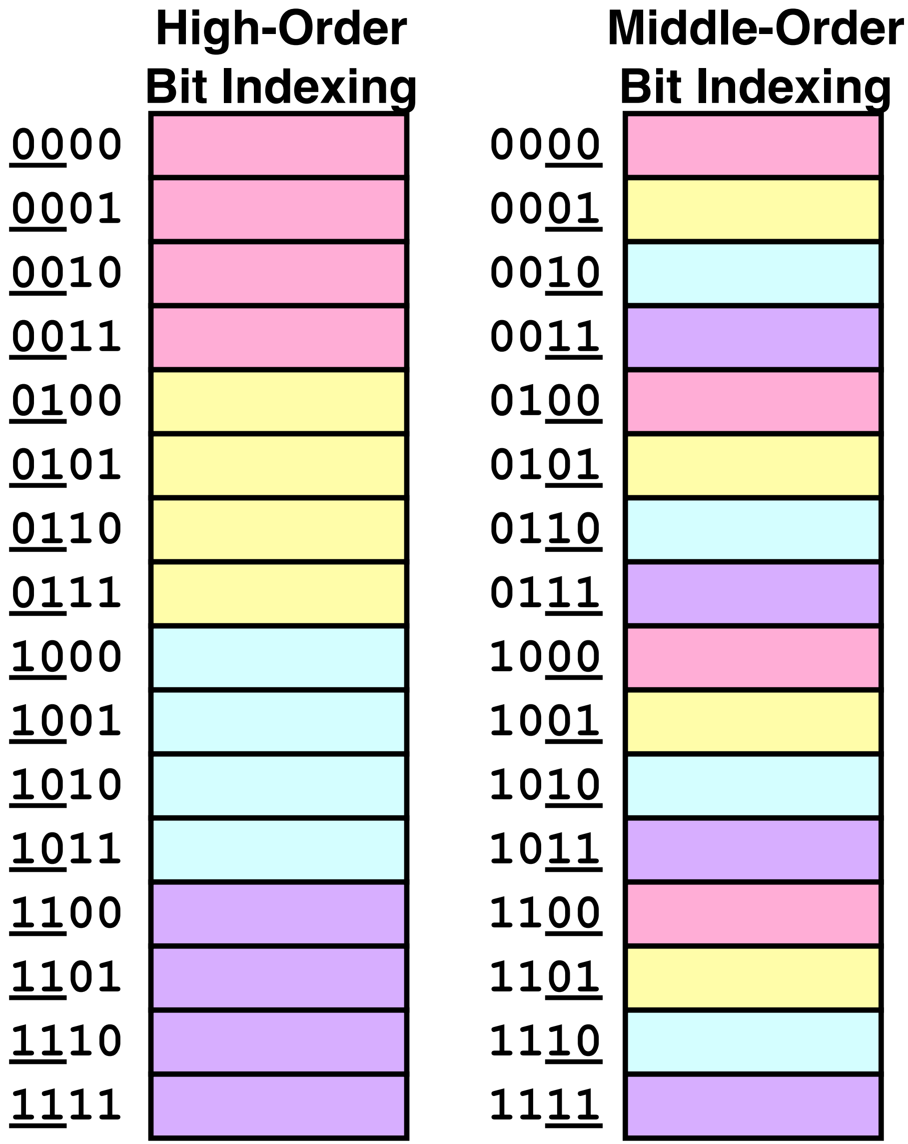


Checkpoint

Why use middle bits as index?

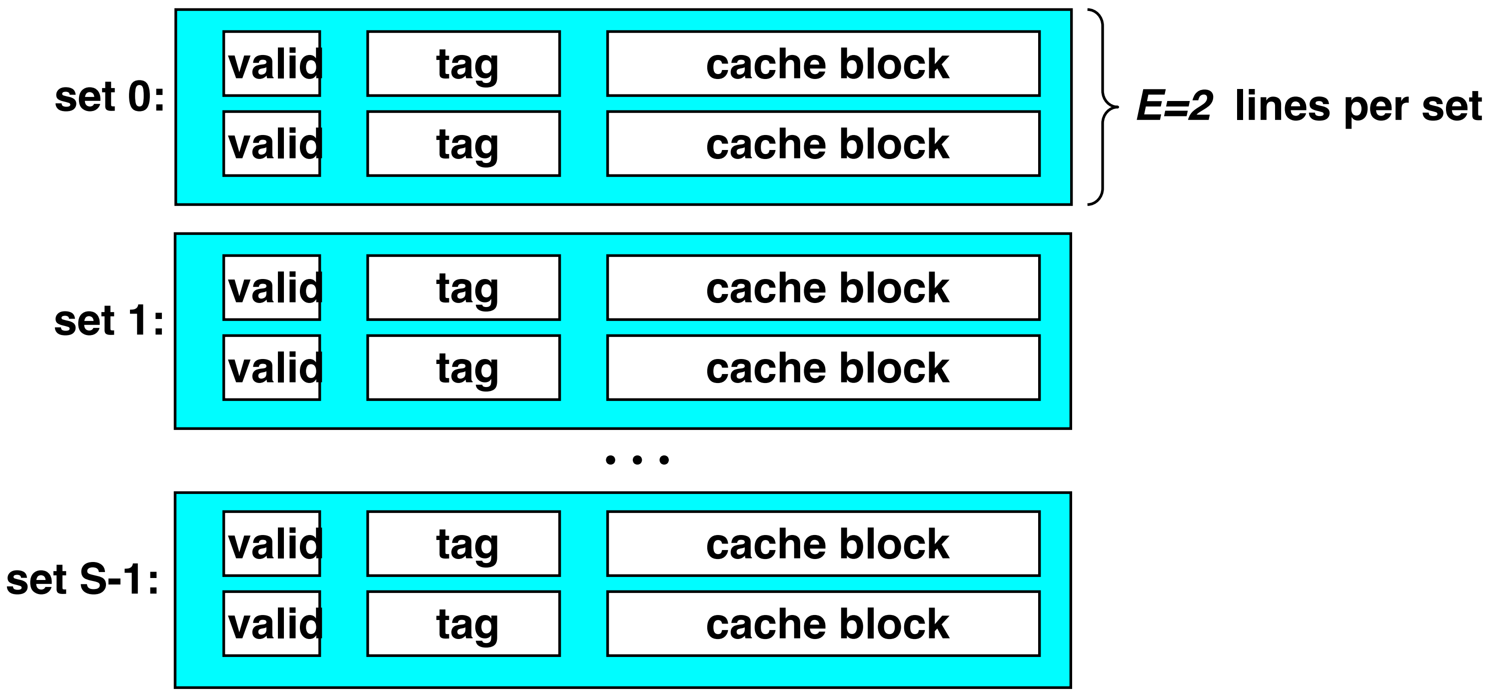


- High-order bit indexing
  - Adjacent memory lines would map to same cache entry
  - Spatially local code would have more cache conflicts
- Middle-order bit indexing
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time



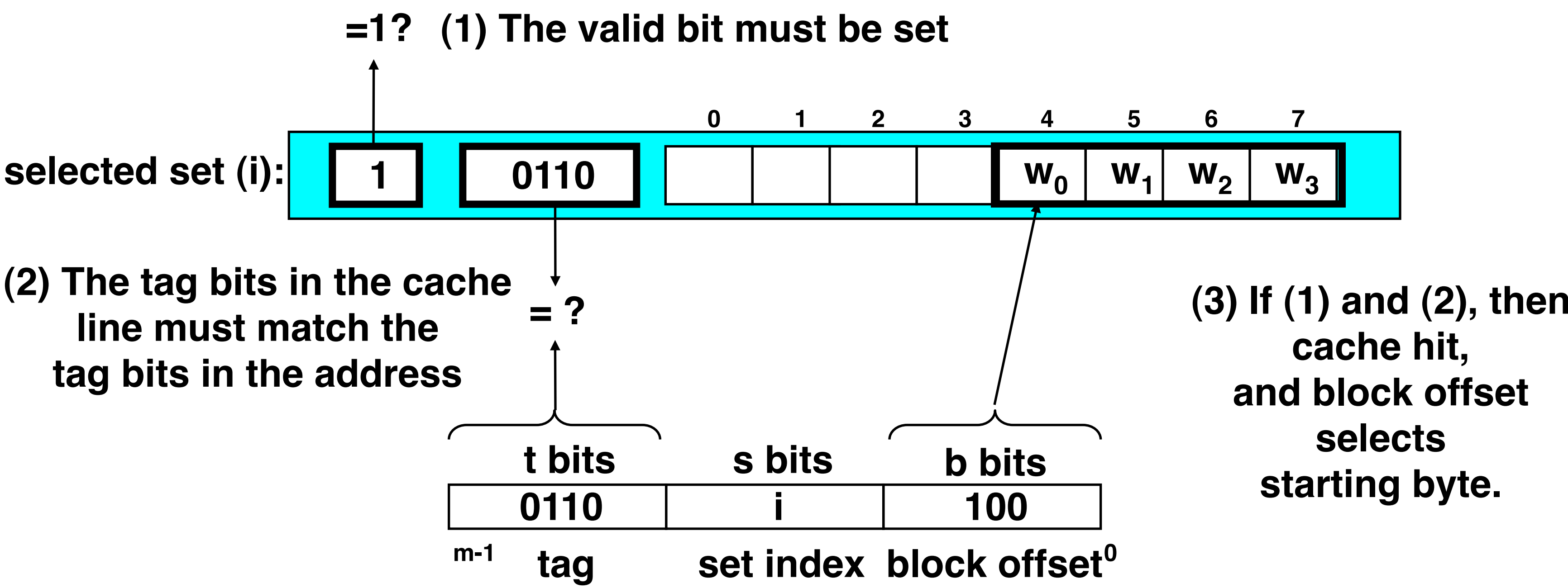
Set associative caches

- Characterized by more than one line per set



Accessing 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



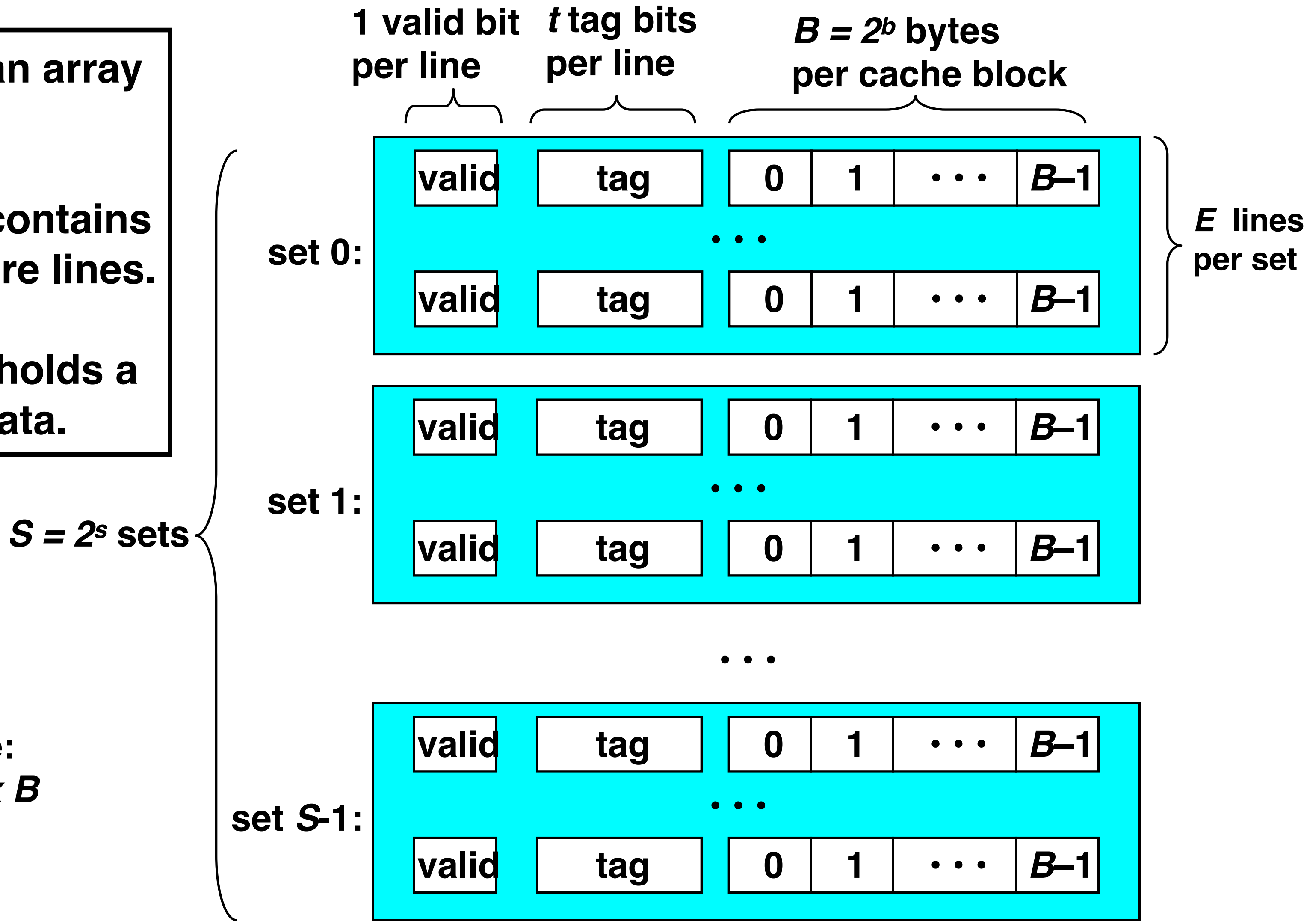
Monday, November 7, 2011

General org of a cache memory

Cache is an array of sets.

Each set contains one or more lines.

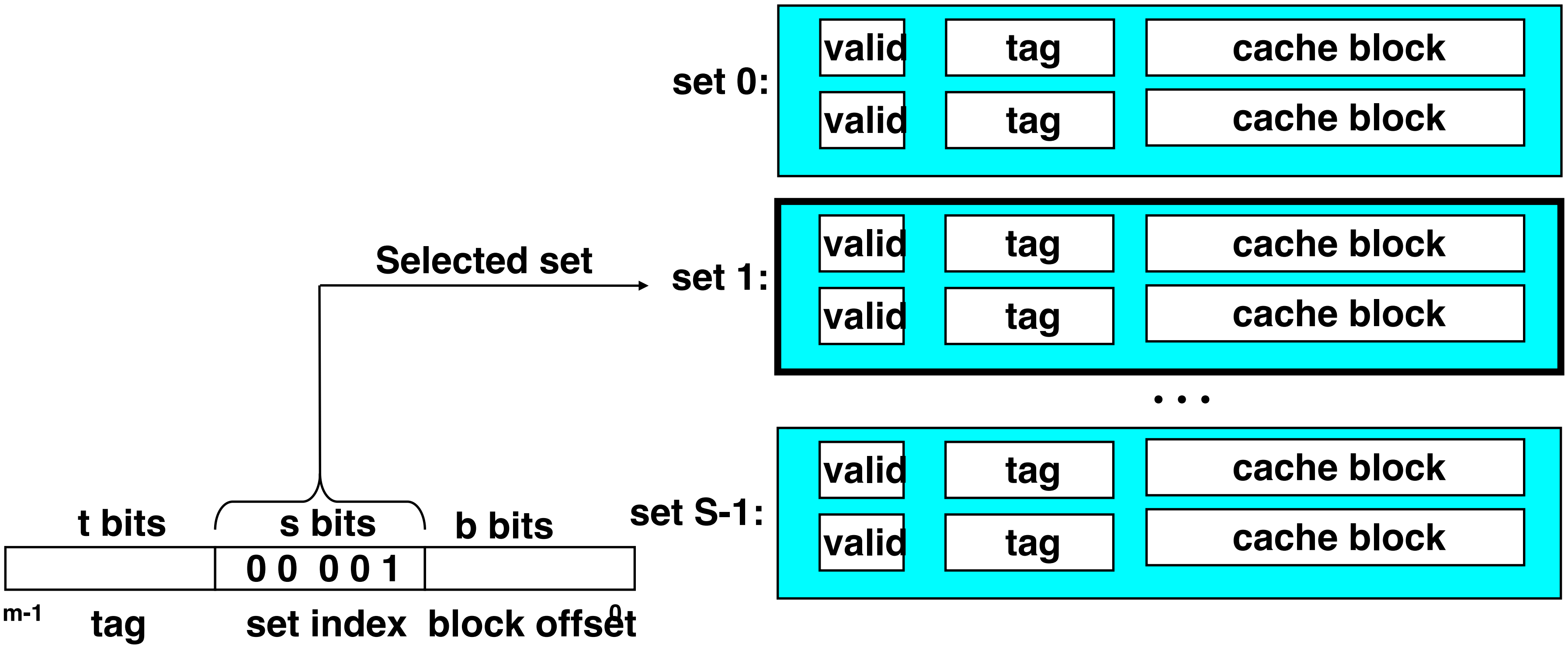
Each line holds a block of data.



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

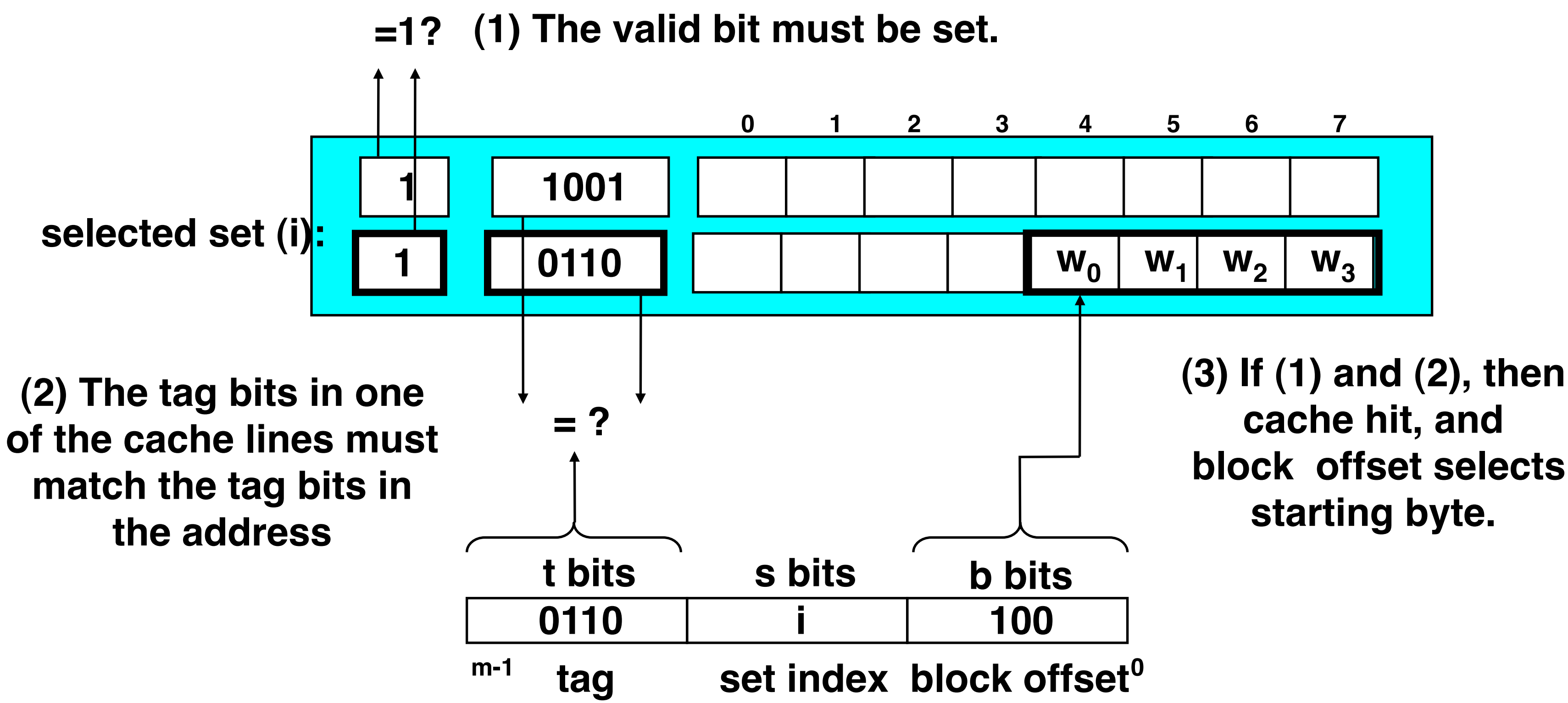
Accessing set associative caches

- Set selection
  – identical to direct-mapped cache

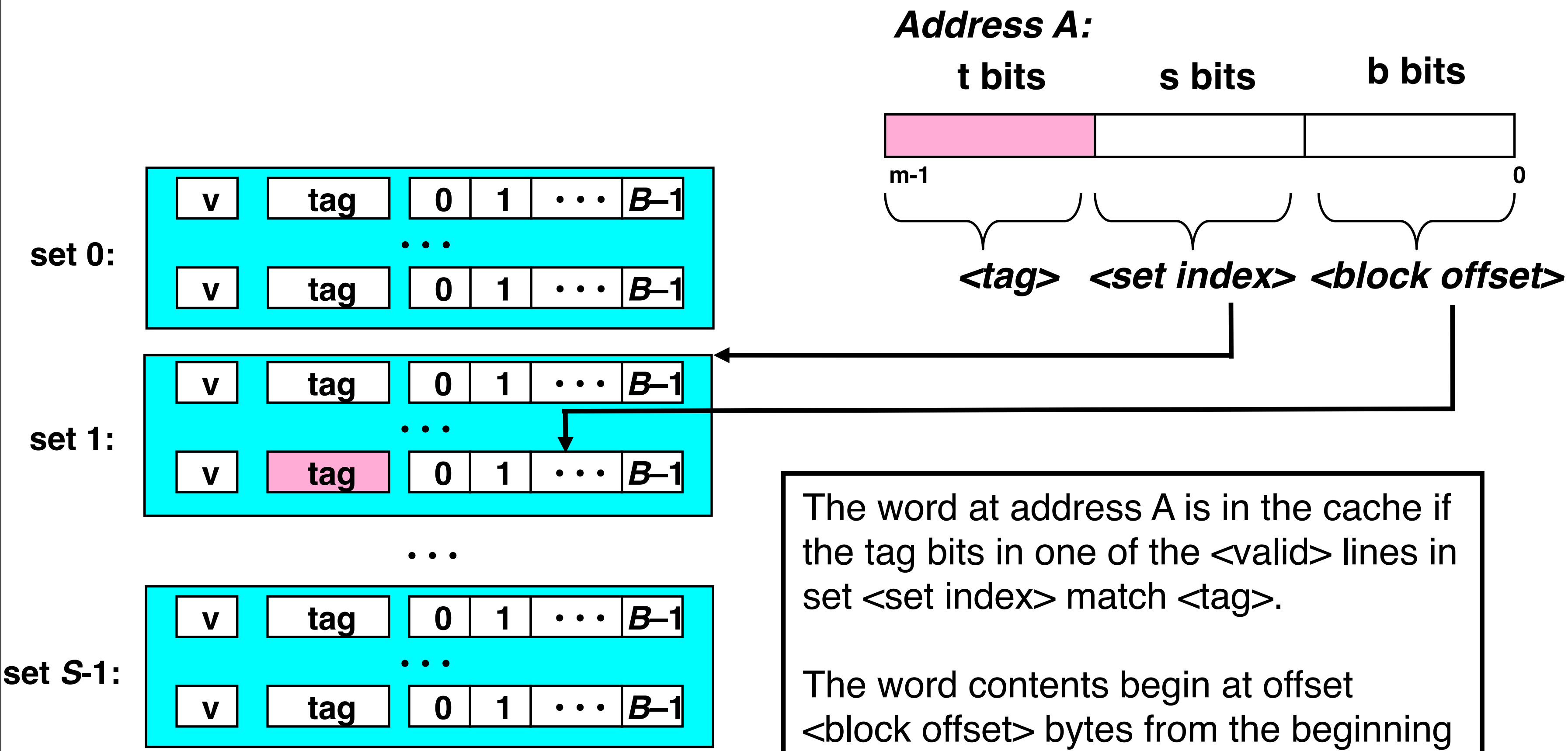


Accessing set associative caches

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



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.

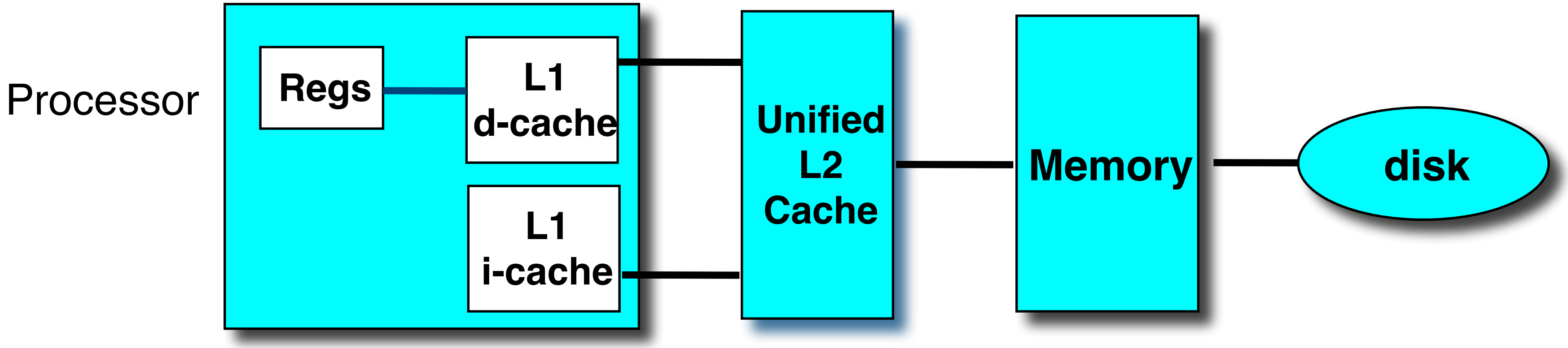
Cache Parameters

- $S = 2^s$: number of sets
- E: number of lines / set (E = 1 direct-mapped)
- $B = 2^b$: block size in bytes
- $m = \log_2(M)$: number of address bits
- $t = m - (s + b)$: number of tag bits
- $C = B \times E \times S$: cache size in bytes (blocks only, not valid and tag bits)

Checkpoint

Multi-level caches

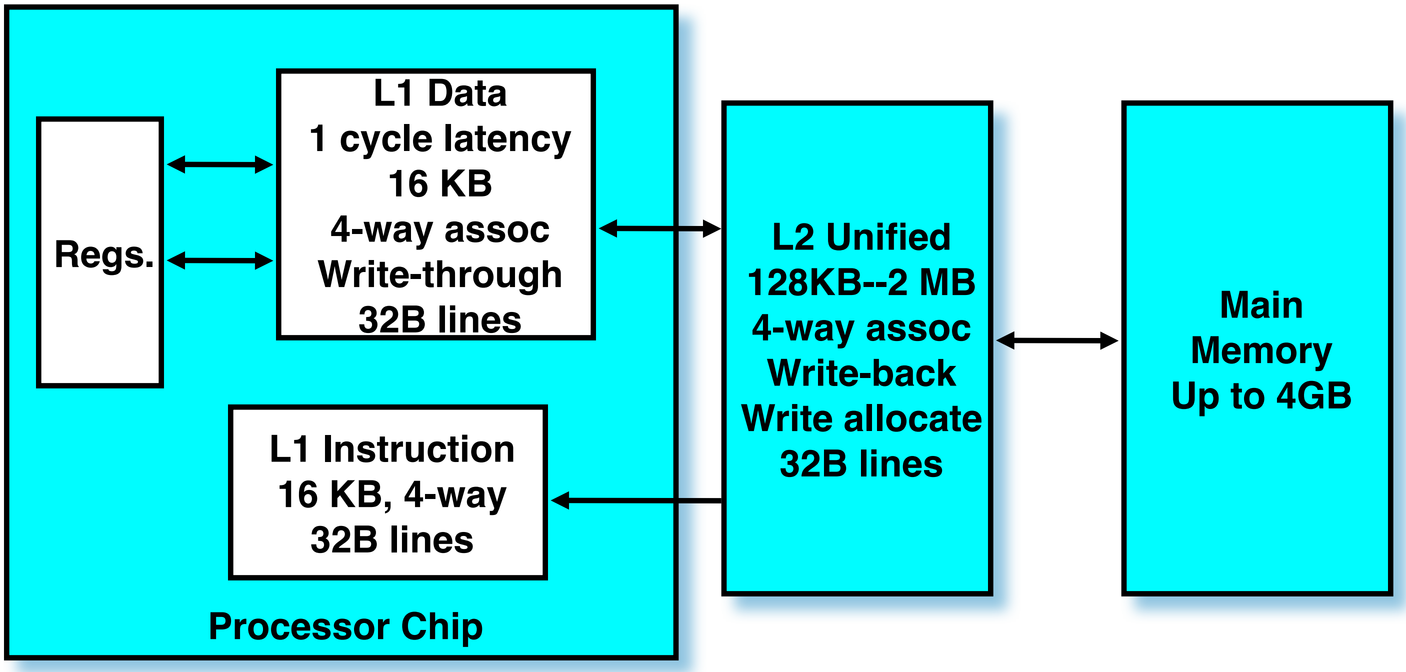
- Options: separate data and instruction caches, or a unified cache



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 →

Intel Pentium 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:
  - assume 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%**

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

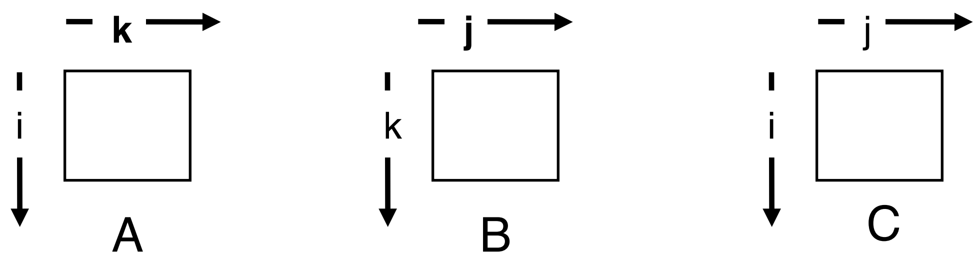
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

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%)

Conflict misses in Direct-Mapped Caches

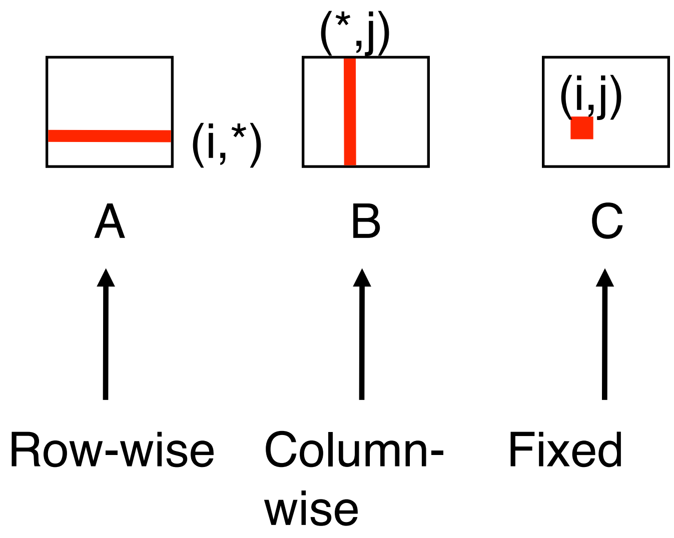
```
float dotprod(float x[8], float y[8])
{
  float sum = 0.0; int i;
  for (i = 0; i < 8; i++)
    sum += x[i] * y[i];
  return sum;
}
```

- Assume for simplicity
  - 4-byte floats
  - x[] loaded at address 0, y[] at address 32
  - 16 byte cache block (4 floats)
  - 2 sets (cache size = 32 bytes)
- x[0] – x[3] and y[0] – y[3] map to set 0
- x[4] – x[7] and y[4] – y[7] map to set 1
- Almost every array reference clobbers the same cache set
- This is called thrashing. Can make code 2 or 3 times slower.
- Fix by padding arrays to avoid powers of 2, e.g., x[12] and y[12].

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:



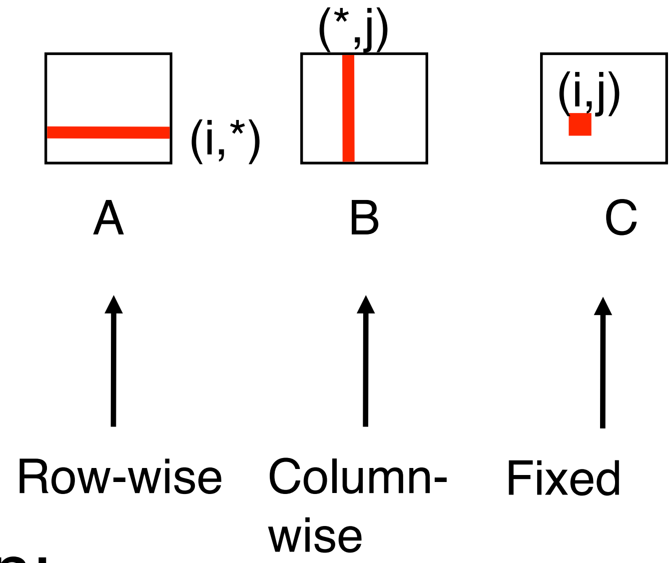
- <u>Misses per inner loop iteration:</u>

<u>A</u>	<u>B</u>	<u>C</u>
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:



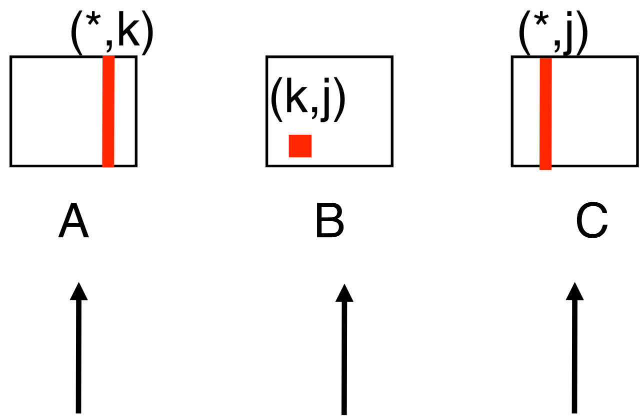
- <u>Misses per inner loop iteration:</u>

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

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:



Column - wise | Fixed | Column- wise

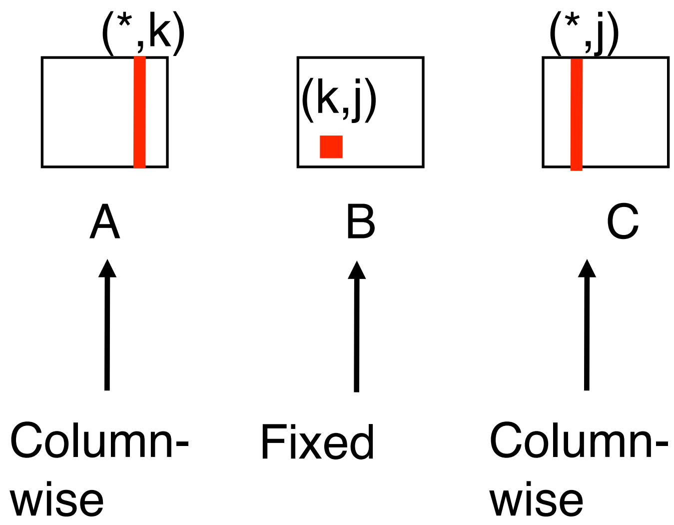
- <u>Misses per inner loop iteration:</u>

<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:



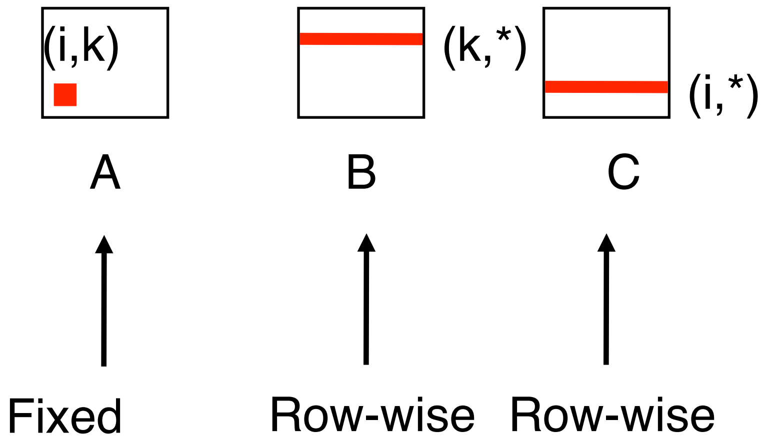
- <u>Misses per inner loop iteration:</u>

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

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];
  }
}
```

Inner loop:



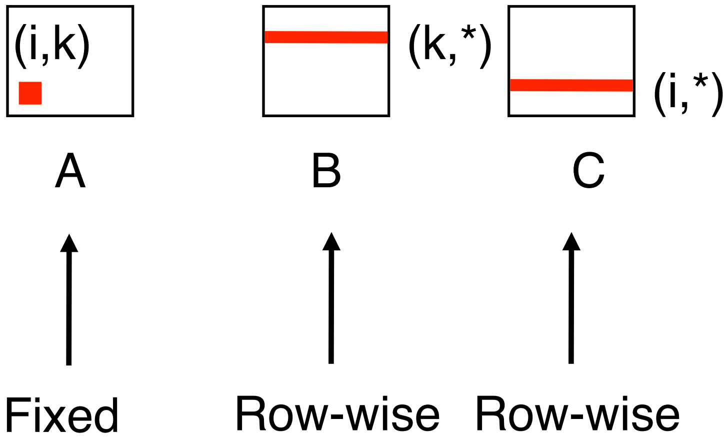
- <u>Misses per inner loop iteration:</u>

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

Inner loop:



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

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

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;
  }
}
```

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;
  }
}
```

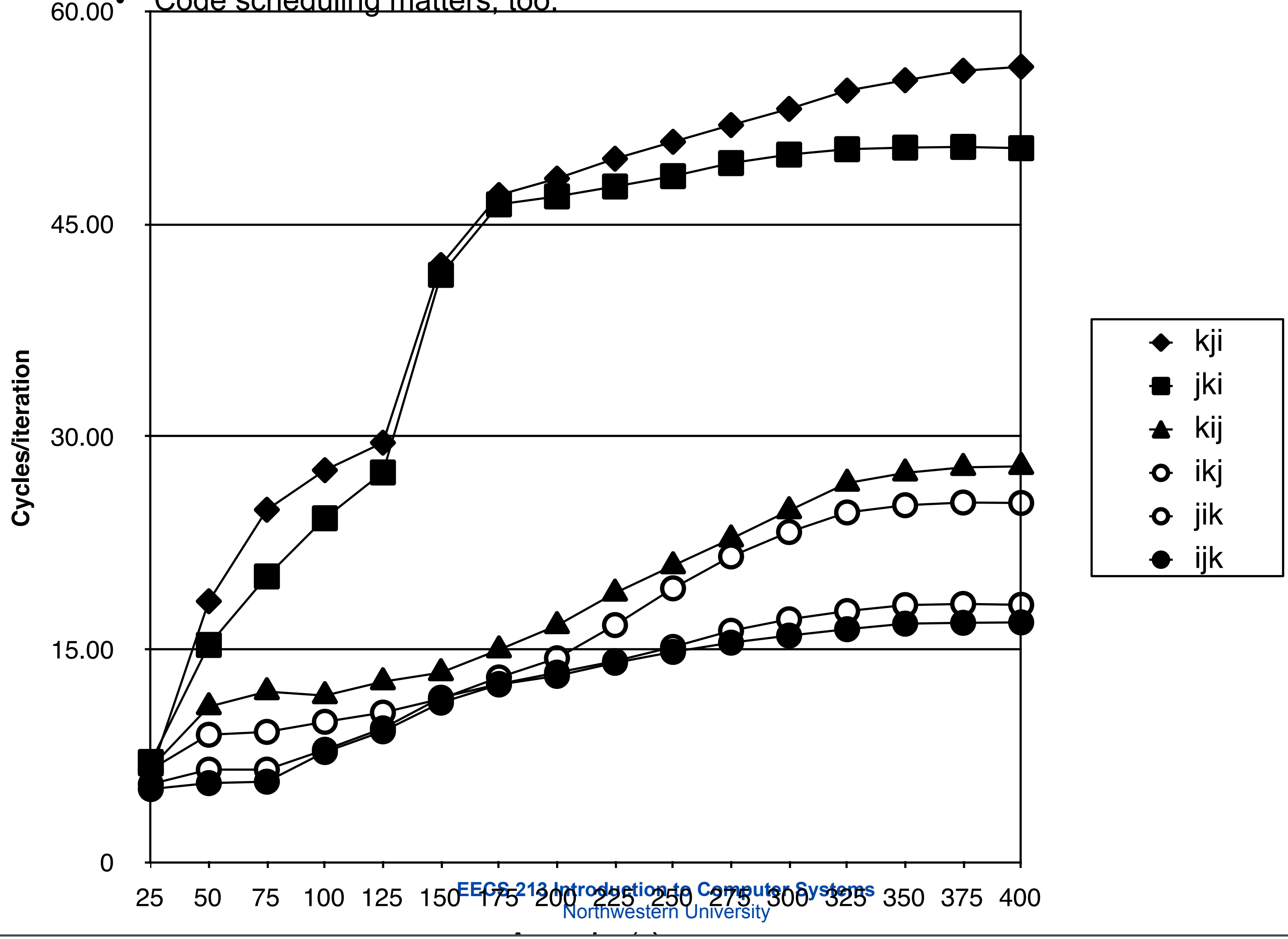
kij & ikj:

- 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];
  }
}
```

Pentium 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

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

Key idea: 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)

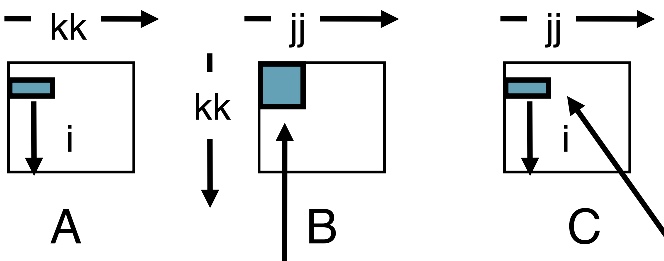
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum +=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



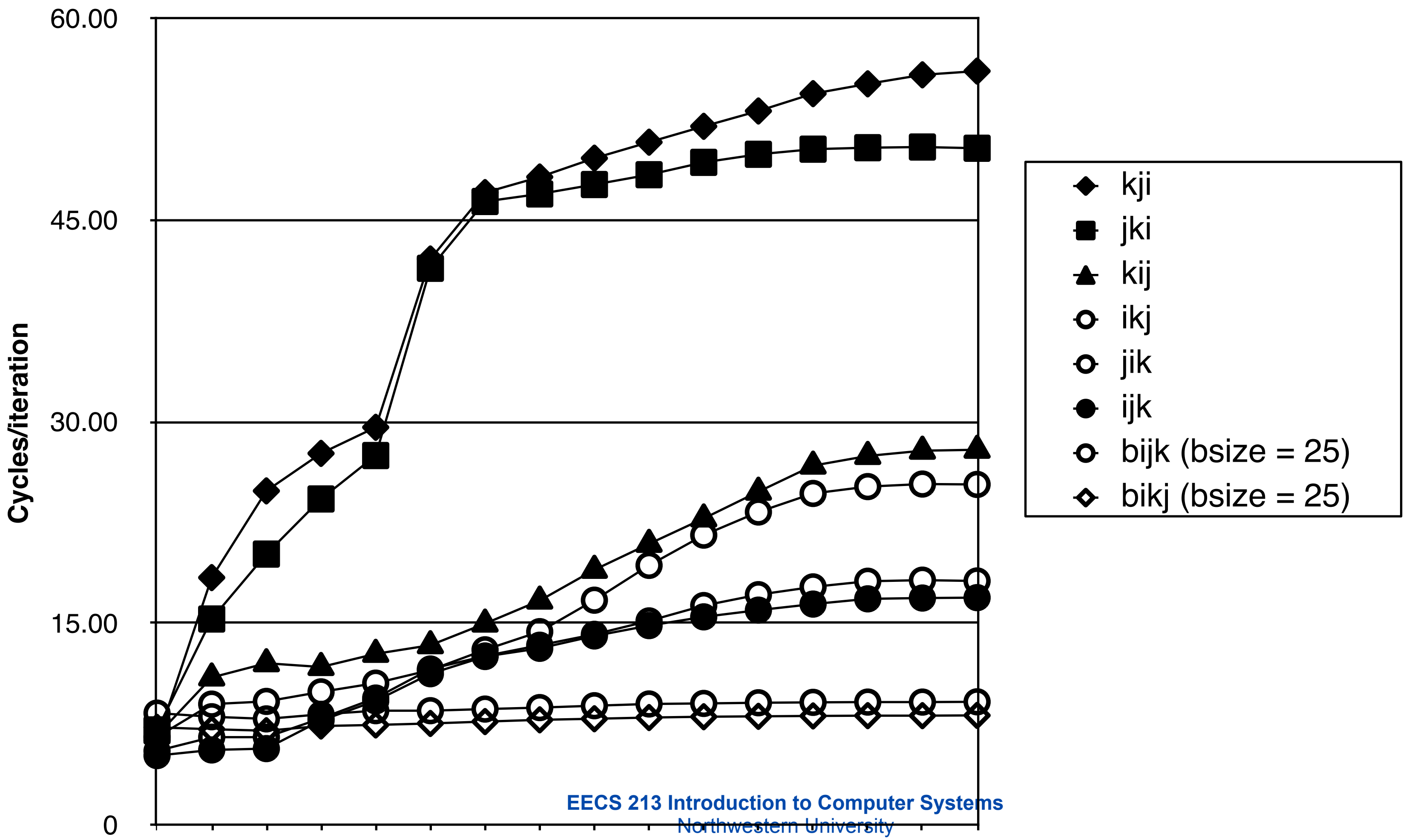
row sliver accessed ***bsize*** times

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)