

# Last Time: Memory

- Technologies: SRAM, DRAM, Flash, EEPROM, disk, ...
- Memory gap is getting wider
- Memories arranged in a hierarchy
  - Some levels are transparent, some are not
  - Works because there is locality
- Caching is the way that locality is exploited

# Today

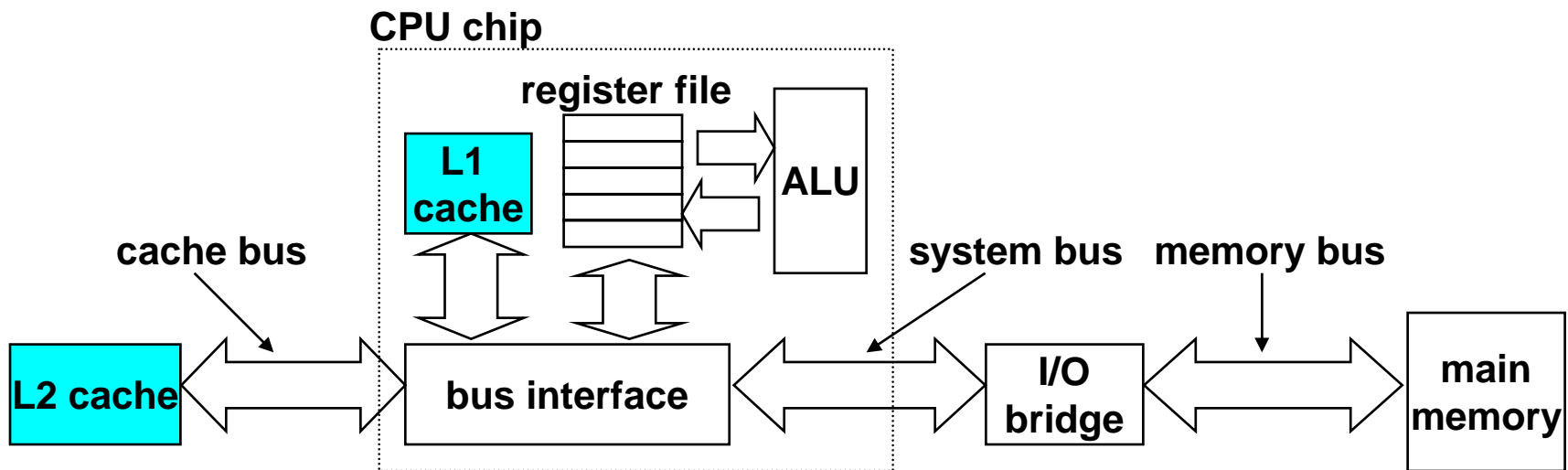
- Lab 4 handed out

# Today: Cache Memories

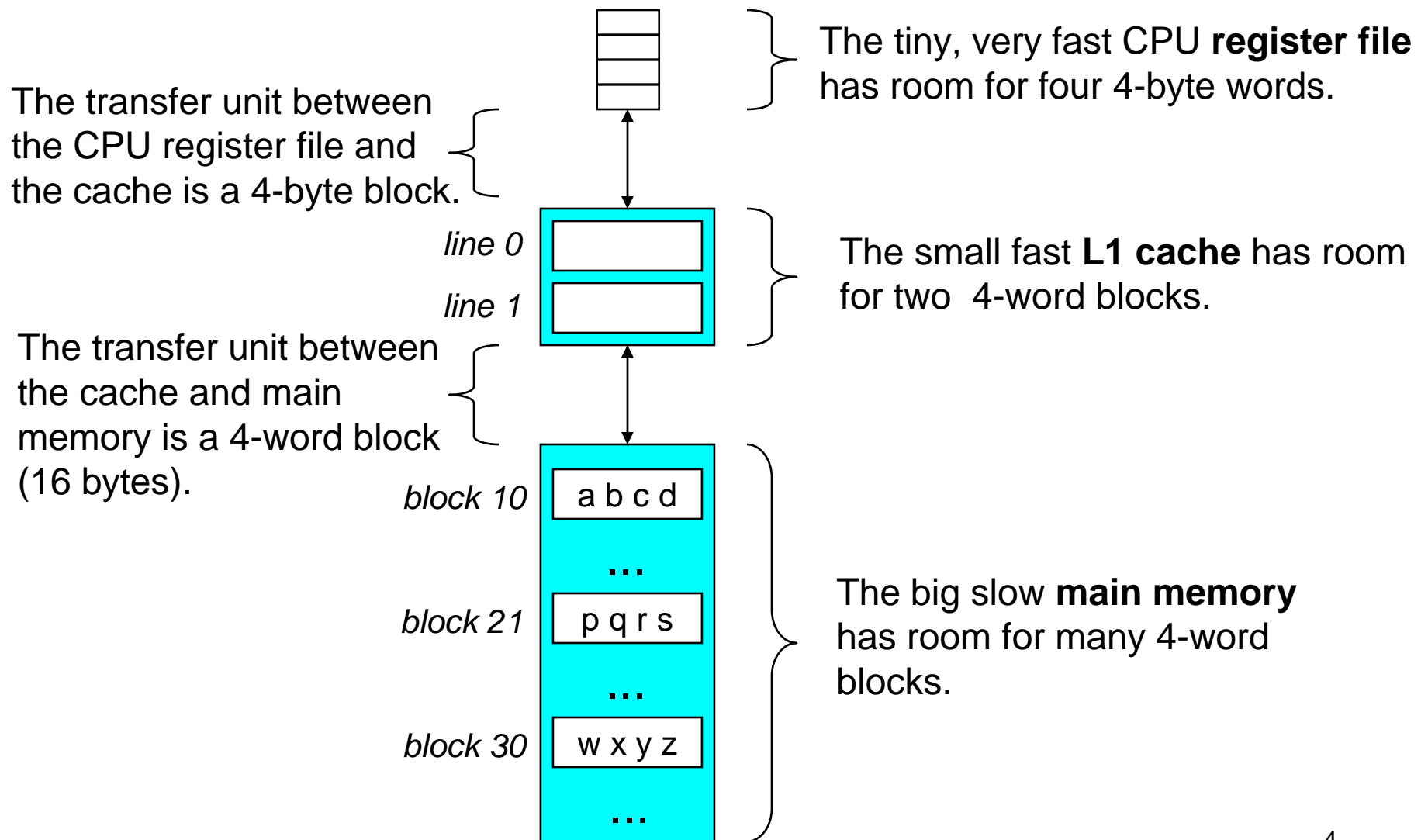
- **Topics**
  - Generic cache memory organization
  - Direct mapped caches
  - Set associative caches
  - Impact of caches on performance

# 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 Between the CPU and Main Memory

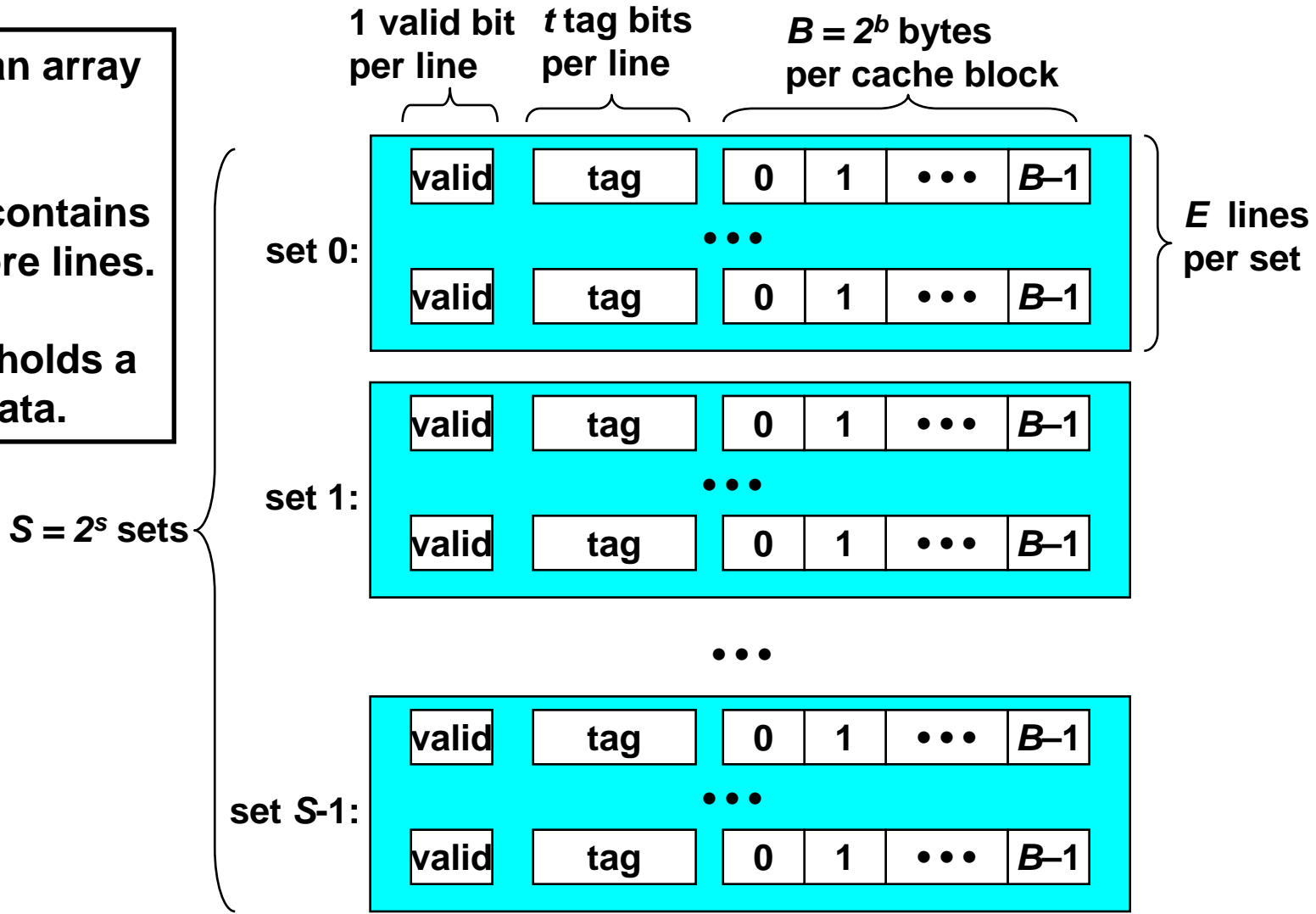


# General Org of a Cache Memory

Cache is an array of sets.

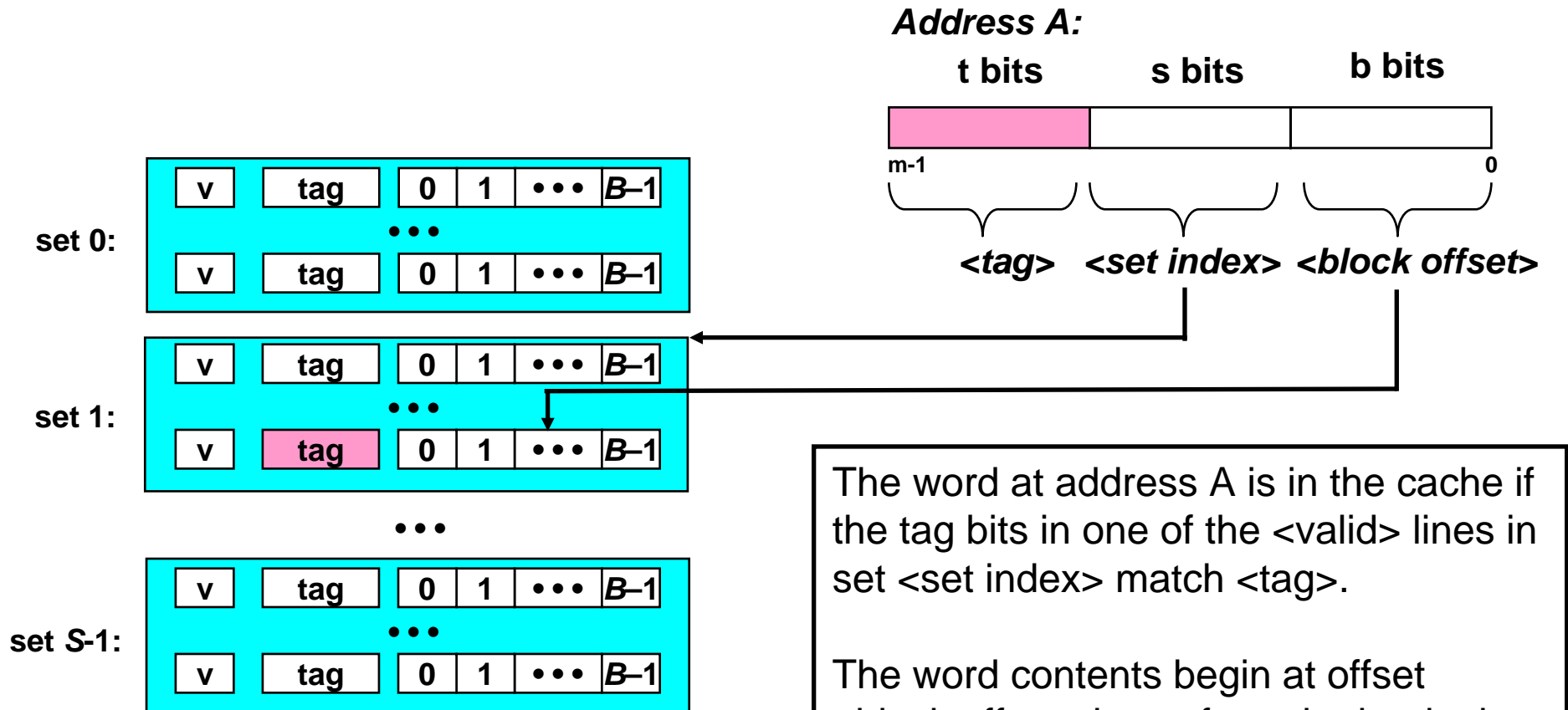
Each set contains one or more lines.

Each line holds a block of data.



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

# Addressing Caches

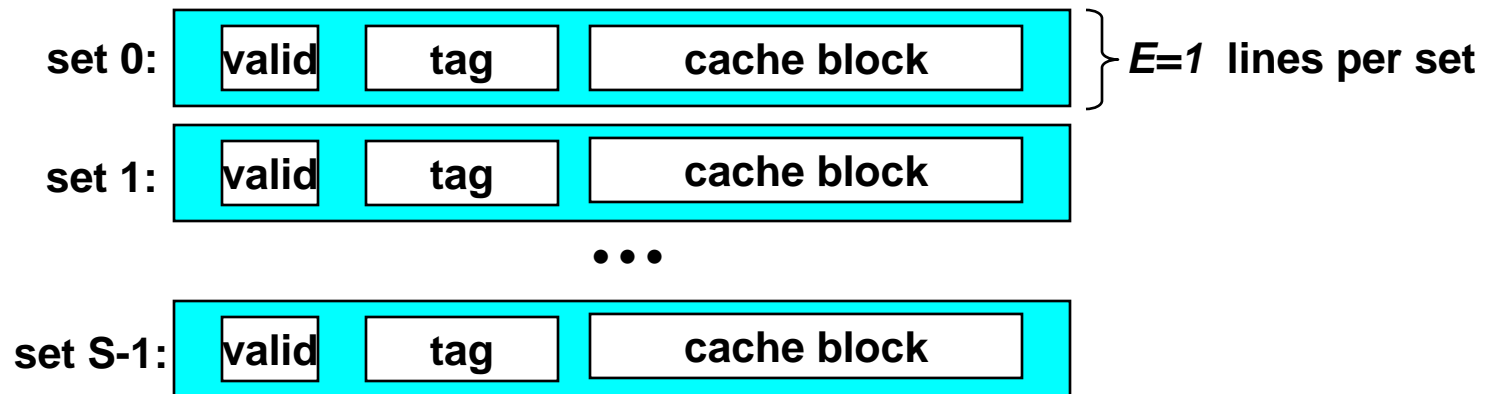


The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

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

# Direct-Mapped Cache

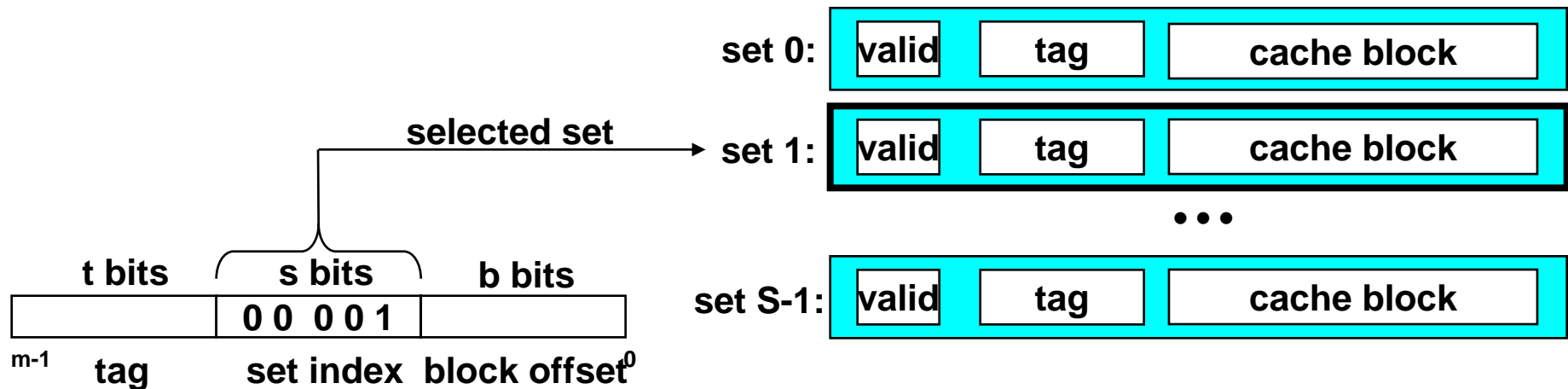
- Simplest kind of cache
- Characterized by exactly one line per set.



# Accessing Direct-Mapped Caches

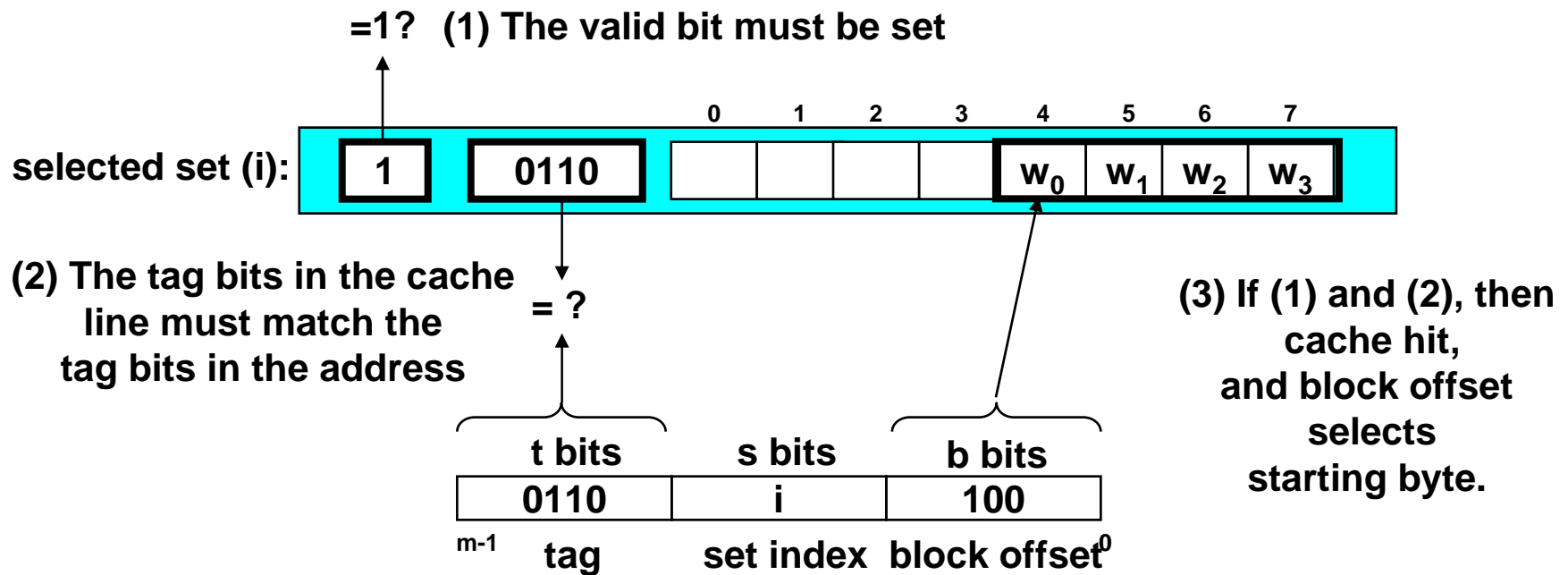
- **Set selection**

- 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 trace (reads):

0 [0000<sub>2</sub>], 1 [0001<sub>2</sub>], 13 [1101<sub>2</sub>], 8 [1000<sub>2</sub>], 0 [0000<sub>2</sub>]

0 [0000<sub>2</sub>] (*miss*)

v	tag	data
1	0	M[0-1]

(1)

13 [1101<sub>2</sub>] (*miss*)

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

(3)

8 [1000<sub>2</sub>] (*miss*)

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

(4)

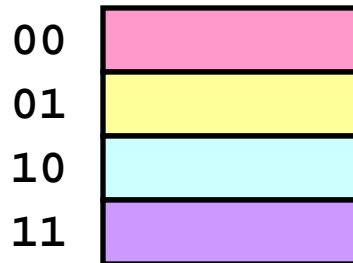
0 [0000<sub>2</sub>] (*miss*)

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

(5)

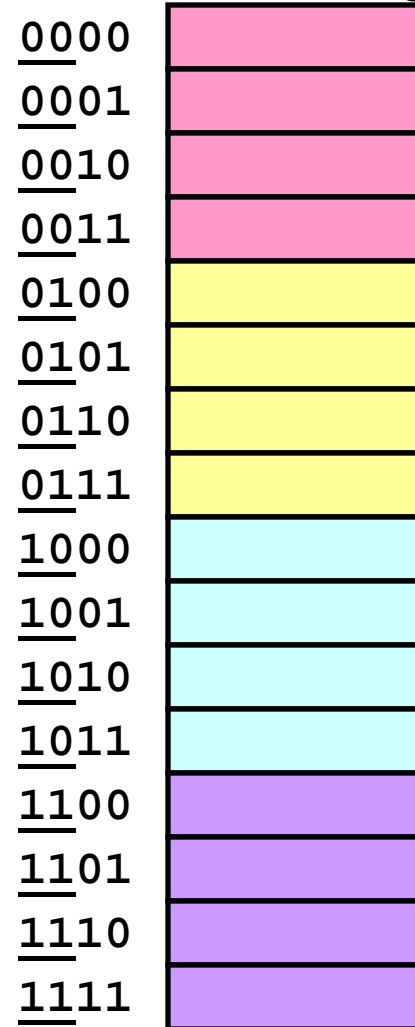
# Why Use Middle Bits as Index?

4-line Cache

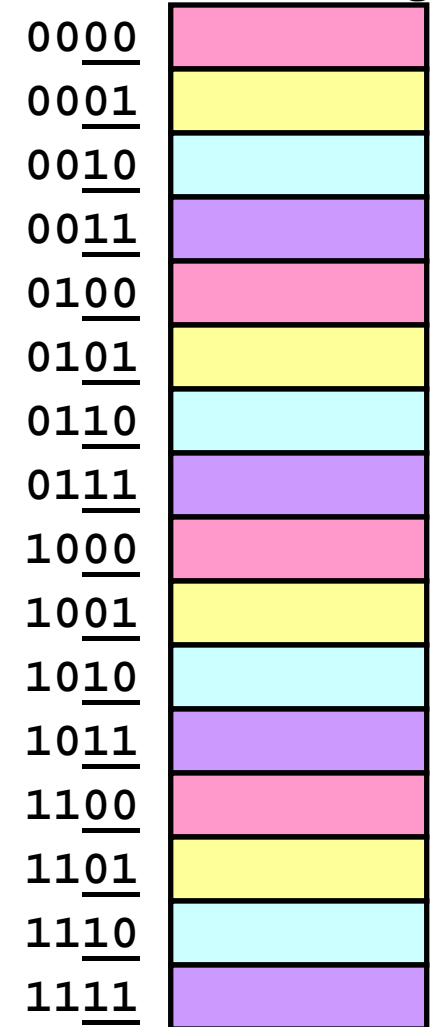


- **High-Order Bit Indexing**
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality
- **Middle-Order Bit Indexing**
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time

High-Order Bit Indexing

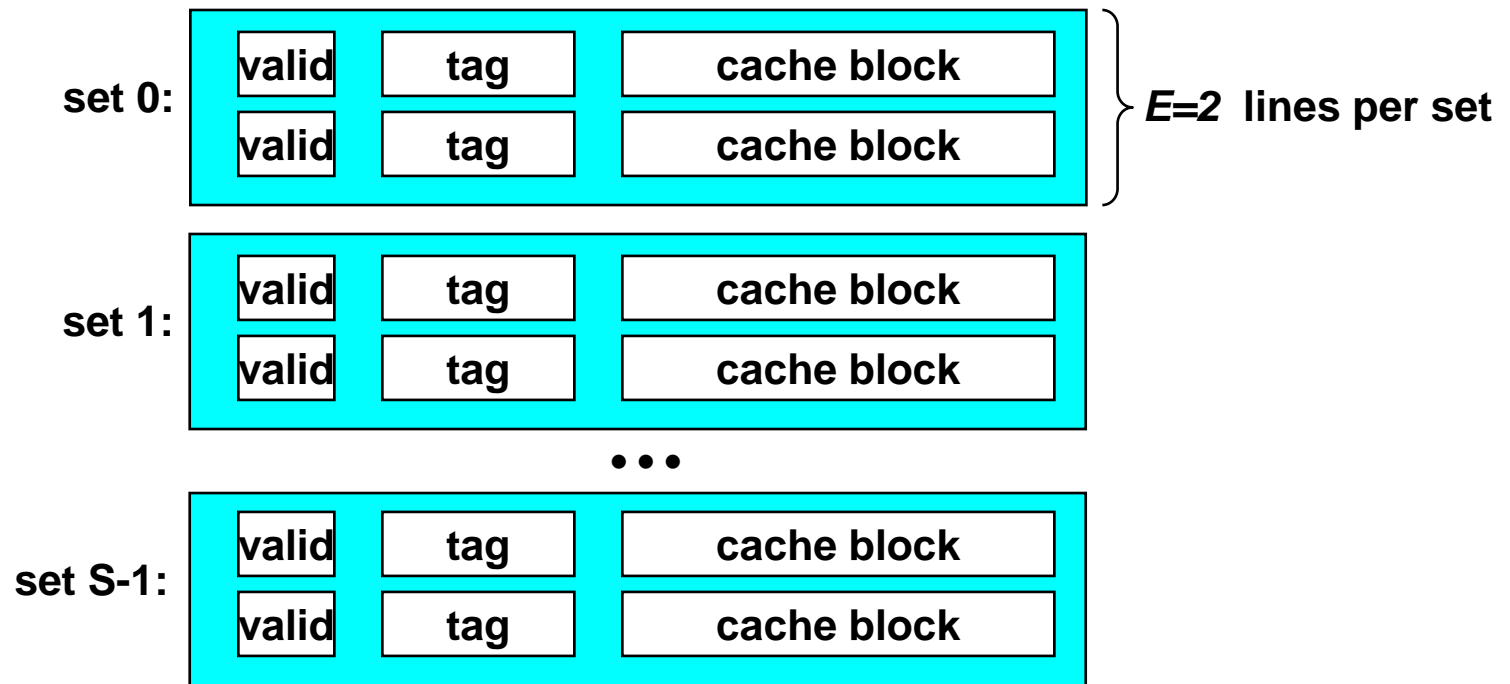


Middle-Order Bit Indexing



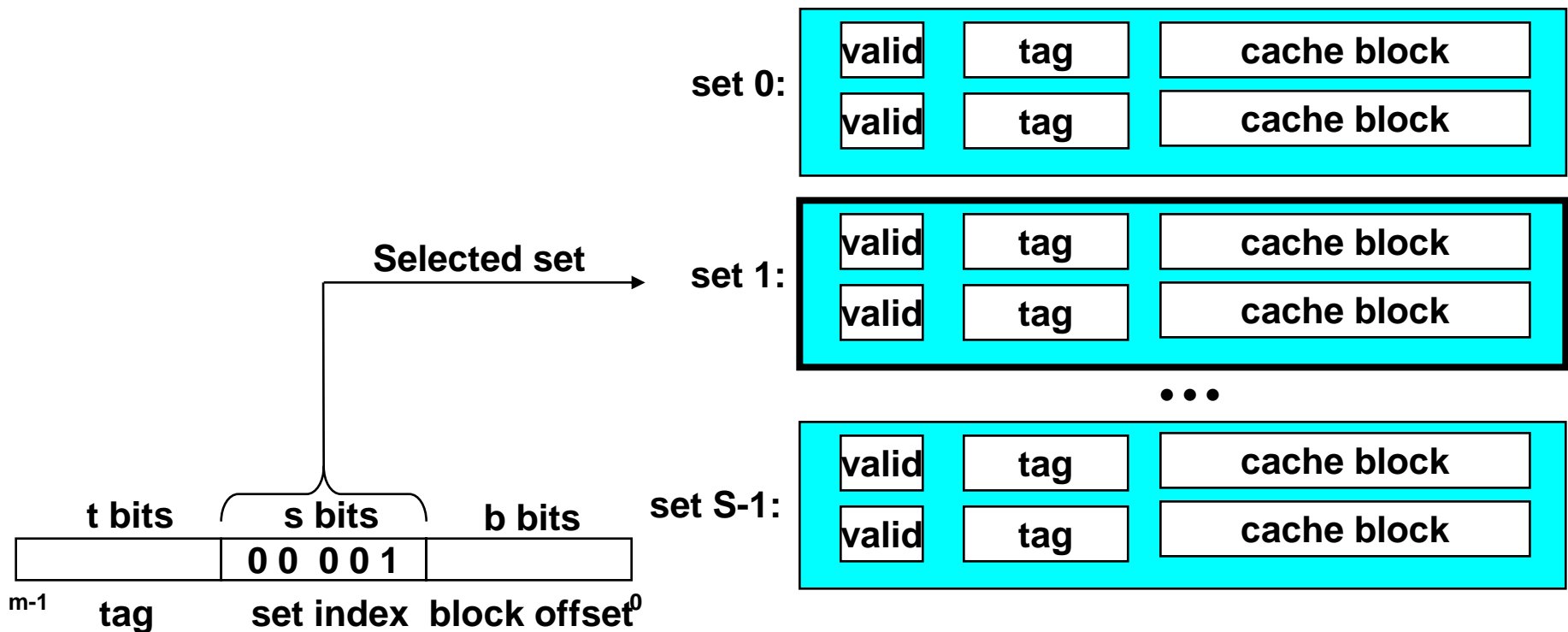
# Set Associative Caches

- Characterized by more than one line per set



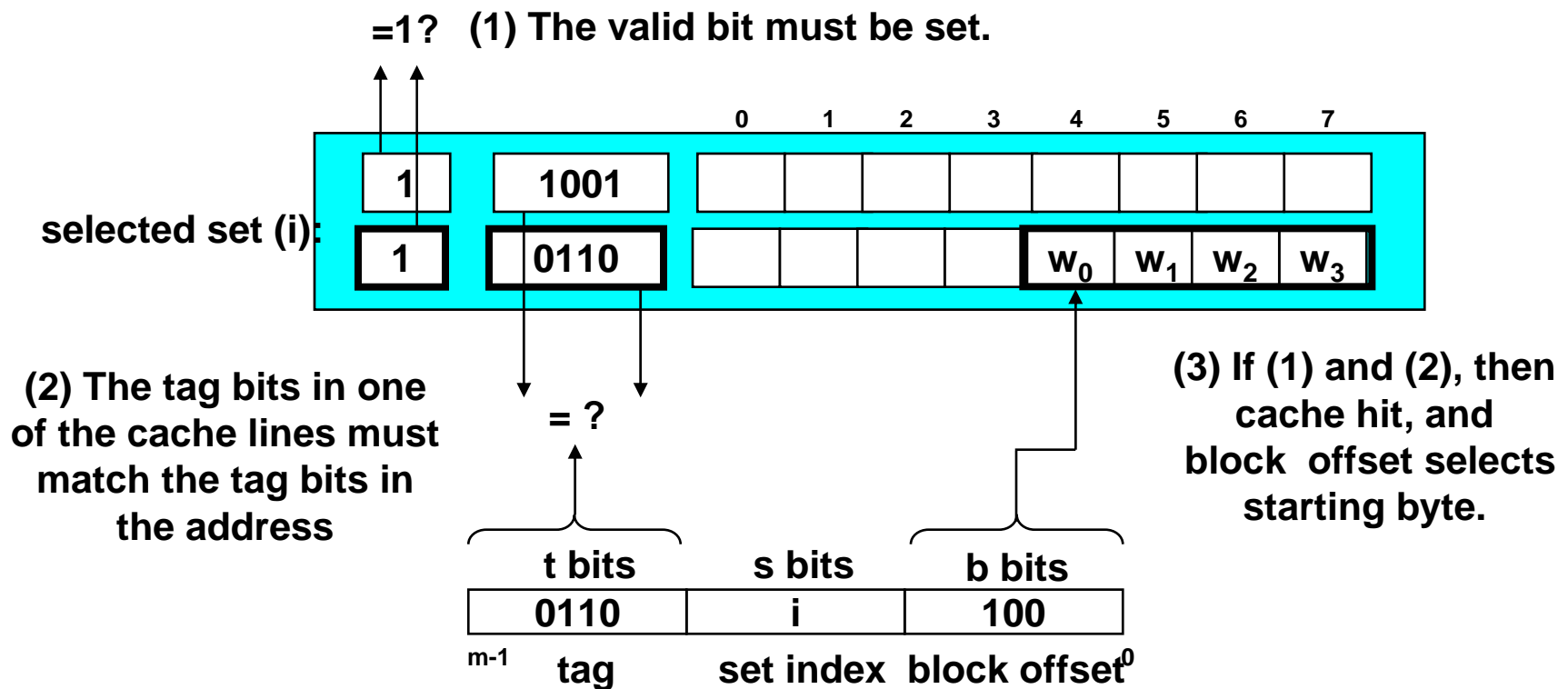
# Accessing Set Associative Caches

- **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.

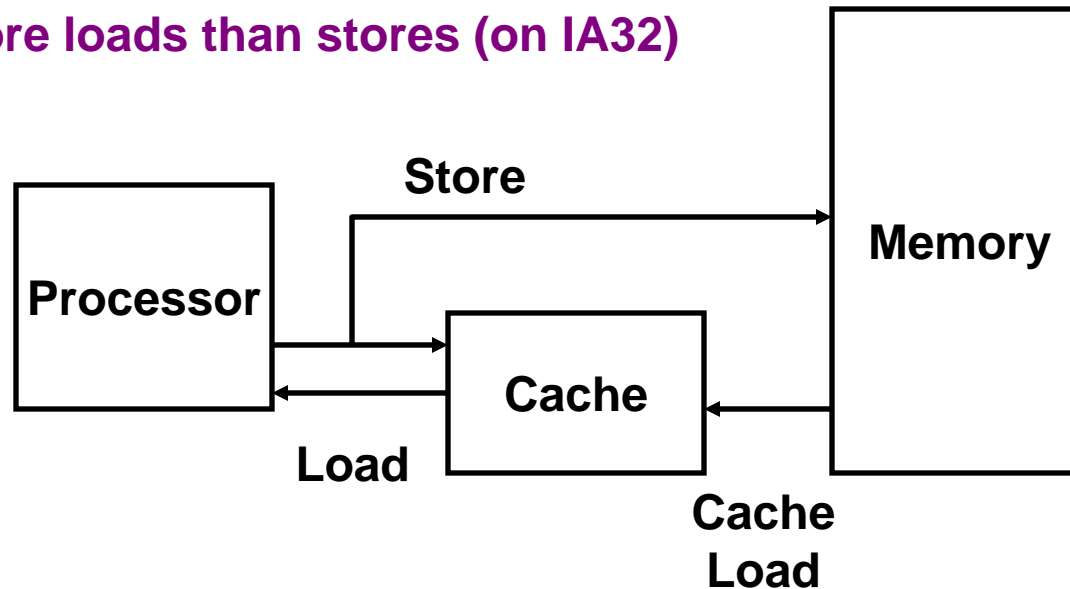


# LRU Replacement Issues

- **True LRU**
  - requires a timer per line → too expensive
- **LRU'ish**
  - add a status tag bit U
    - » U = used
    - » periodically clear use bit
      - can be a hardware timer, counter, ....
      - or periodic OS software clear
        - often a side effect of setting a particular bit in a status register
- **LRU'ish replacement**
  - if any invalid ways then randomly choose 1 of them for replacement
    - » else if “unused” ways then randomly choose 1 for replacement
    - » else randomly choose a way for replacement

# Write Policy

- What happens when processor writes to the cache?
- Should memory be updated as well?
- **Write Through:**
  - Store by processor updates cache *and* memory.
  - Memory always consistent with cache
  - Never need to store from cache to memory
  - ~2X more loads than stores (on IA32)



# Write Through Issues

- **Advantages**

- **memory is kept up to date**
  - » inclusion property is always valid

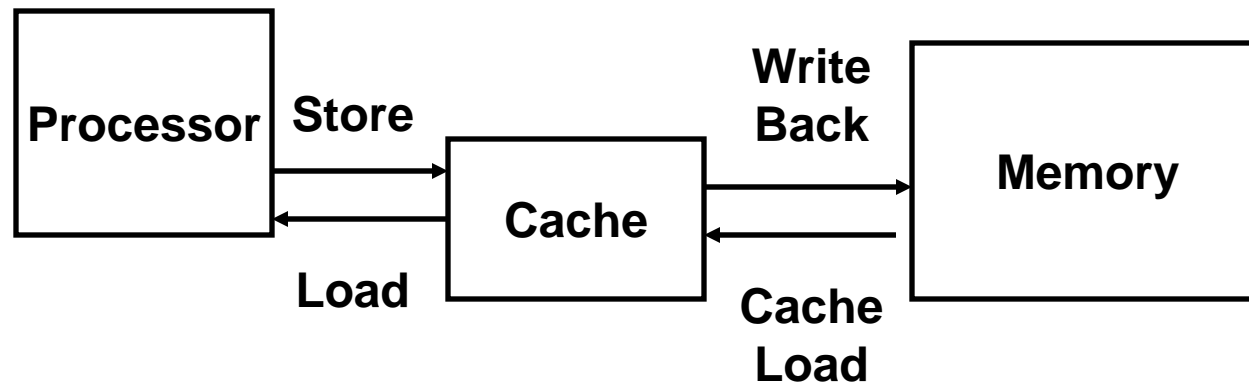
- **Disadvantages**

- **processor directly coupled to main memory performance**
  - » widening memory gap → big problem
- **NOTE however**
  - » CPU is really dependent on reads
  - » writes can effectively complete in the background
    - so stage the write data in a write-buffer
    - now processor only stalls when write buffer fills
    - BUT this kills the ONLY write-through advantage – DOH!

# Write Strategies (Cont.)

- **Write Back:**

- Store by processor only updates cache line
- Modified line written to memory only when it is evicted
  - » Requires “dirty bit” for each line (another status tag bit)
    - Set when line in cache is modified
    - Indicates that line in memory is stale
    - dirty bit can be used in place of a U bit for LRU’ish replacement
- Memory not always consistent with cache



# More Write Issues

- **What happens on a write miss?**
  - **2 options**
  - **write – allocate**
    - » get the line and put it in the cache
    - » then complete the write
  - **write – around**
    - » don't allocate the line
    - » just complete the write to the memory
- **Which is better**
  - **depends on the application & the cache policy**
    - » for normal data and write-back caches
      - write-allocate tends to make more sense
    - » for normal data and write-through caches
      - write-around is INCONSISTENT/ILLEGAL
    - » how about I/O data??

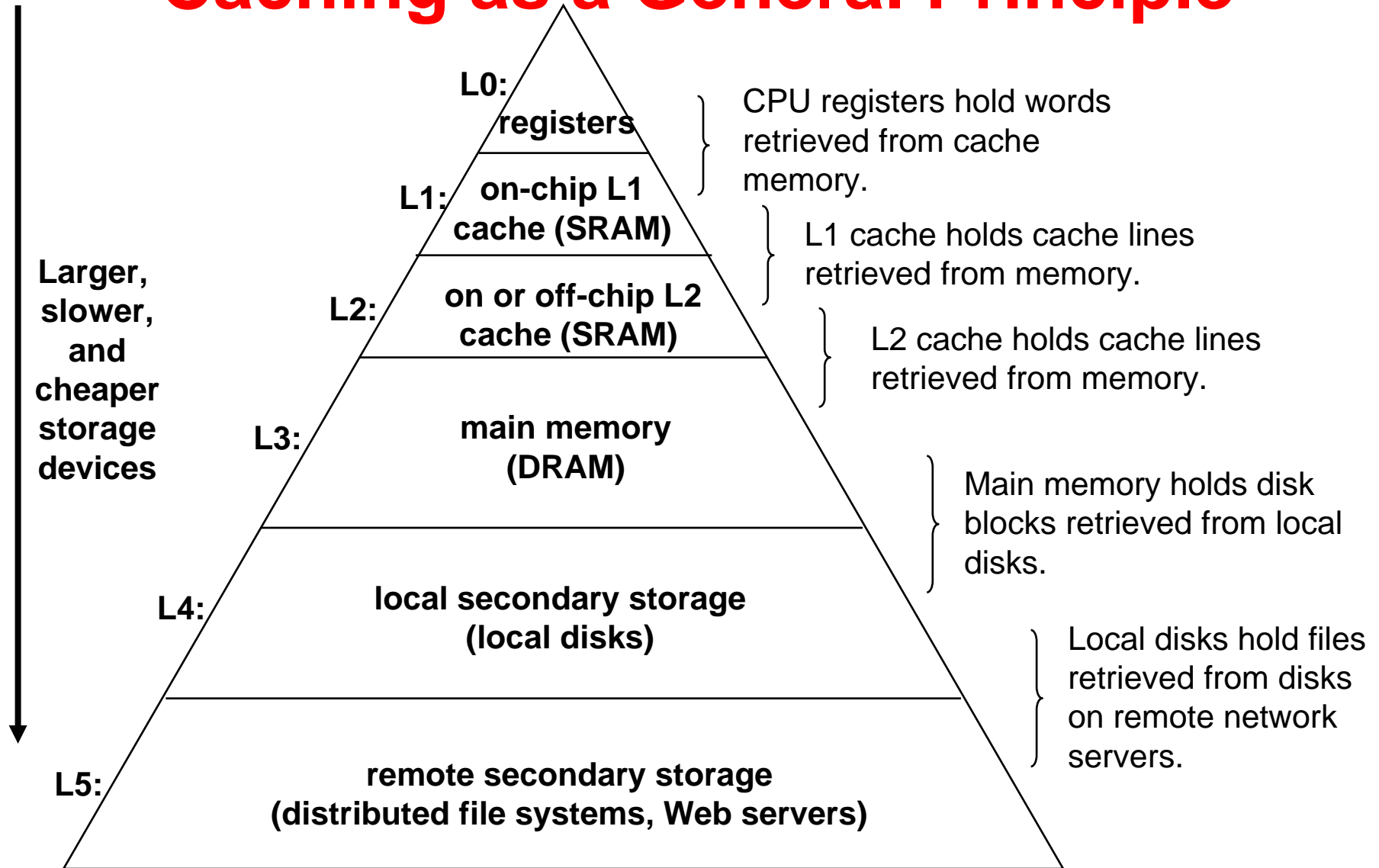
# I/O & Caches

- **Normal data and high performance processors**
  - **out of order reads and writes**
    - » writes usually deferred
    - » even reads can complete out of order
    - » reads can be retried
    - » many ordering options
      - all have some control complexity overhead however
- **I/O is a VERY different beast**
  - **often I/O reads and writes involve side-effects**
  - **hence**
    - » **I/O reads and writes MUST**
      - be done exactly once and in-order
    - » normal and I/O data policy is different
  - **allow non-cacheable reads and writes**
    - » available on all modern high performance processors

# 4 Questions & 3 Cache Organizations

	<b>Block Placement</b>	<b>Block Identification</b>	<b>Block Replacement</b>	<b>Write Strategy</b>
<b>Direct Mapped</b>	no choice	no choice but valid bit and tags must match	no choice	write-through & write-allocate write-back & (around OR allocate)
<b>N-way Set-Associative</b>	address set and choose way (invalid or LRU victim or random)	address set and then match tags and valid bit to confirm	LRU'ish or random	write-through & write-allocate write-back & (around OR allocate)
<b>Fully Associative</b>	choose any (random, LRU, invalid, ...)	find matching tags and valid	choose any (random, LRU, invalid, ...)	write-through & write-allocate write-back & (around OR allocate)

# Caching as a General Principle

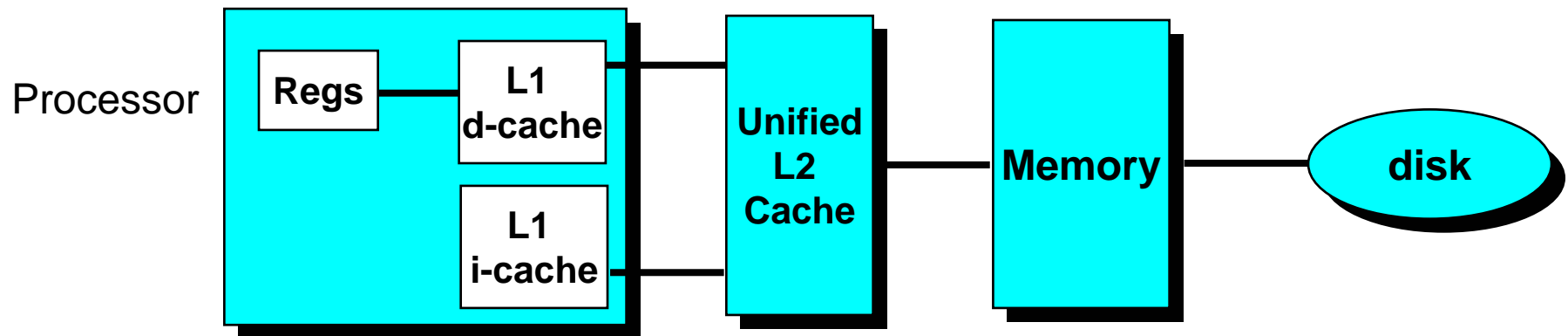


# Forms of Caching

Cache Type	What Cached	Where Cached	Latency (cycles)	Managed By
Registers	4-byte word	CPU Registers	1	Compiler
TLB	Address Translations	On-Chip TLB	1	Hardware
SRAM	32-byte block	On-Chip L1	2-3	Hardware
SRAM	32-byte block	Off-Chip L2	10-20	Hardware
Virtual Memory	4-KB page	Main Memory	100+	MMU+OS
Buffered Files	File Buffer	Main Memory	100+	OS
Network File Cache	Parts of Files	Processor Disk	10,000,000	AFS Client
Browser Cache	Web Pages	Processor Disk	10,000,000	Browser
Web Cache	Web Pages	Server Disks	1,000,000,000	Akamai Server

# Multi-Level Caches

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

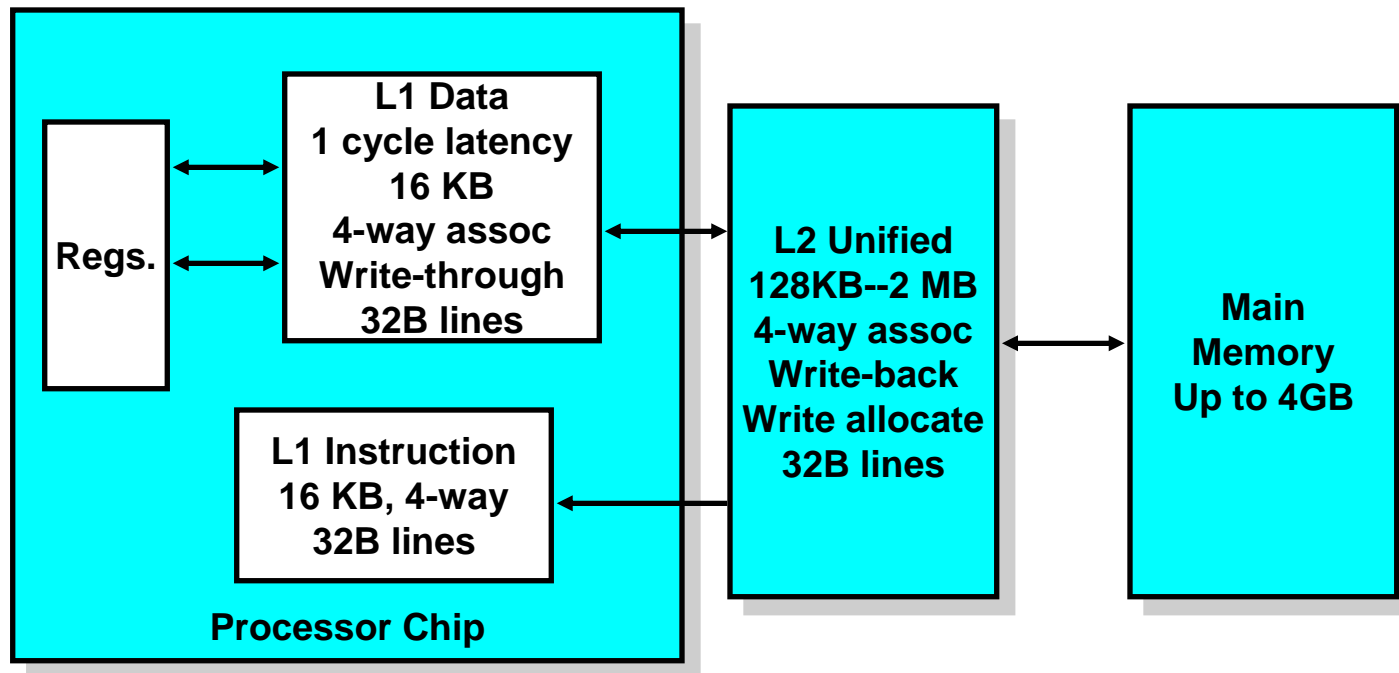


size:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	30 GB
speed:	3 ns	3 ns	6 ns	60 ns	8 ms
\$/Mbyte:			\$100/MB	\$1.50/MB	\$0.05/MB
line size:	8 B	32 B	32 B	8 KB	

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:
  - 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 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

```
/* 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

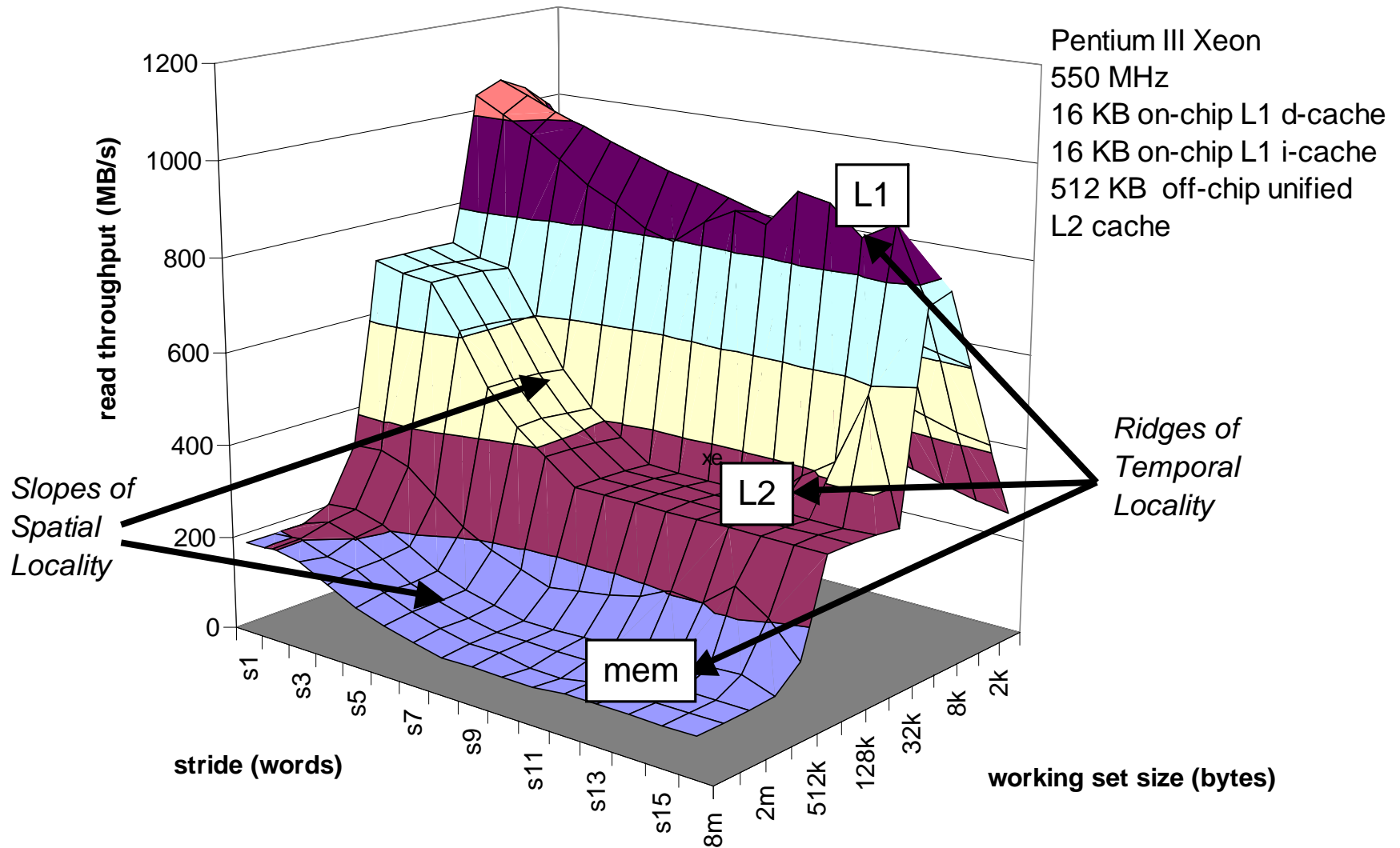
```
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10) /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23) /* ... up to 8 MB */
#define MAXSTRIDE 16 /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

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 */

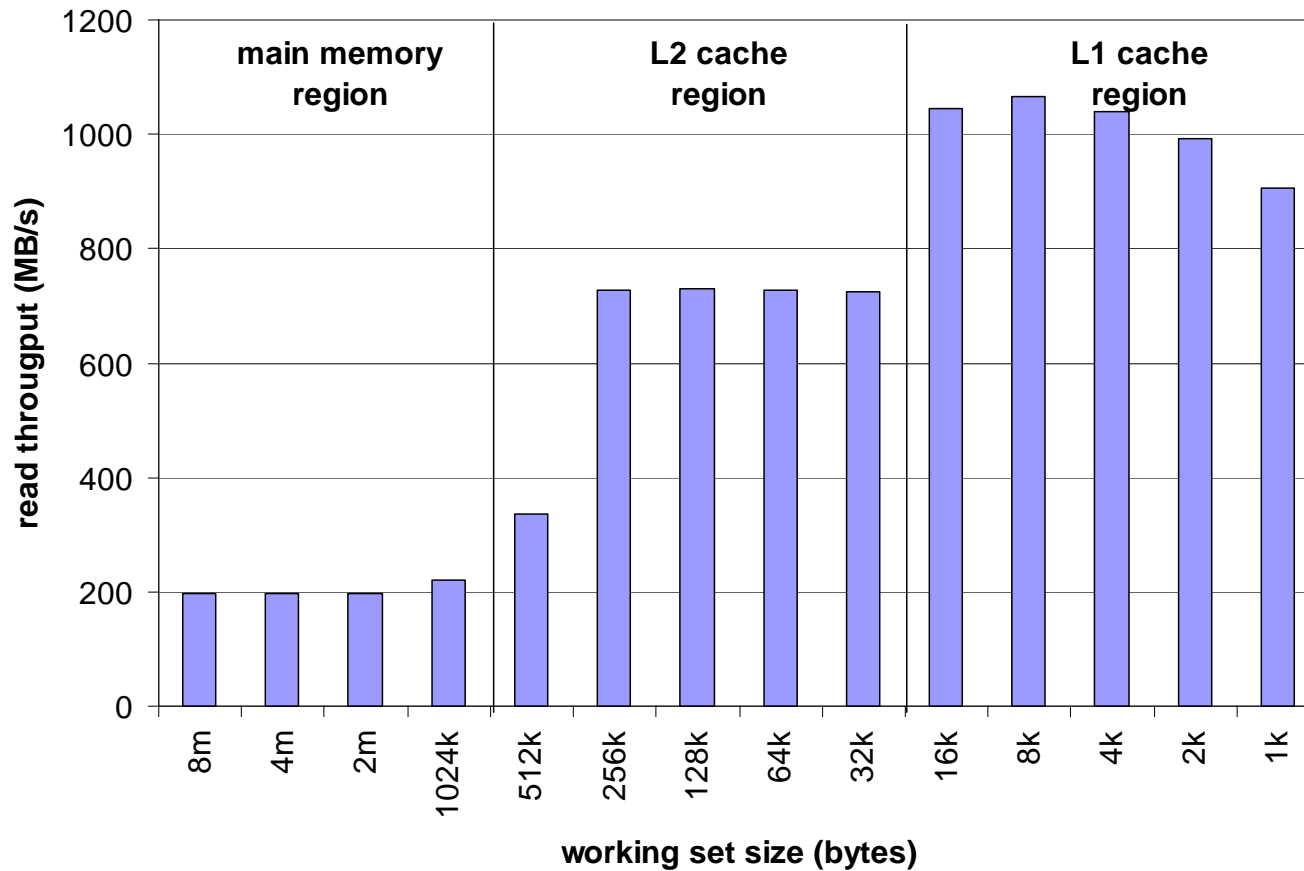
    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0); /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");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

# The Memory Mountain



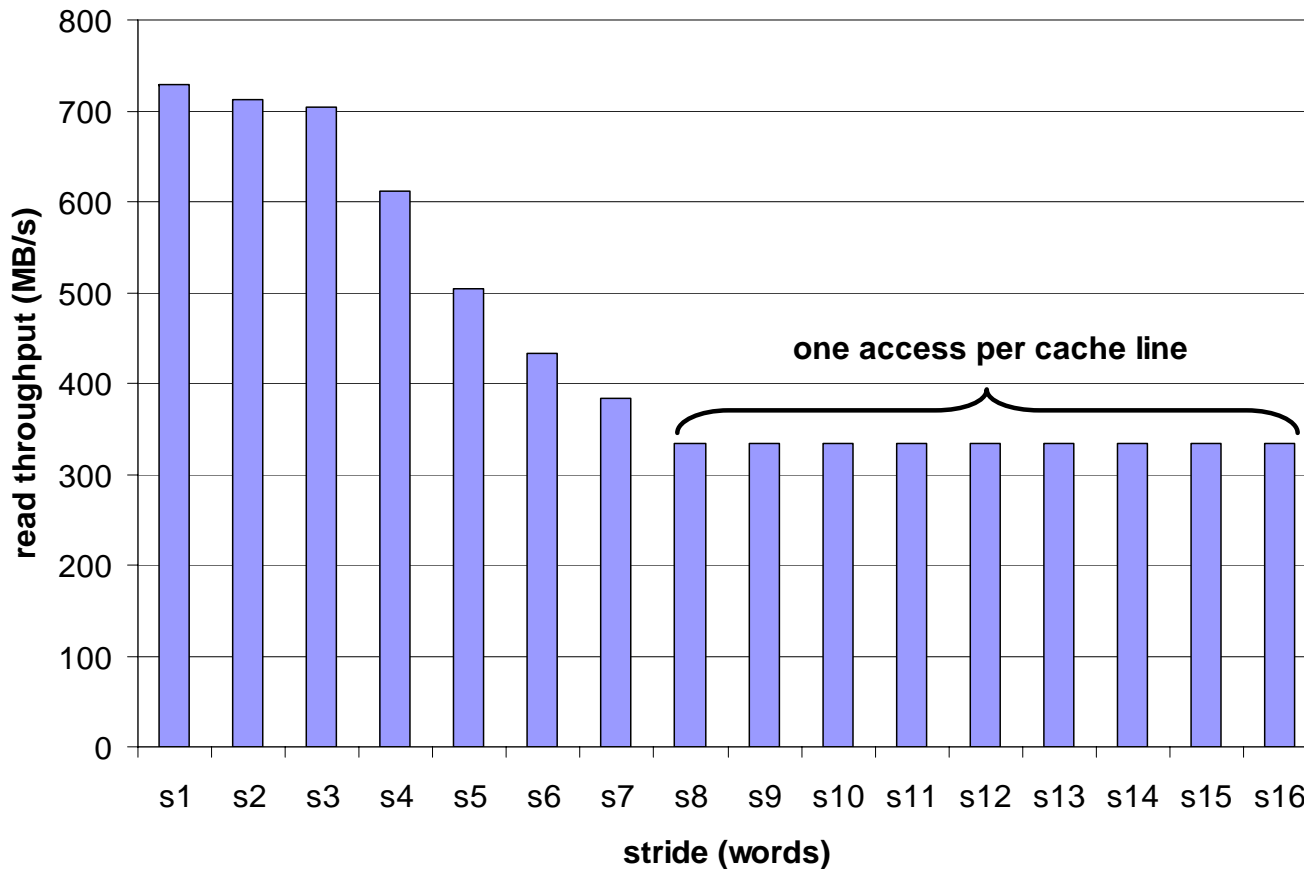
# Ridges of Temporal Locality

- 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.



# 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

- Description:

- Multiply N x N matrices

- $O(N^3)$  total operations

- Accesses

- » N reads per source element

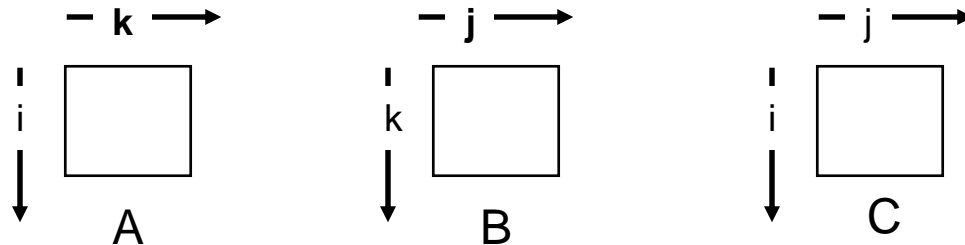
- » N values summed per destination

- but may be able to hold in register

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

# 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



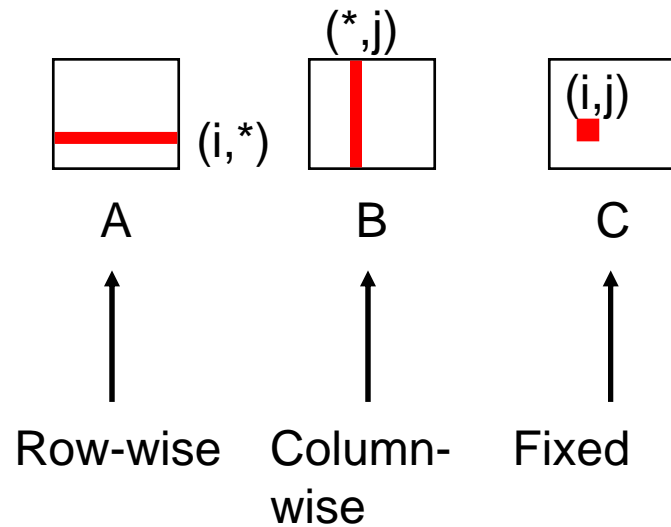
# Layout of C Arrays in Memory (review)

- **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;  
  }  
}
```

Inner loop:



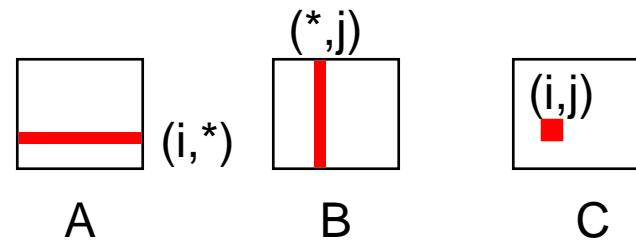
- Misses per Inner Loop Iteration:

<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:



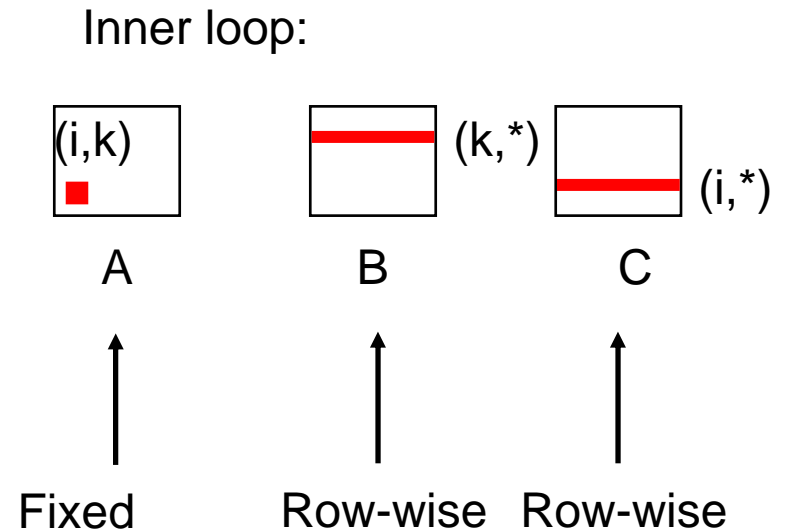
Row-wise      Column-wise      Fixed

- Misses per Inner Loop Iteration:

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



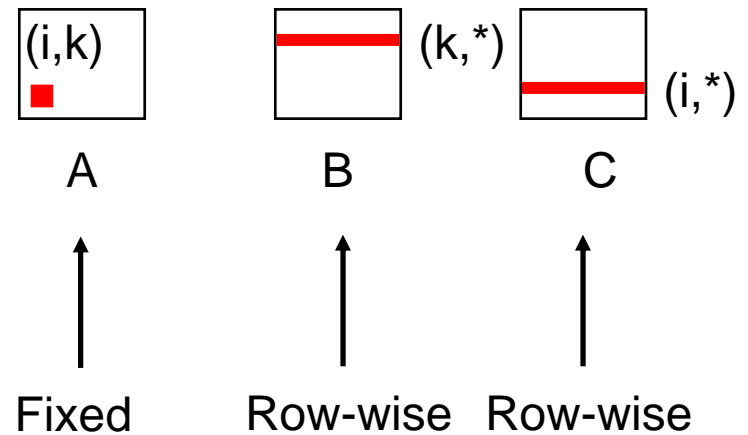
- Misses per Inner Loop Iteration:

<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:



- Misses per Inner Loop Iteration:

A  
0.0

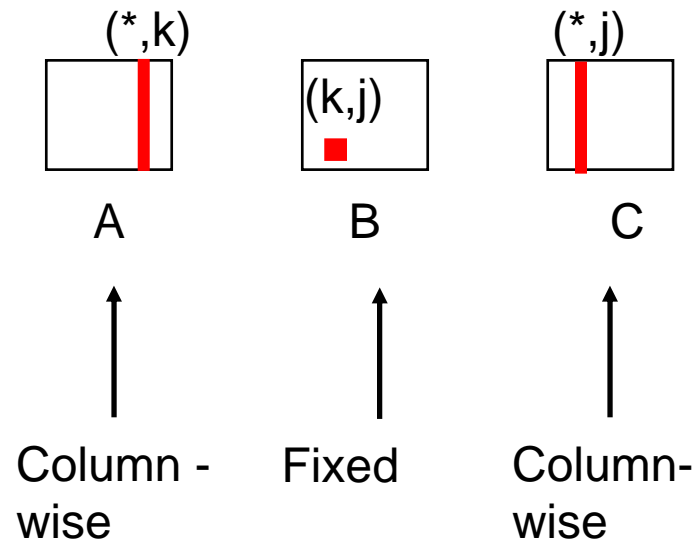
B  
0.25

C  
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;  
  }  
}
```

Inner loop:

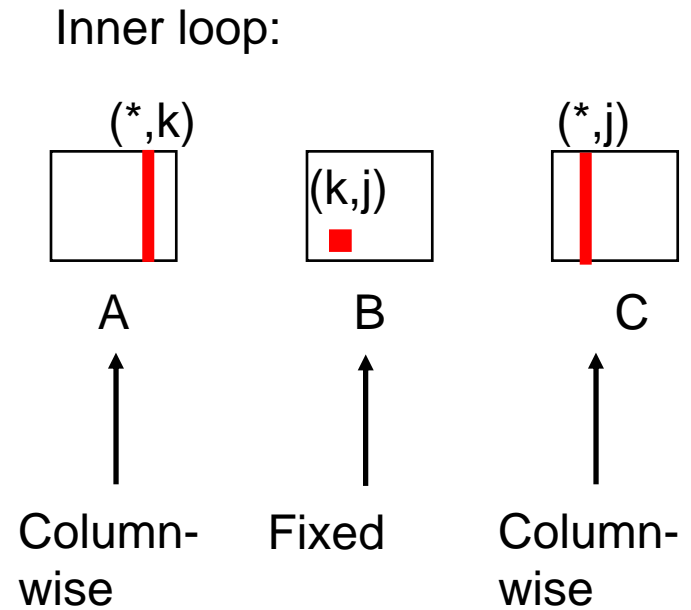


- Misses per Inner Loop Iteration:

<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;
  }
}
```



- Misses per Inner Loop Iteration:

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

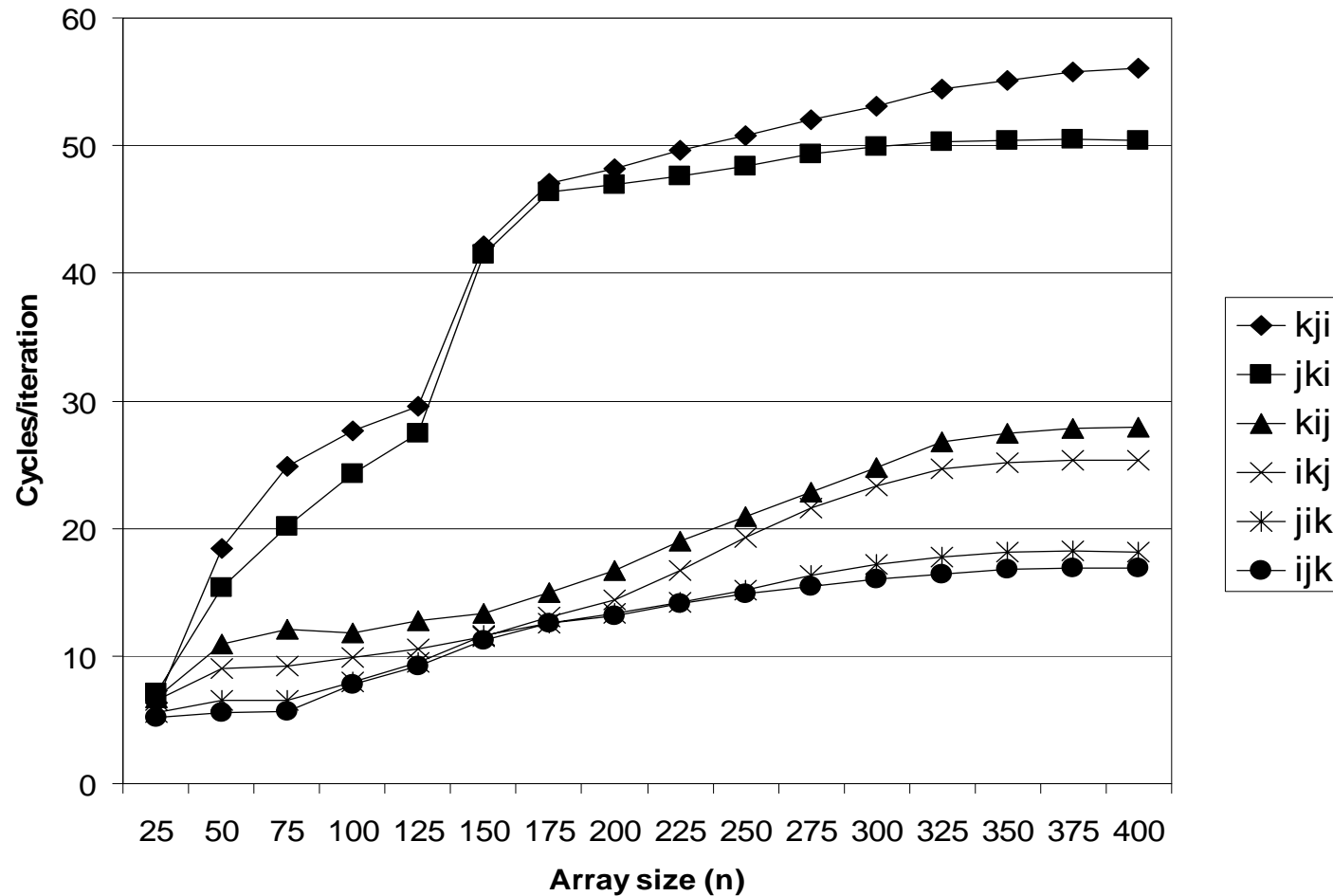
## jki (& kji):

- 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;  
  }  
}
```

# 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

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

Key idea: Sub-blocks (i.e.,  $A_{xy}$ ) can be treated just like scalars.

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

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

# Blocked Matrix Multiply (bijk)

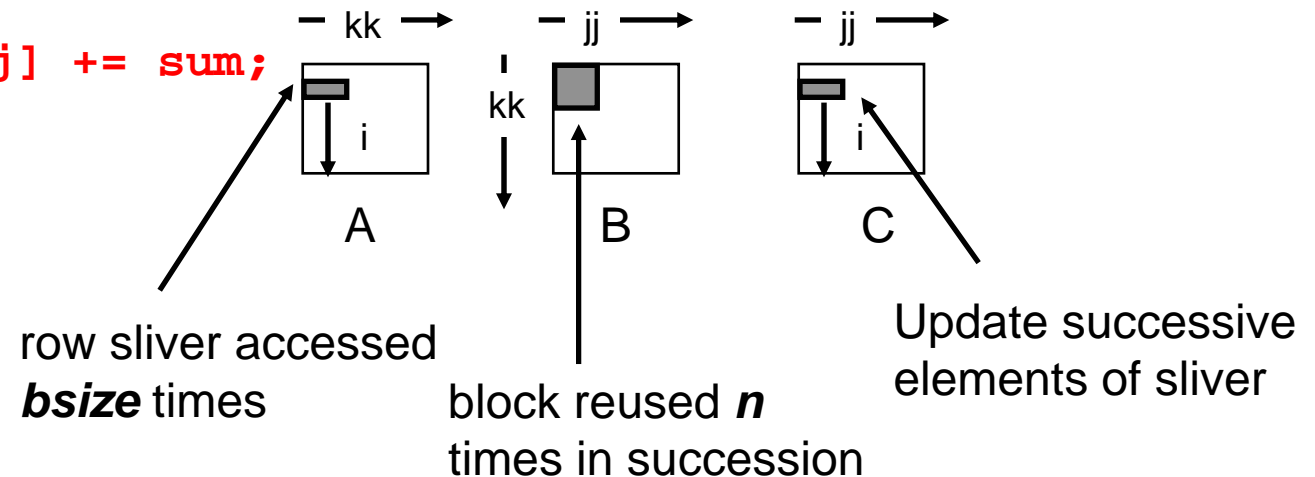
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum +=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

# Blocked Matrix Multiply Analysis

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

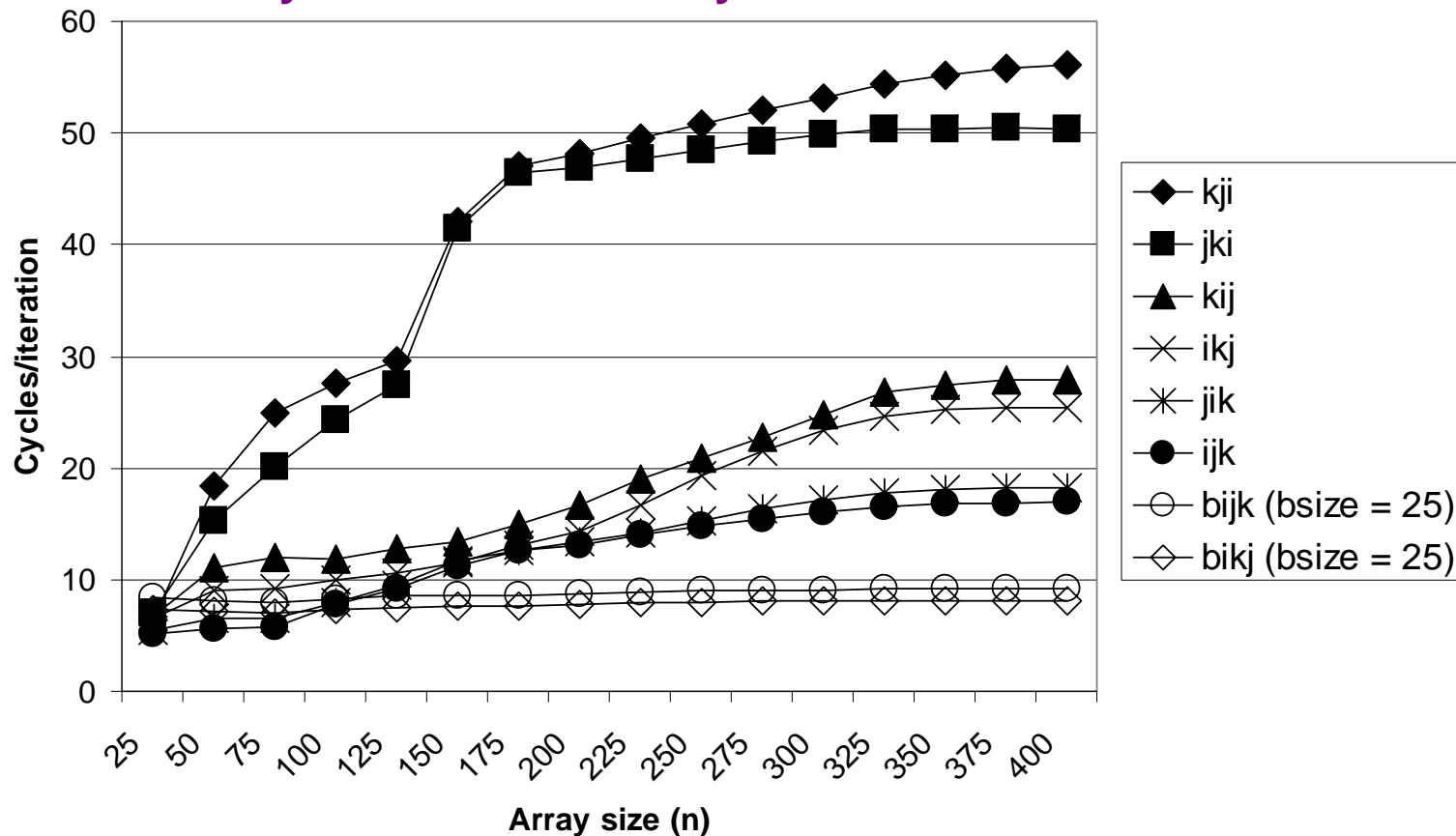
```
for (i=0; i<n; i++) {  
    for (j=jj; j < min(jj+bsize,n); j++) {  
        sum = 0.0  
        for (k=kk; k < min(kk+bsize,n); k++) {  
            sum += a[i][k] * b[k][j];  
        }  
        c[i][j] += sum;  
    }  
}
```

Innermost  
Loop Pair



# Pentium Blocked Matrix Multiply 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)