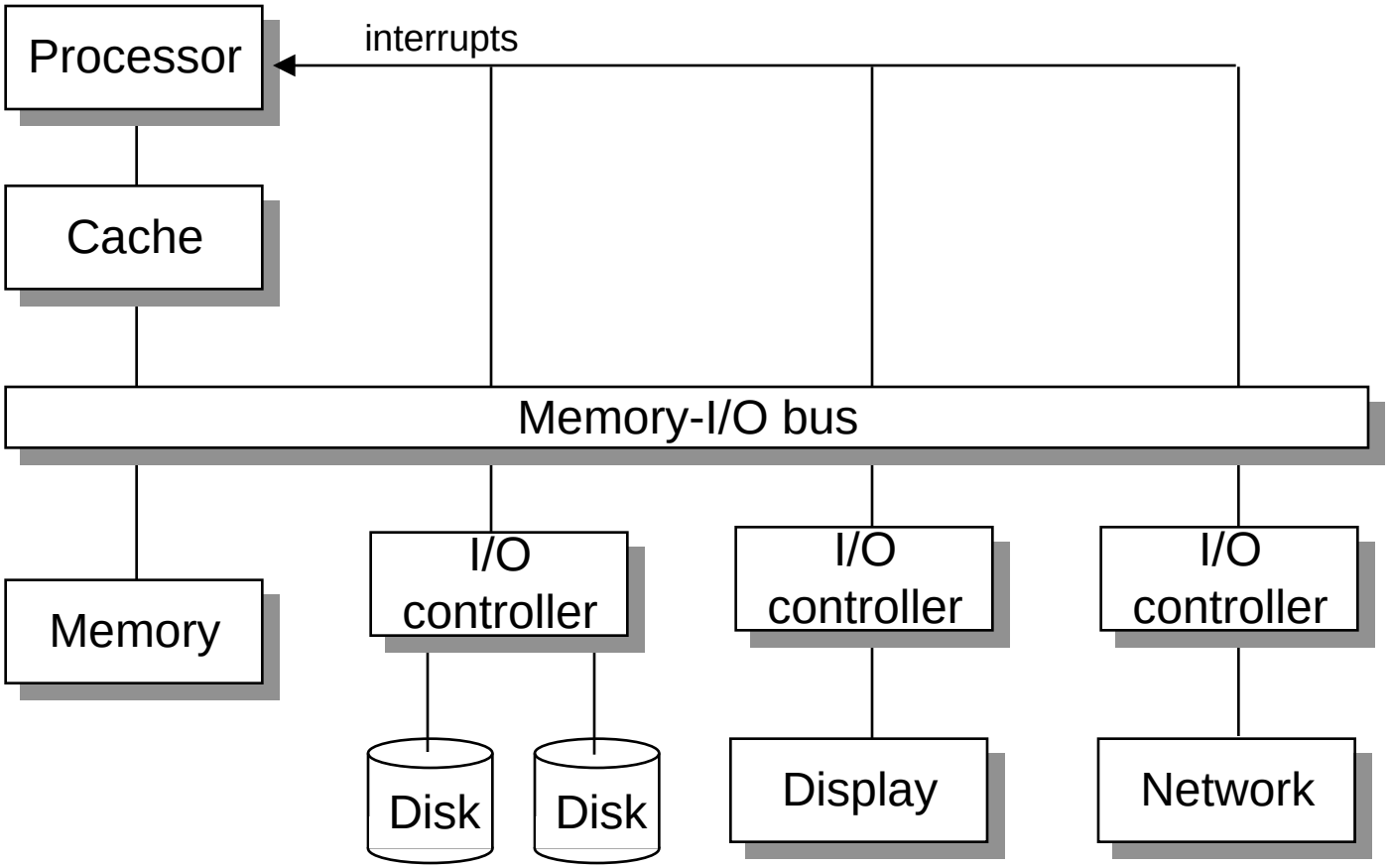
15-740

Caches
Oct. 8, 1998

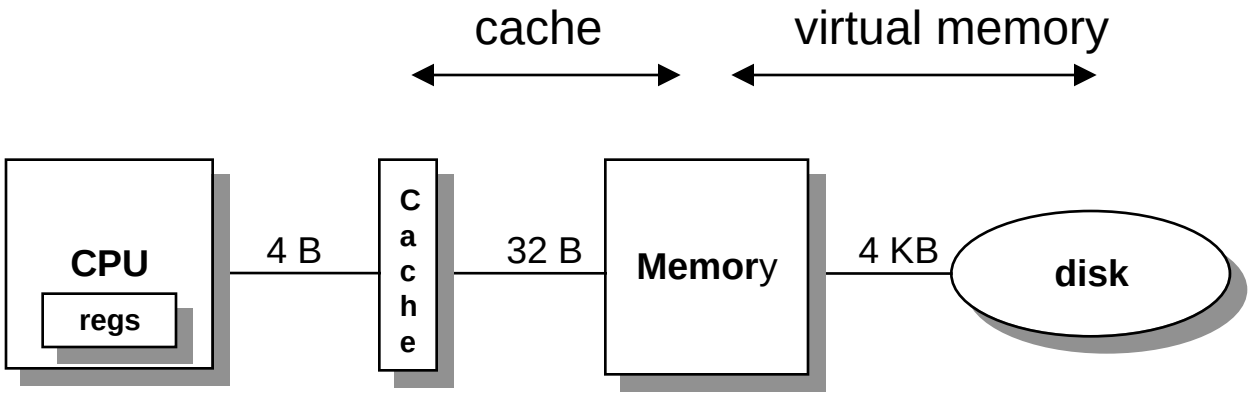
Topics

- Memory Hierarchy
- Cache Design
- Optimizing Program Performance

Computer System



Levels in a typical memory hierarchy



	register reference	cache reference	memory reference	disk memory reference
size:	200 B	32KB -- 4MB	128 MB	10GB
speed:	5 ns	6 ns	100 ns	10 ms
$/Mbyte:		$256/MB	$2/MB	$0.10/MB
block size:	4 B	16 B	4 KB	

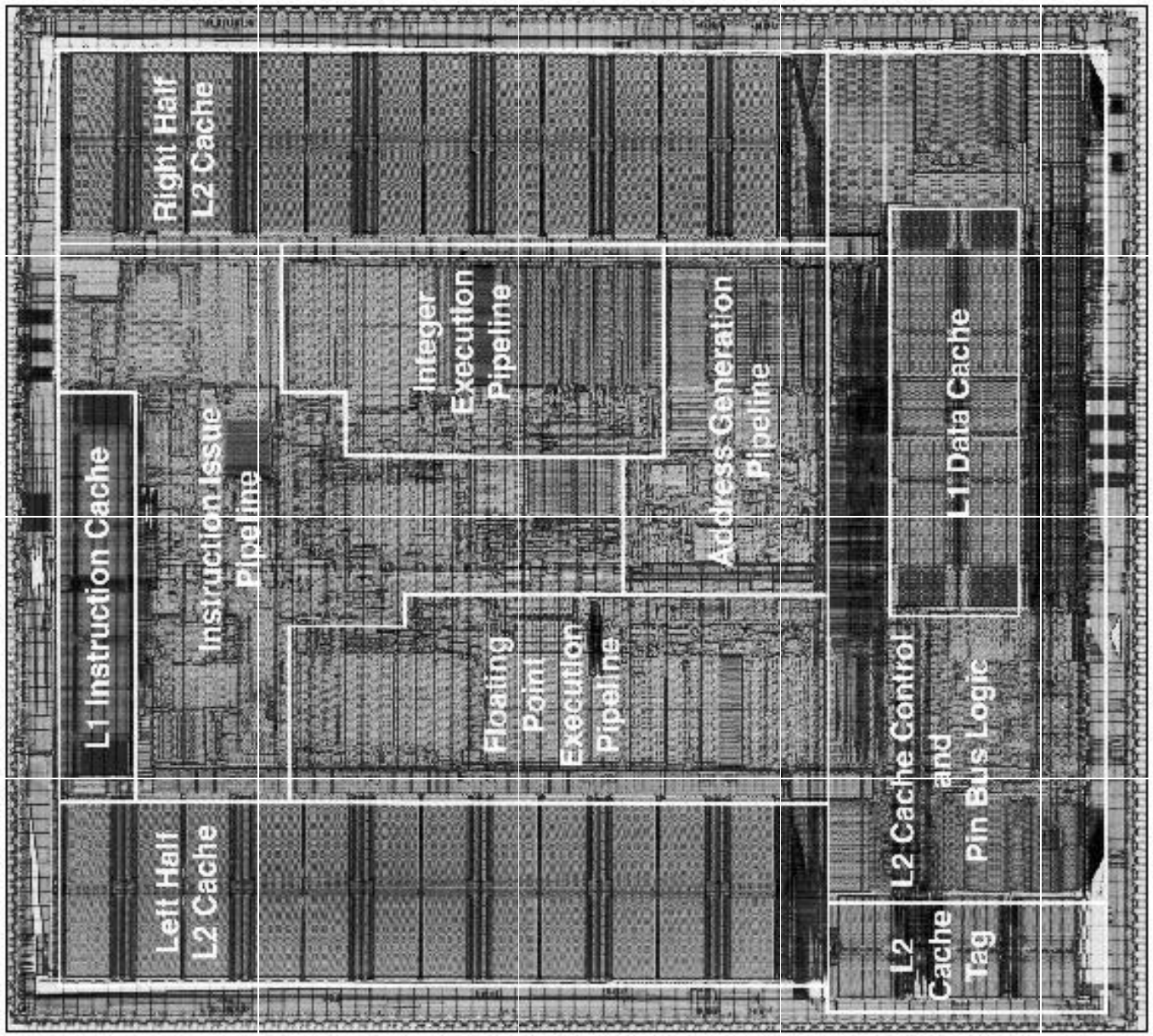
larger, slower, cheaper →

Alpha 21164 Chip Photo

- Microprocessor Report 9/12/94

Caches:

- L1 data
- L1 instruction
- L2 unified
- TLB
- Branch history

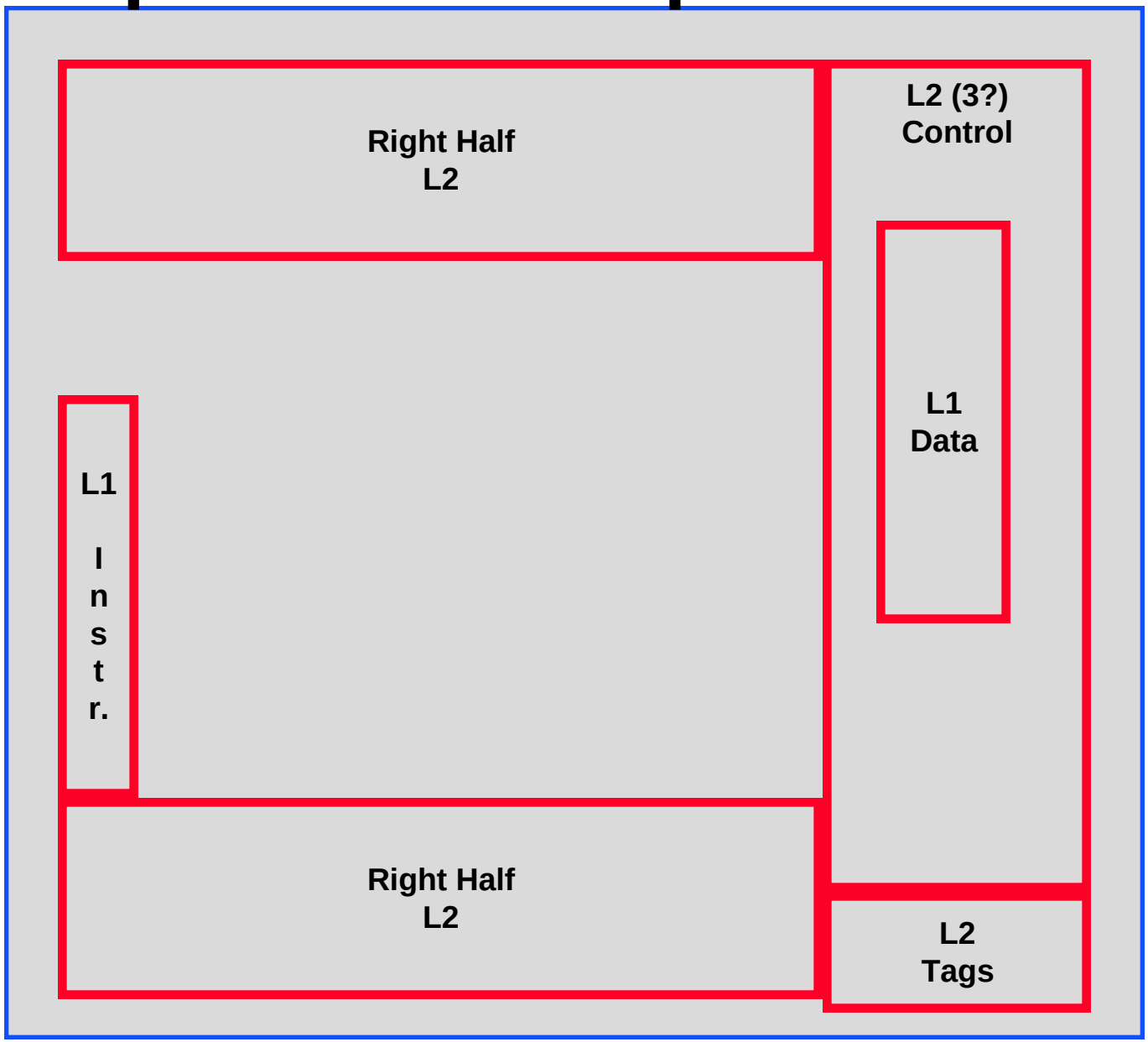


Alpha 21164 Chip Caches

- Microprocessor Report 9/12/94

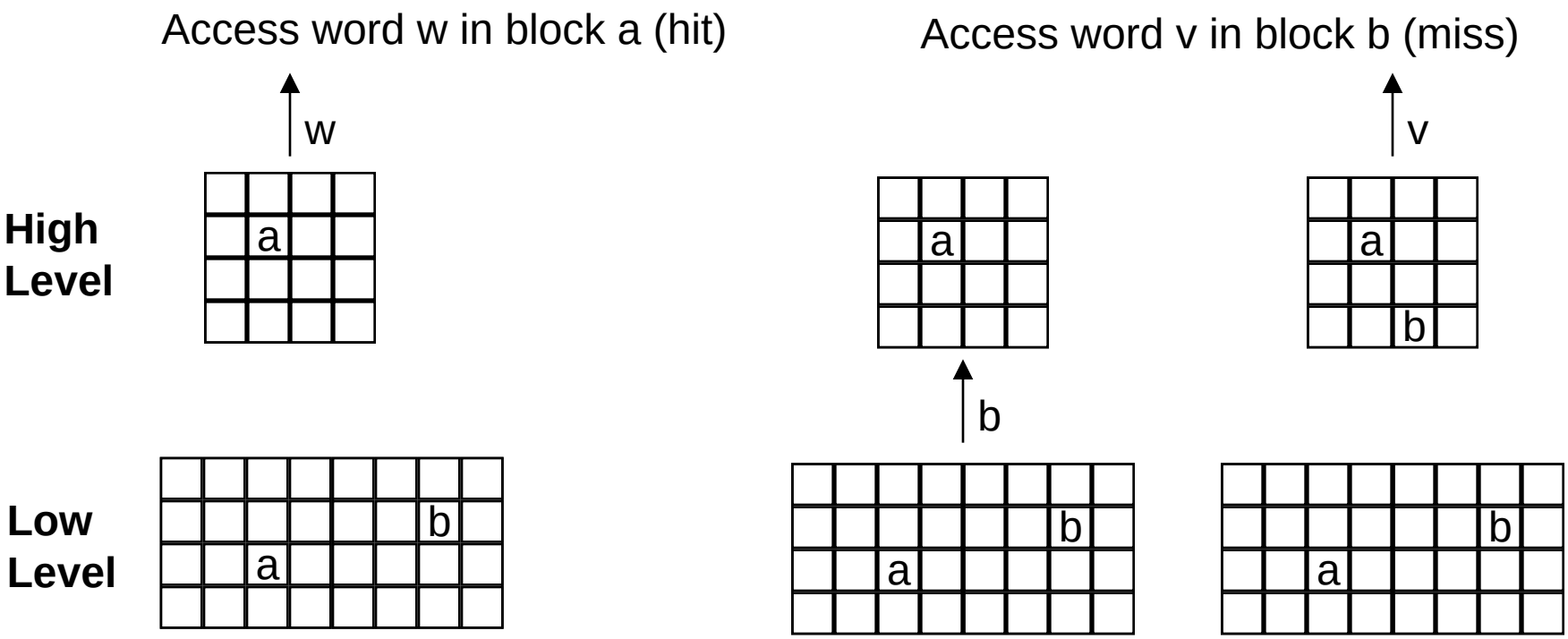
Caches:

- L1 data
- L1 instruction
- L2 unified
- TLB
- Branch history



Accessing data in a memory hierarchy

- **Between any two levels, memory divided into blocks.**
- **Data moves between levels on demand, in block-sized chunks.**
- **Upper-level blocks a subset of lower-level blocks.**



Locality of reference

Principle of Locality

- **Programs tend to reuse data and instructions near those they have used recently.**
- ***Temporal locality:*** **recently referenced items are likely to be referenced in the near future.**
- ***Spatial locality:*** **items with nearby addresses tend to be referenced close together in time.**

Locality in Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

- **Data**
  - Reference array elements in succession (spatial)
- **Instruction**
  - Reference instructions in sequence (spatial)
  - Cycle through loop repeatedly (temporal)

Key questions for caches

Q1: Where should a block be placed in the cache? (block placement)

Q2: How is a block found in the cache? (block identification)

Q3: Which block should be replaced on a miss? (block replacement)

Q4: What happens on a write? (write strategy)

Address spaces

An n-bit address defines an address space of 2^n items: $0,...,2^{n-1}$.

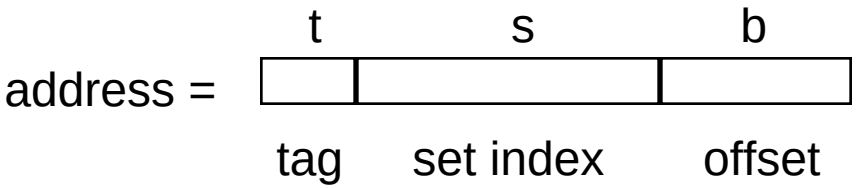
00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Address space for n=5

Partitioning address spaces

Key idea: partitioning the address bits partitions the address space.

In general, an address partitioned into sets of t (tag), s (set index), and b (block offset) bits, e.g.,



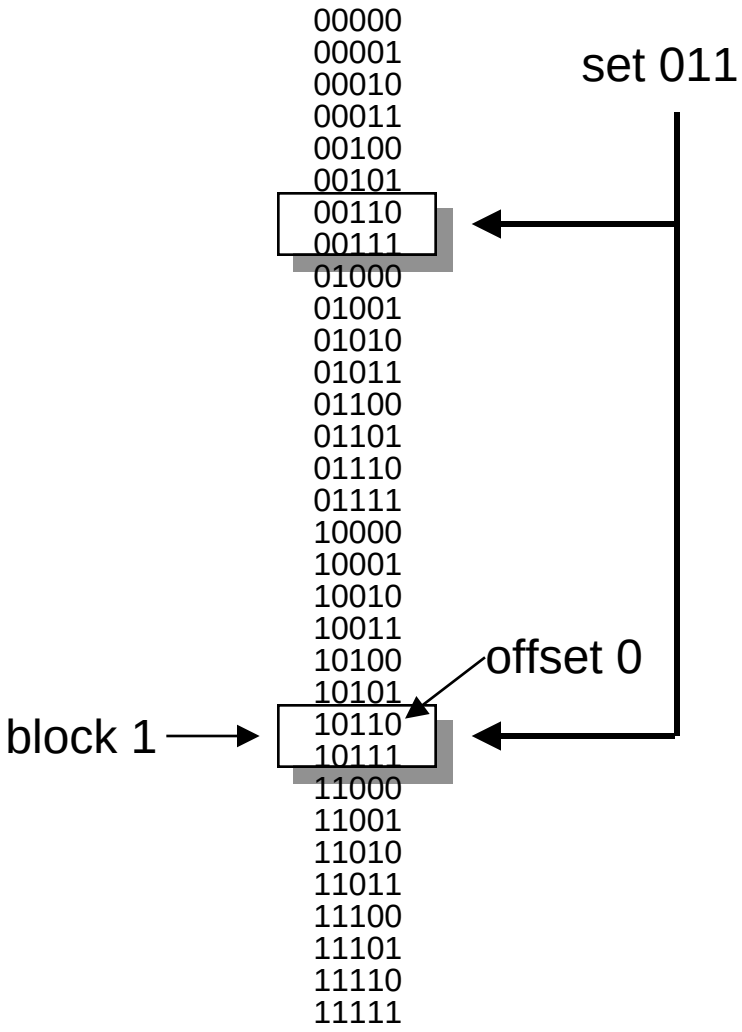
belongs to one of 2^s equivalence classes (sets), where each set consists of 2^t blocks of addresses, and each block consists of 2^b addresses.

The s bits uniquely identify an equivalence class.
The t bits uniquely identify each block in the equivalence class.
The b bits define the offset of an address within a block (block offset).

Partitioning address spaces

t=1	s=3	b=1
1	011	0

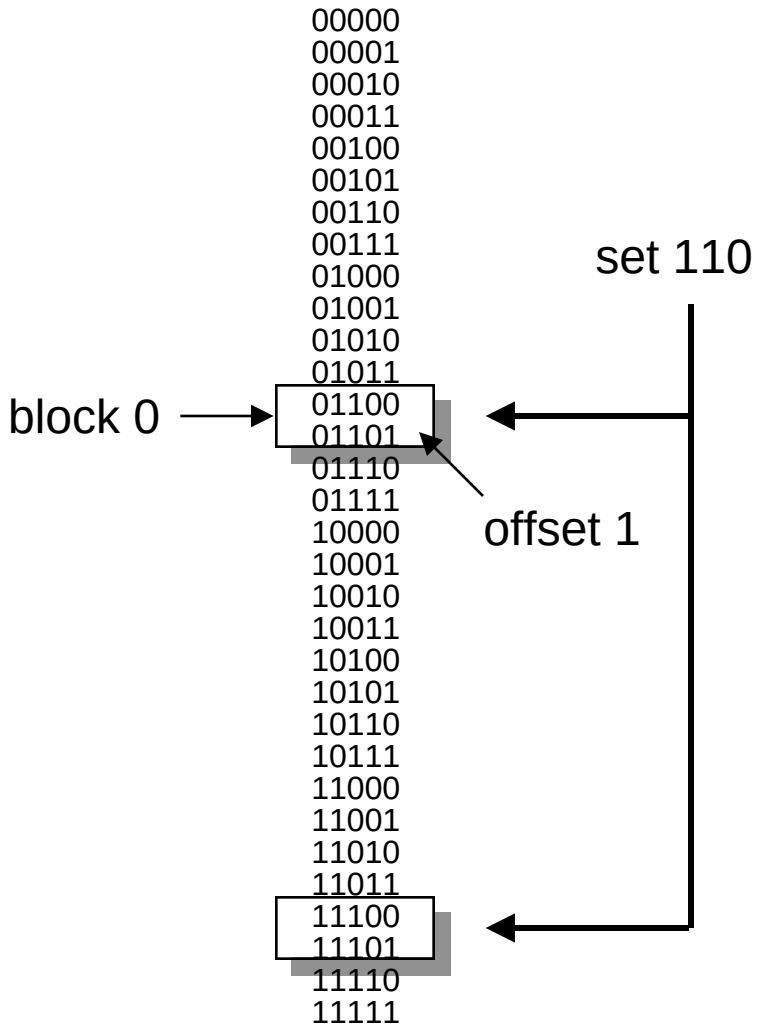
2^s = 8 sets of blocks
2^t = 2 blocks/set
2^b = 2 addresses/block.



Partitioning address spaces

t=1	s=3	b=1
0	110	1

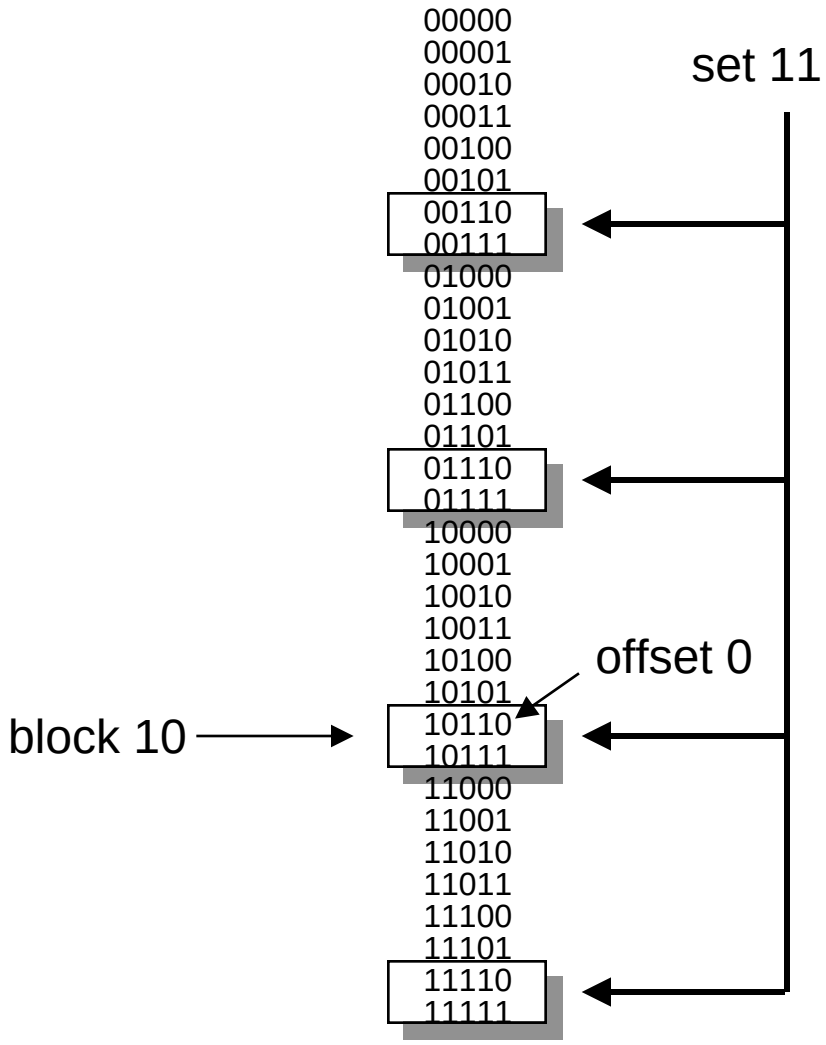
2^s = 8 sets of blocks
2^t = 2 blocks/set
2^b = 2 addresses/block.



Partitioning address spaces

t=2	s=2	b=1
10	11	0

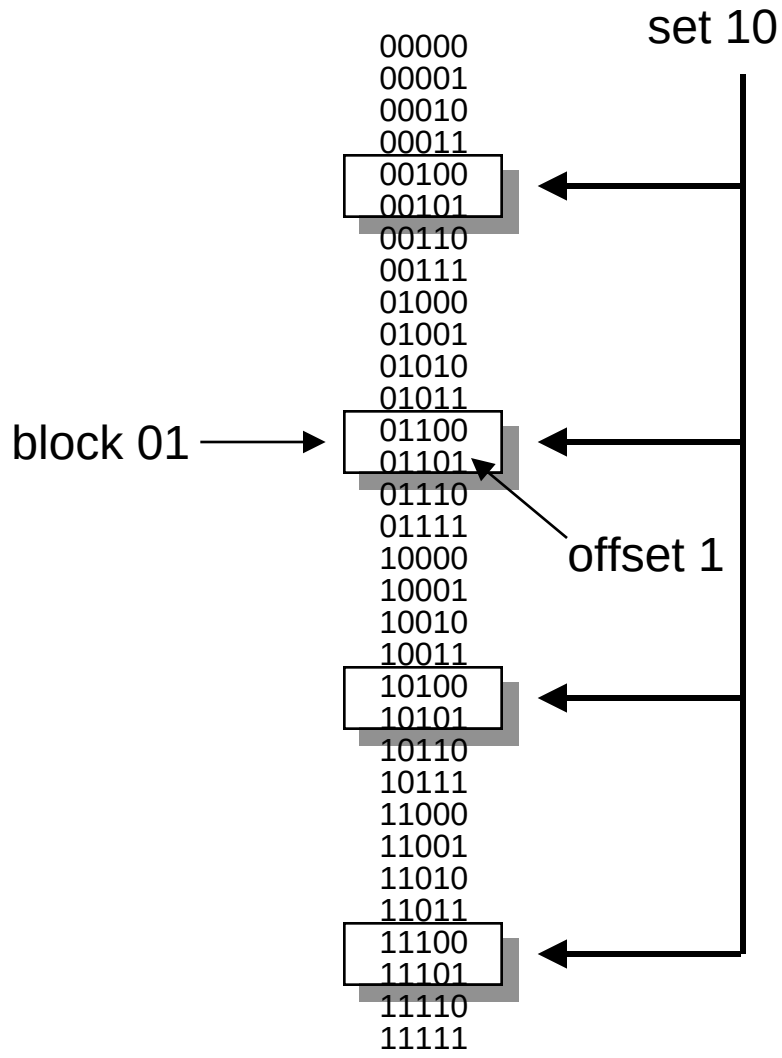
2^s = 4 sets of blocks
2^t = 4 blocks/set
2^b = 2 addresses/block.



Partitioning address spaces

t=2	s=2	b=1
01	10	1

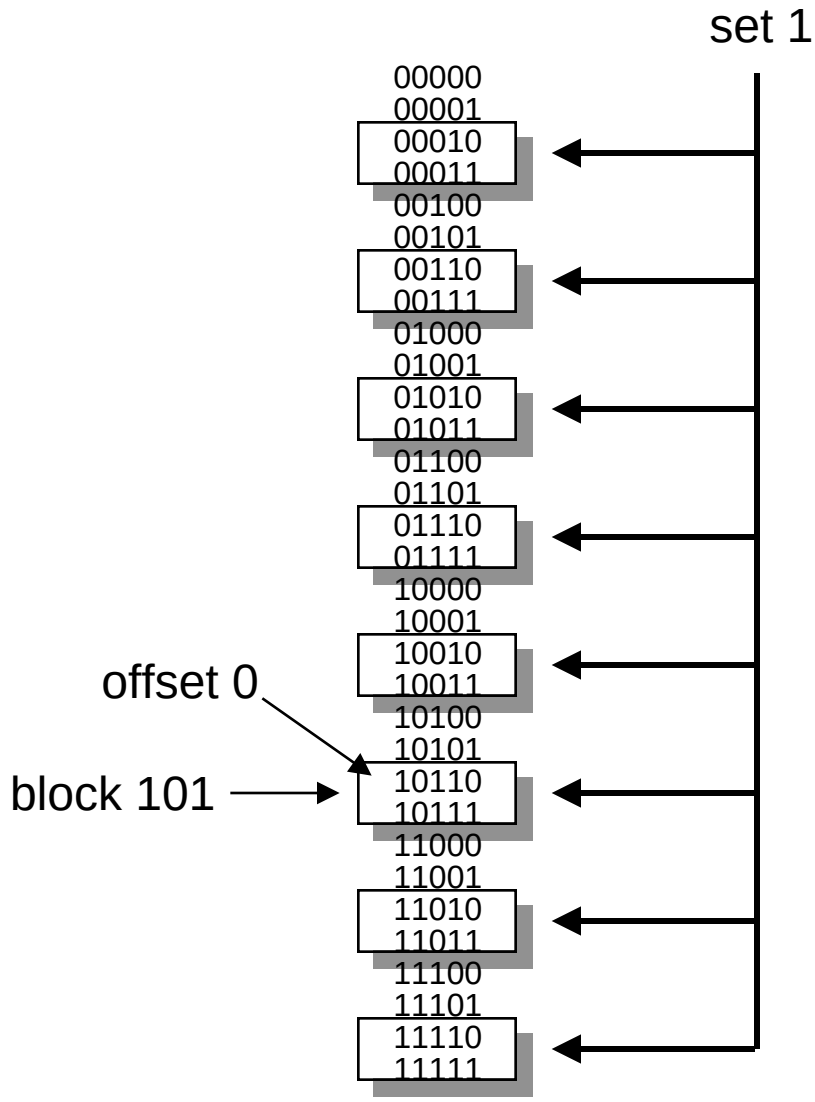
2^s = 4 sets of blocks
2^t = 4 blocks/set
2^b = 2 addresses/block.



Partitioning address spaces

t=3	s=1	b=1
101	1	0

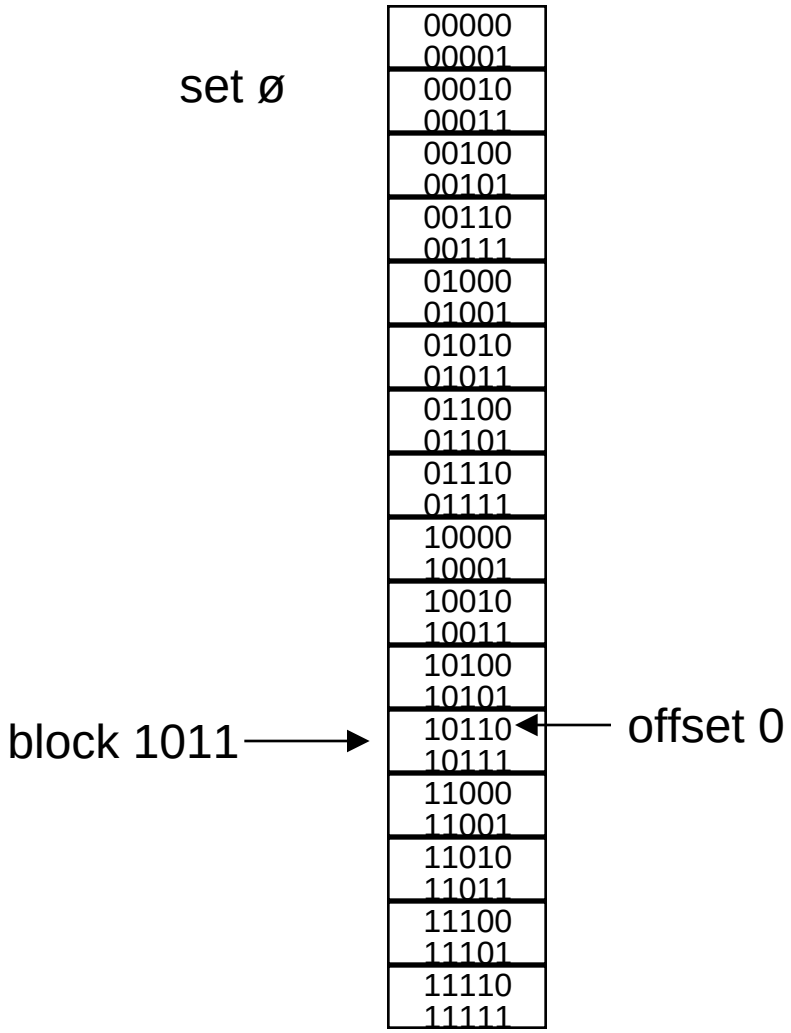
2^s = 2 sets of blocks
2^t = 8 blocks/set
2^b = 2 addresses/block.



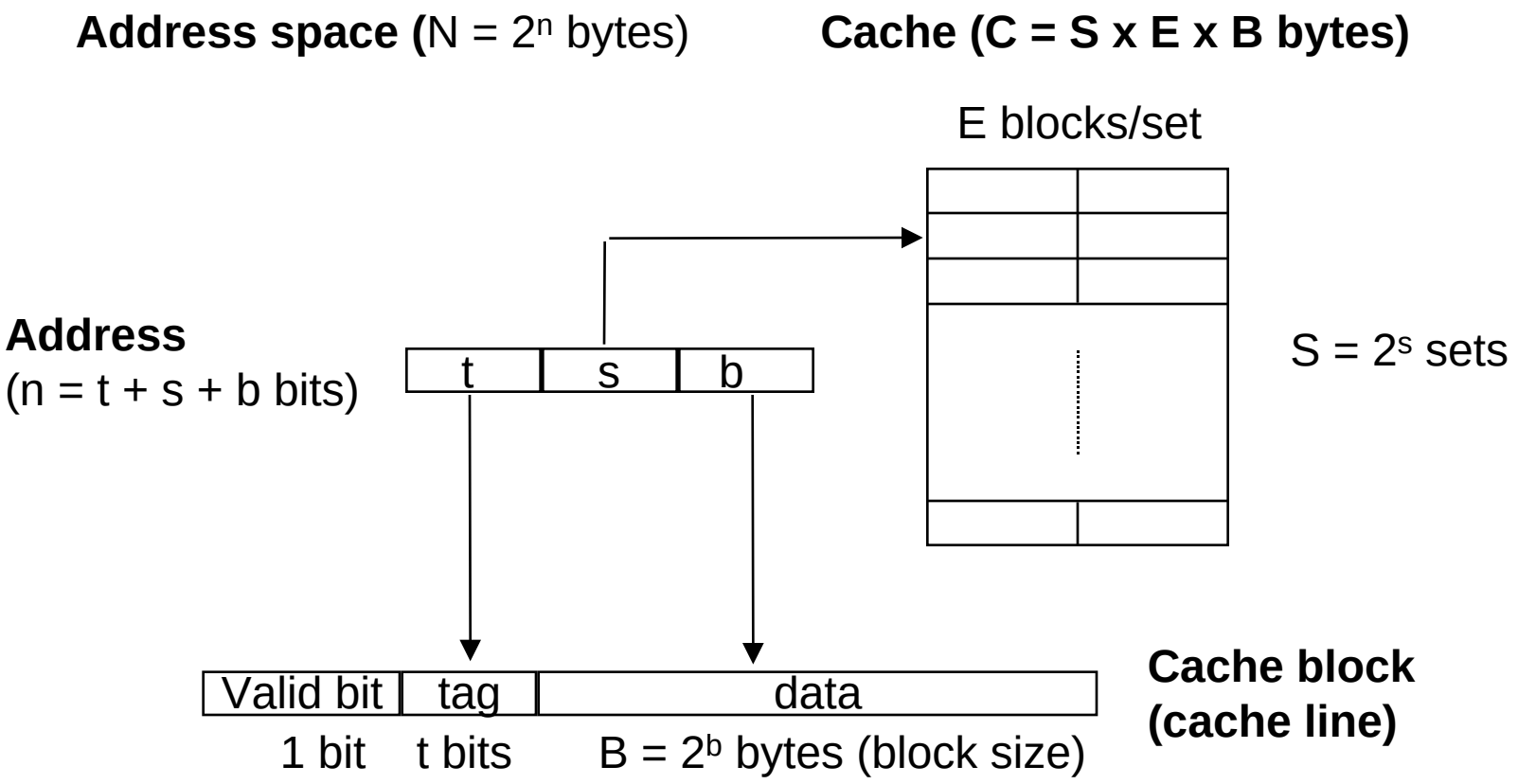
Partitioning address spaces

t=4	s=0	b=1
1011		0

2^s = 1 set of blocks
2^t = 16 blocks/set
2^b = 2 addresses/block.



Basic cache organization



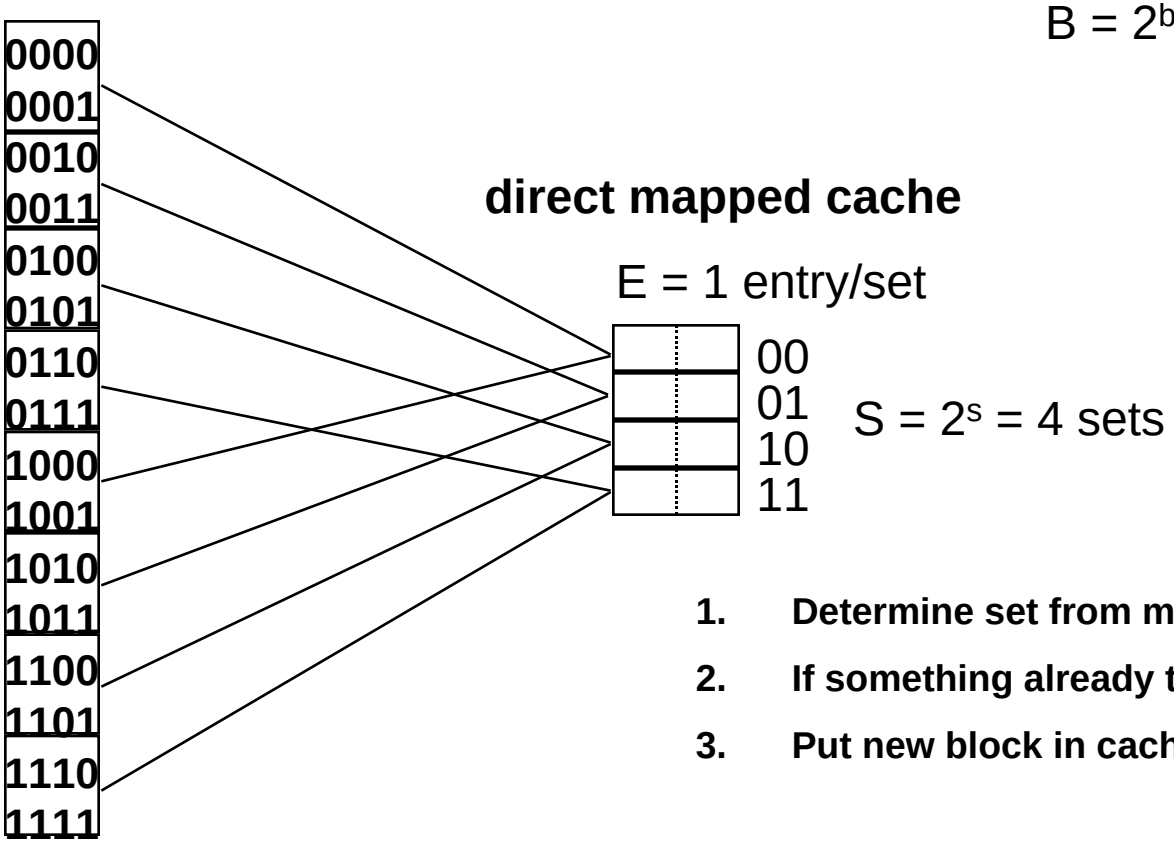
E: Describes *associativity*: how many blocks in set can reside in cache simultaneously
Assume $E < 2^t$

Direct mapped cache (E = 1)

N = 16 byte addresses (n=4)

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

cache size:
C = 8 data bytes
line size:
B = 2^b = 2 bytes/line



1. **Determine set from middle bits**
2. **If something already there, knock it out**
3. **Put new block in cache**

Direct Mapped Cache Simulation

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

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

Address trace (reads):

0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

0 [0000] (miss)

(1)

v	tag	data
1	0	m[1] m[0]

13 [1101] (miss)

(2)

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

8 [1000] (miss)

(3)

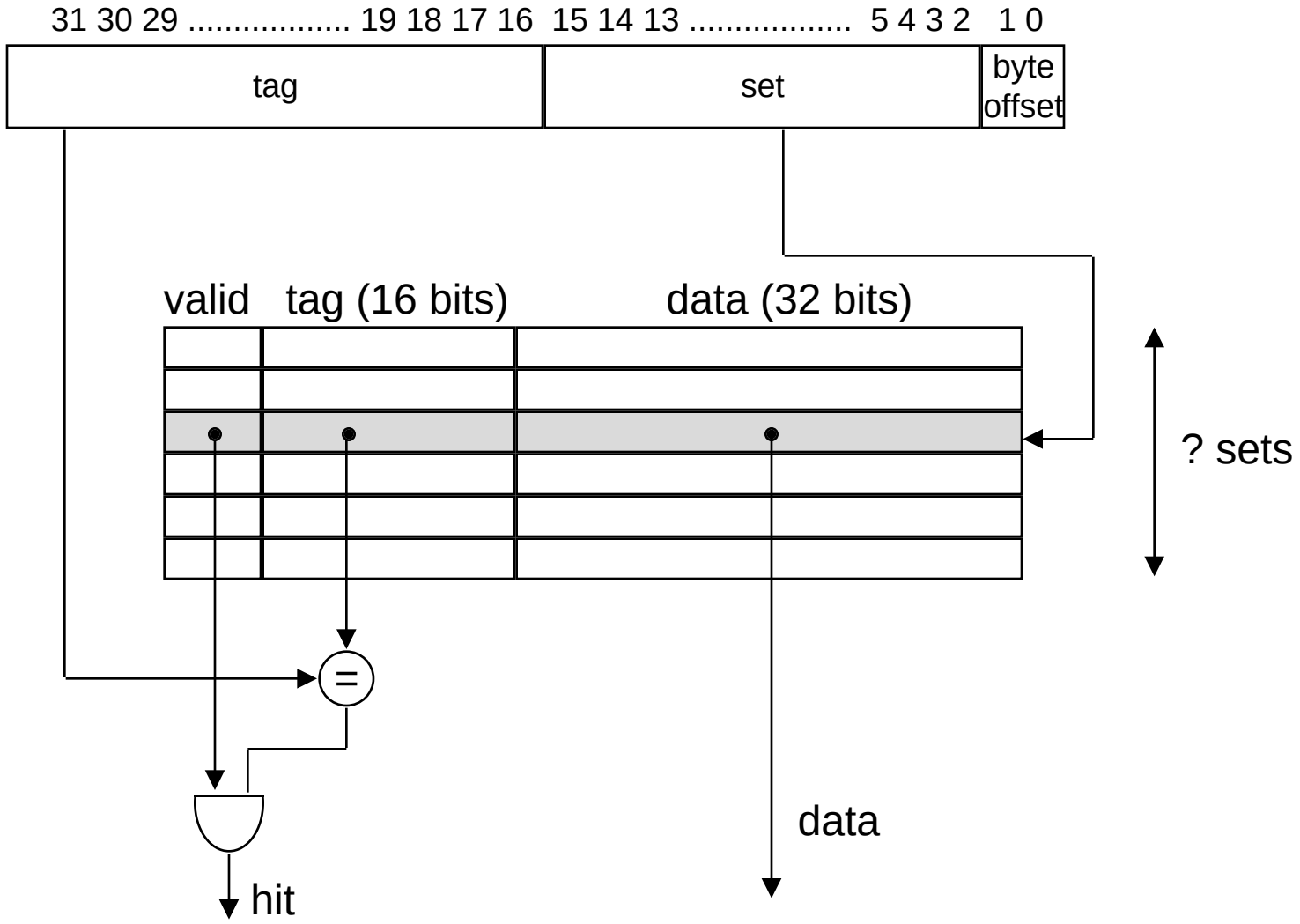
v	tag	data
1	1	m[9] m[8]

0 [0000] (miss)

(4)

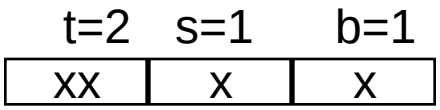
v	tag	data
1	0	m[1] m[0]
1	1	m[13] m[12]

Direct Mapped Cache Implementation (DECStation 3100)

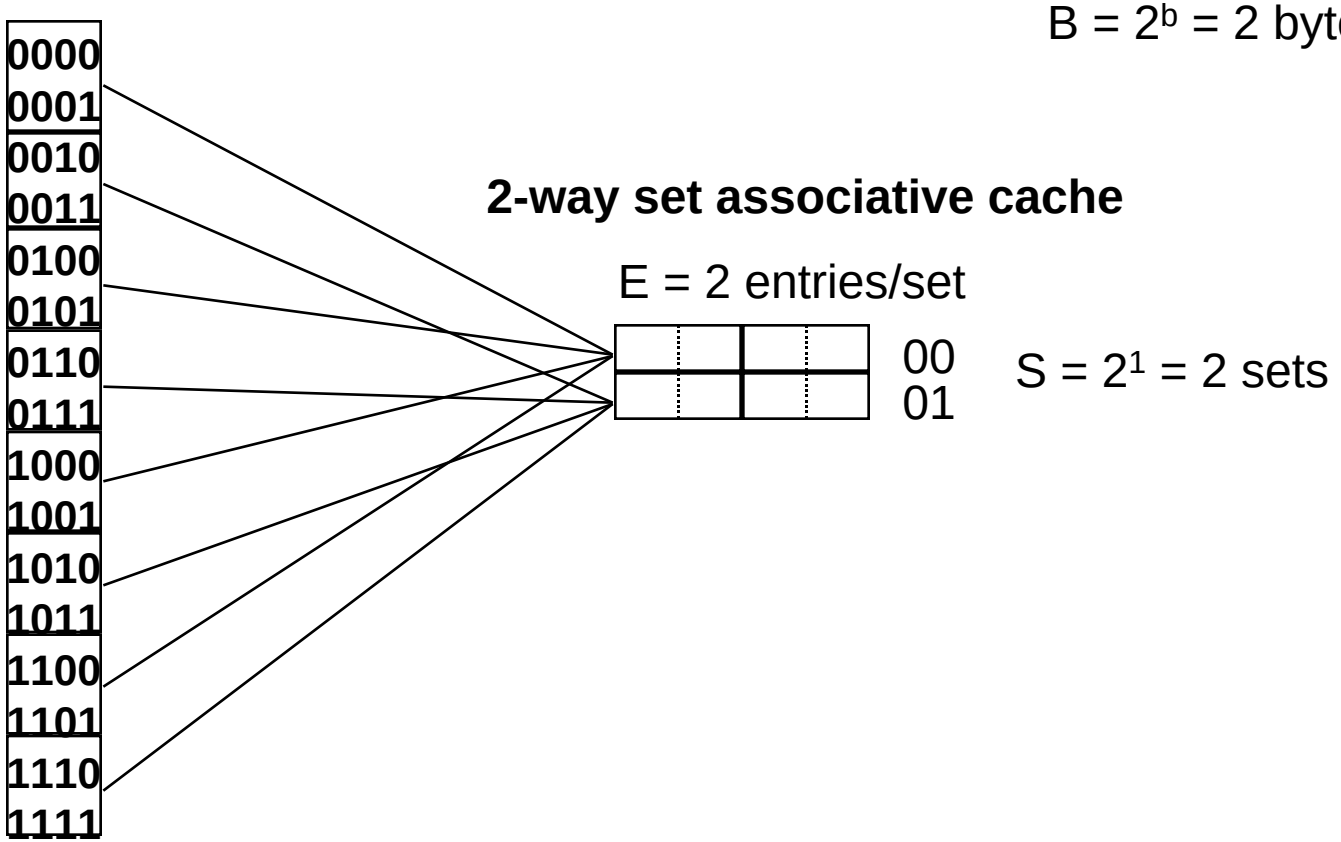


E-way Set-Associative Cache

N = 16 addresses (n=4)



Cache size:
C = 8 data bytes
Line size:
B = 2^b = 2 bytes



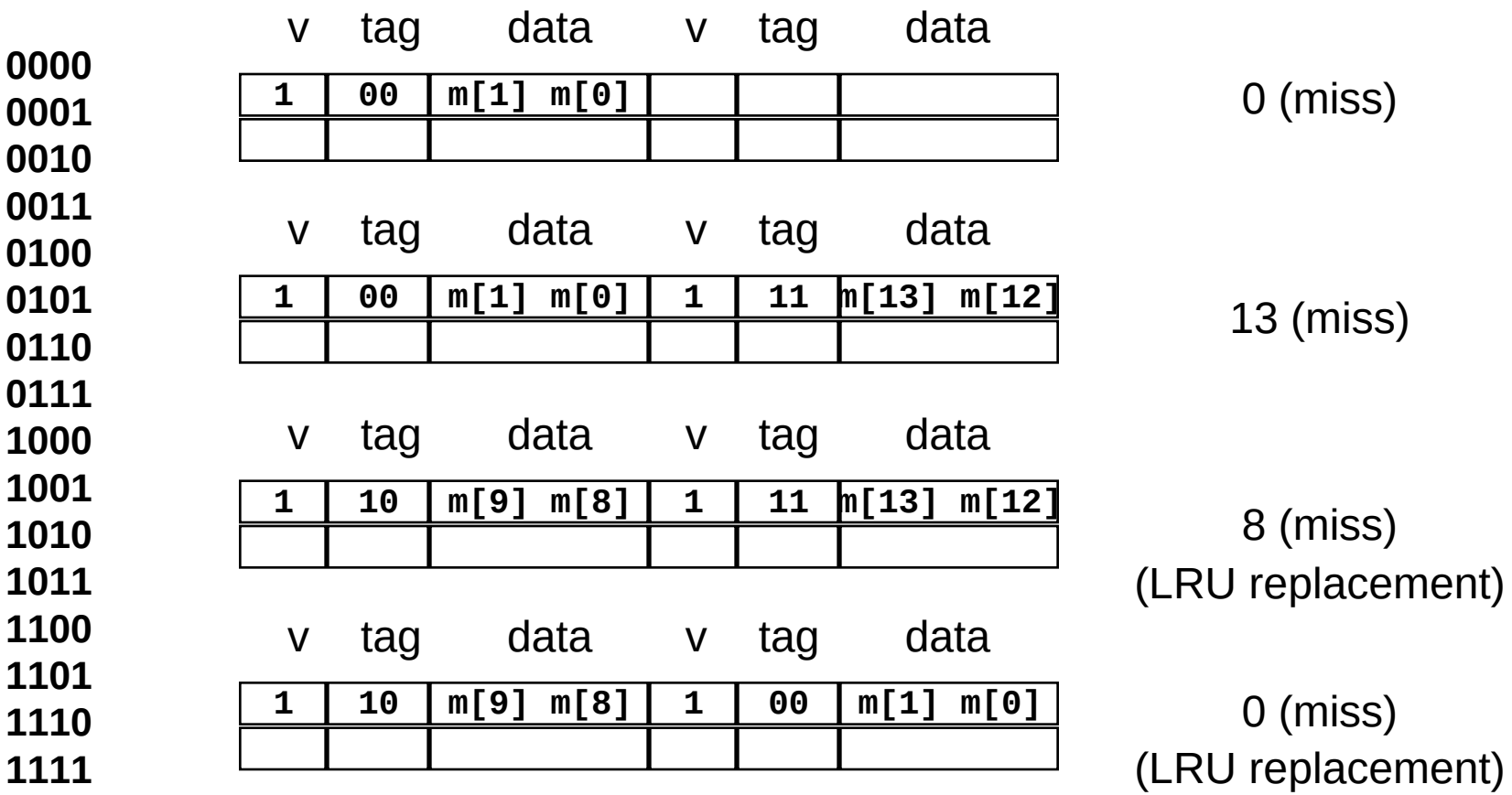
2-Way Set Associative Simulation

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

N=16 addresses B=2 bytes/line S=2 sets E=2 entries/set

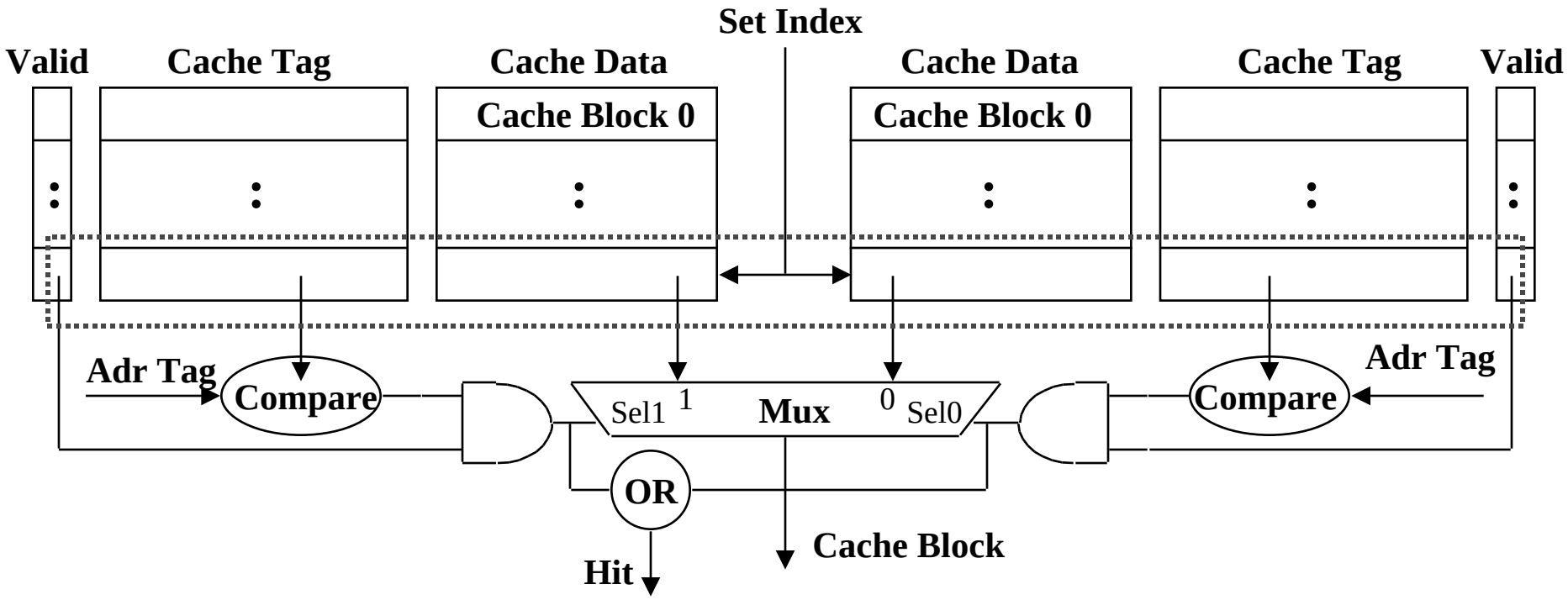
Address trace (reads):

0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]



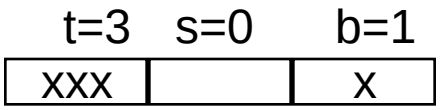
Two-Way Set Associative Cache Implementation

- Set index selects a set from the cache
- The two tags in the set are compared in parallel
- Data is selected based on the tag result

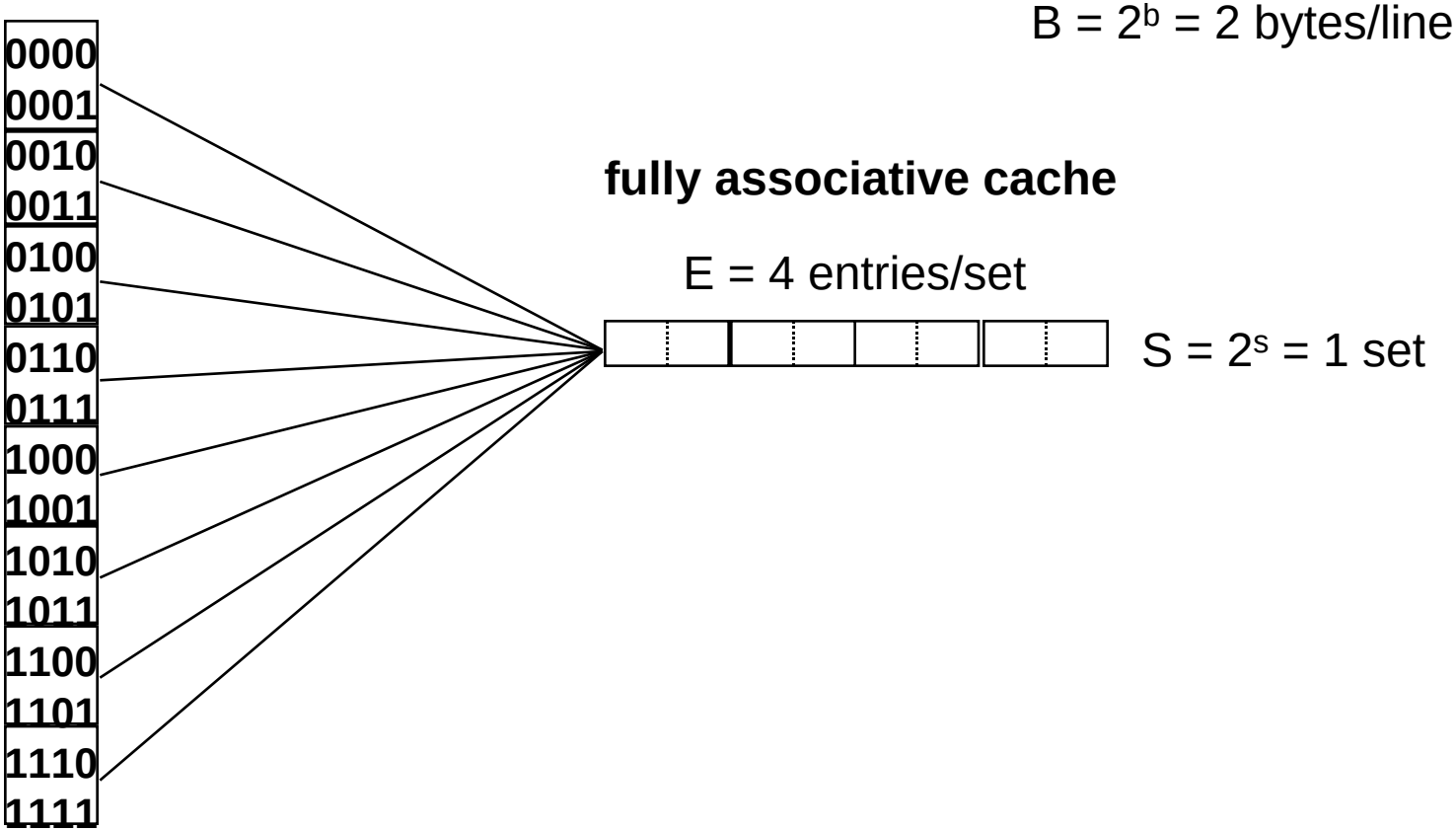


Fully associative cache (E = C/B)

N = 16 addresses (n=4)



cache size:
C = 8 data bytes
line size:
B = 2^b = 2 bytes/line



Fully associative cache simulation

N=16 addresses B=2 bytes/line S=1 sets E=4 entries/set
Address trace (reads):
0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]

t=3	s=0	b=1
xxx		x

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

0 (miss)

(1)

v	tag	data
1	00	m[1] m[0]

set ø

13 (miss)

(2)

v	tag	data
1	000	m[1] m[0]
1	110	m[13] m[12]

8 (miss)

(3)

v	tag	data
1	000	m[1] m[0]
1	110	m[13] m[12]
1	100	m[9] m[8]

Replacement Algorithms

- ***When a block is fetched, which block in the target set should be replaced?***

Usage based algorithms:

- **Least recently used (LRU)**
  - replace the block that has been referenced least recently
  - hard to implement

Non-usage based algorithms:

- **First-in First-out (FIFO)**
  - treat the set as a circular queue, replace block at head of queue.
  - easy to implement
- **Random (RAND)**
  - replace a random block in the set
  - even easier to implement

Implementing RAND and FIFO

FIFO:

- **maintain a modulo E counter for each set.**
- **counter in each set points to next block for replacement.**
- **increment counter with each replacement.**

RAND:

- **maintain a single modulo E counter.**
- **counter points to next block for replacement in any set.**
- **increment counter according to some schedule:**
  - each clock cycle,
  - each memory reference, or
  - each replacement anywhere in the cache.

LRU

- **Need state machine for each set**
- **Encodes usage ordering of each element in set**
- **E! possibilities ==> ~ E log E bits of state**

Write Strategies

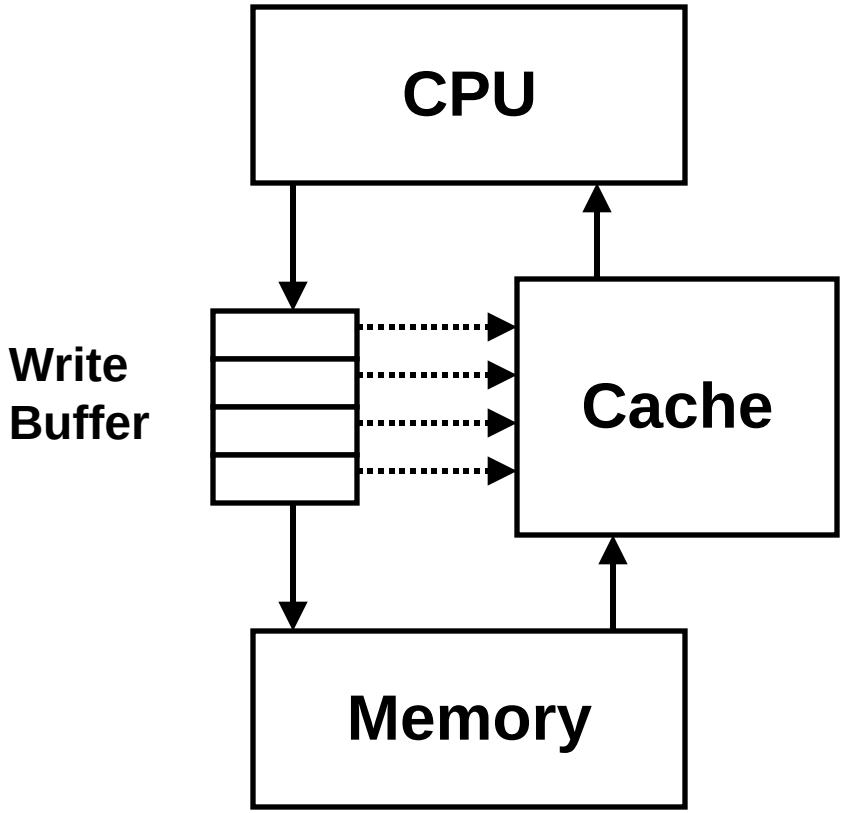
Write Policy

- **What happens when processor writes to the cache?**
- ***write through***
  - information is written to the block in cache *and* memory.
  - memory always consistent with cache
  - Can overwrite cache entry
- ***write back***
  - information is written only block in cache. Modified block written to memory only when it is replaced.
  - requires a dirty bit for each block
    - » To remove dirty block from cache, must write back to main memory
  - memory not always consistent with cache

Write Buffering

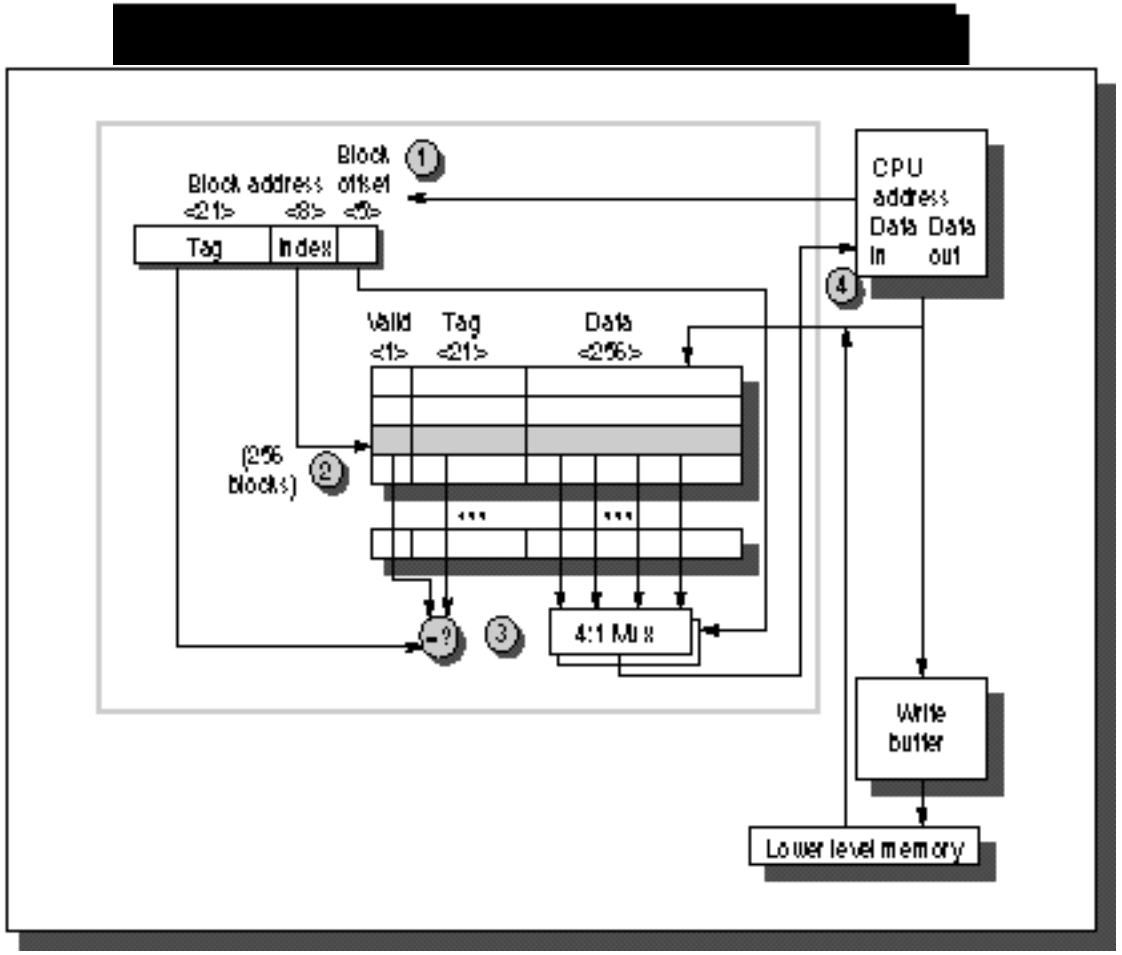
Write Buffer

- Common optimization for write-through caches
- Overlaps memory updates with processor execution
- Read operation must check write buffer for matching address



Alpha AXP 21064 direct mapped data cache

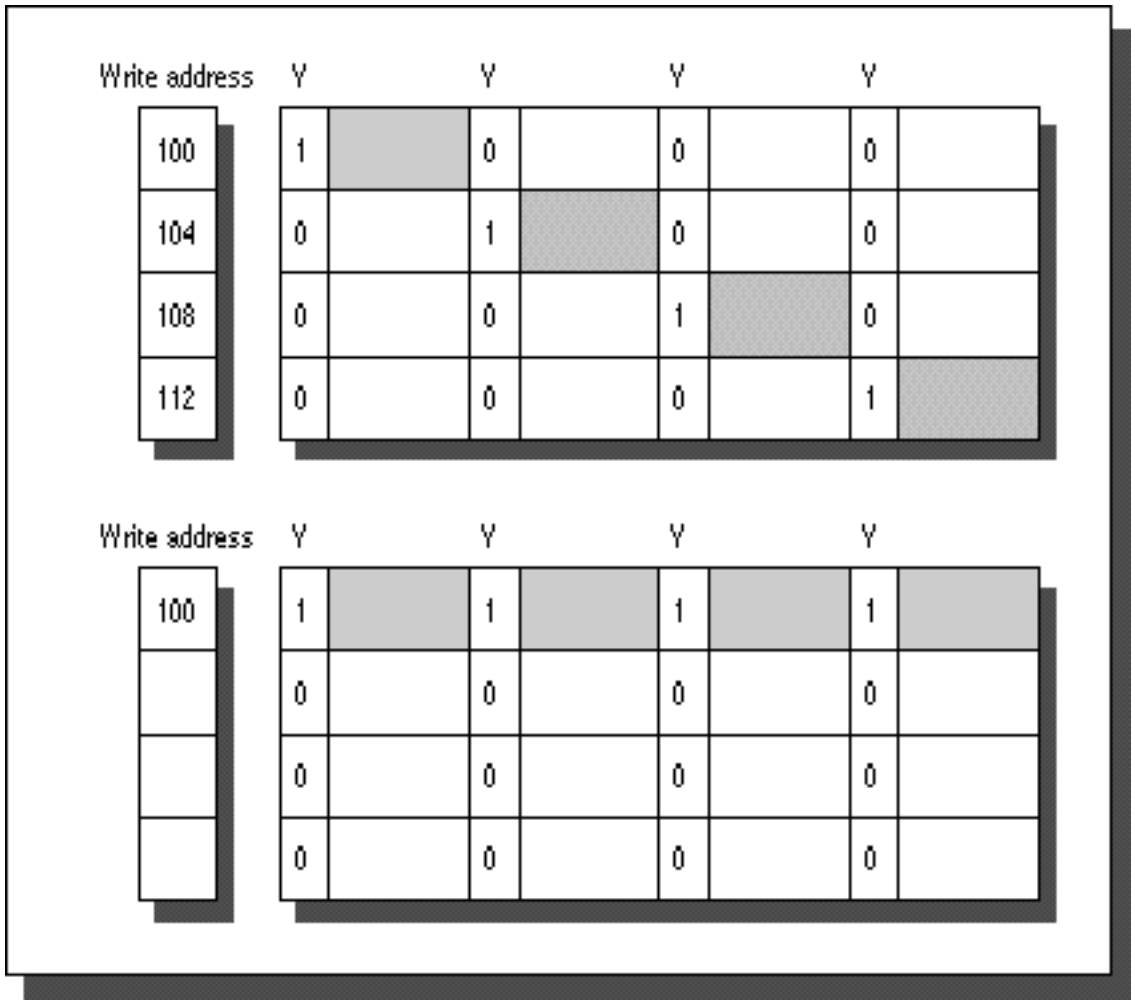
34-bit address
256 blocks
32-bytes/block
(= 8 quadwords)



Write merging

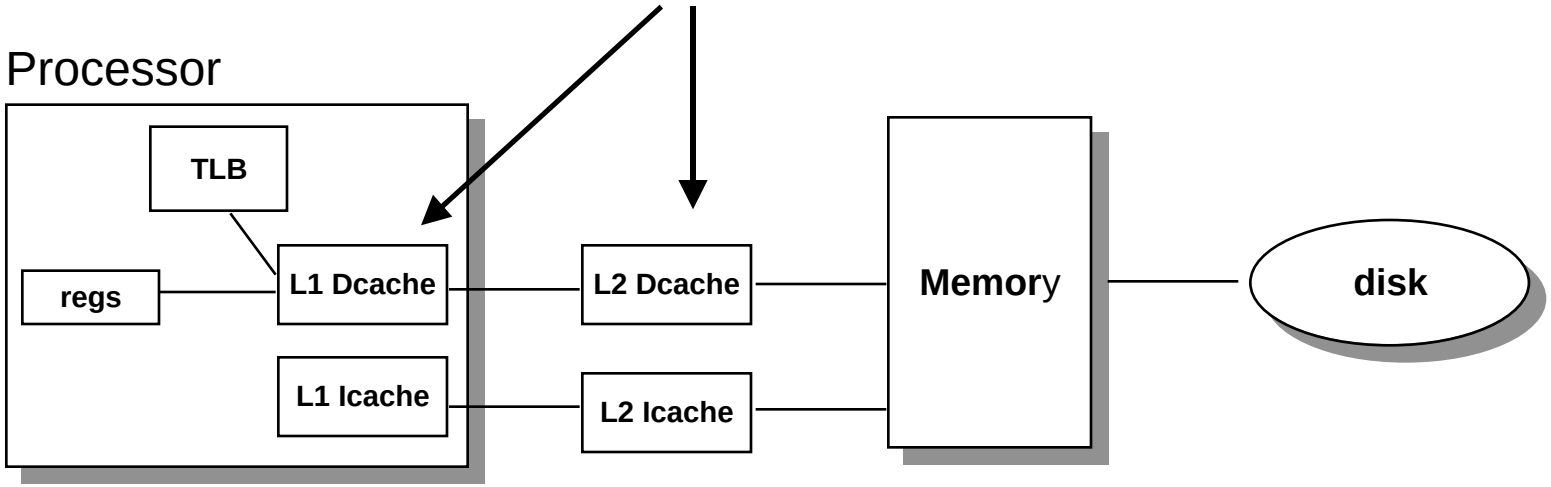
A write buffer that does not do write merging

A write buffer that does write merging



Multi-level caches

Can have separate Icache and Dcache or *unified* Icache/Dcache

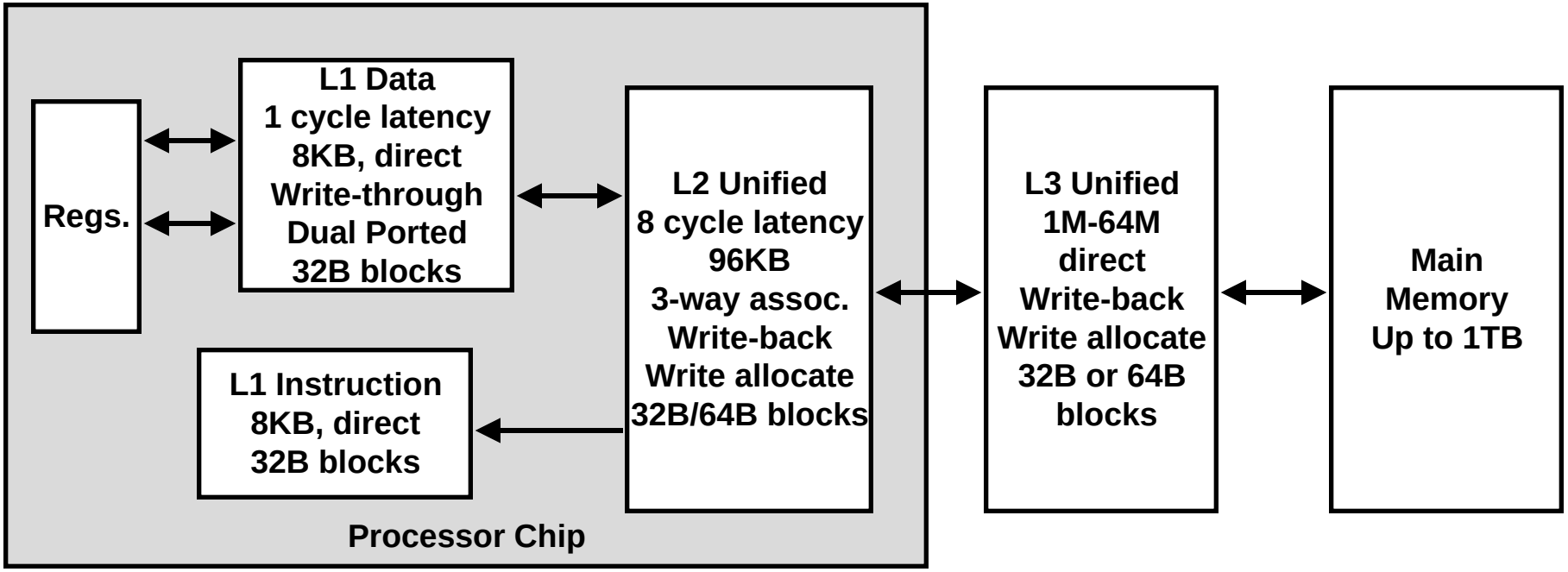


size:	200 B	8 KB	1M SRAM	128 MB DRAM	10 GB
speed:	5 ns	5 ns	6 ns	100 ns	10 ms
$/Mbyte:			$256/MB	$2/MB	$0.10/MB
block size:	4 B	16 B	32 KB	4 KB	

larger, slower, cheaper

larger block size, higher associativity, more likely to write back

Alpha 21164 Hierarchy

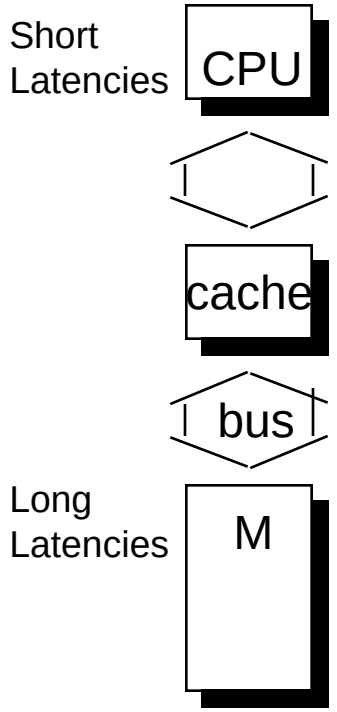


- **Improving memory performance was a main design goal**
- **Earlier Alpha's CPUs starved for data**

Bandwidth Matching

Challenge

- **CPU works with short cycle times**
- **DRAM (relatively) long cycle times**
- ***How can we provide enough bandwidth between processor & memory?***



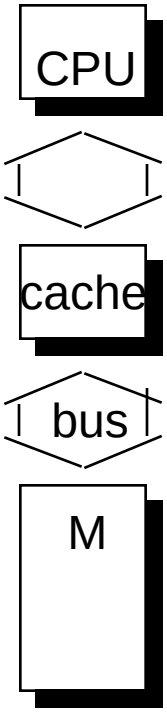
Effect of Caching

- **Caching greatly reduces amount of traffic to main memory**
- **But, sometimes need to move large amounts of data from memory into cache**

Trends

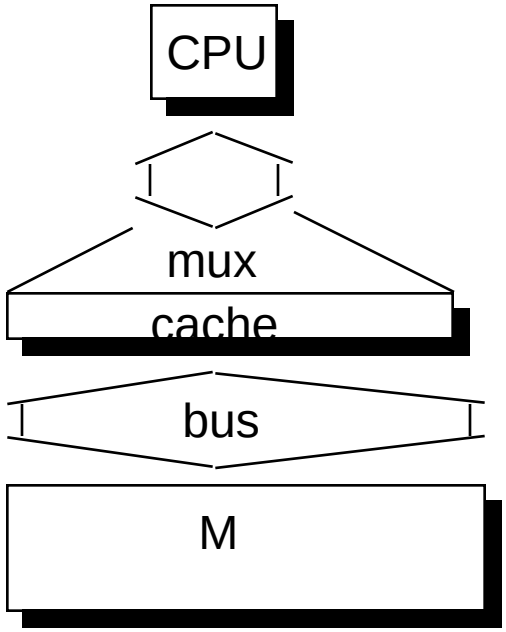
- **Need for high bandwidth much greater for multimedia applications**
  - Repeated operations on image data
- **Recent generation machines (e.g., Pentium II) greatly improve on predecessors**

High Bandwidth Memory Systems



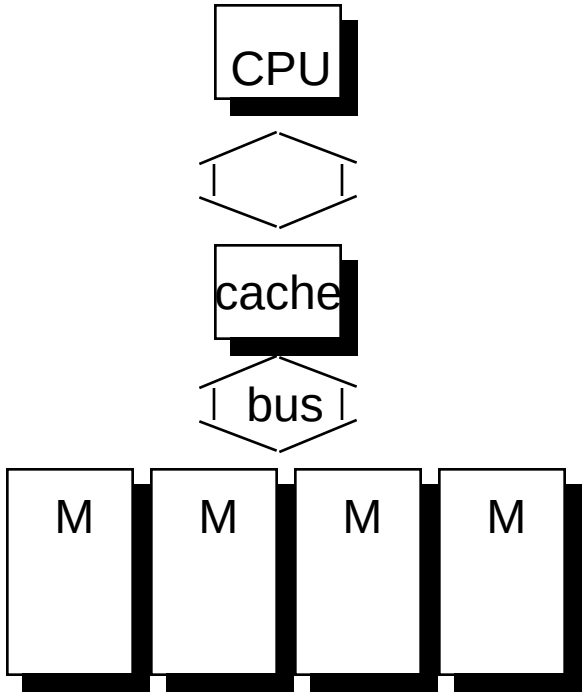
Solution 1
High BW DRAM

Example:
Page Mode DRAM
RAMbus



Solution 2
Wide path between memory & cache

Example: Alpha AXP 21064
256 bit wide bus, L2 cache,
and memory.



Solution 3
Memory bank interleaving

Example: Alpha 8400
4 GByte 4 bank memory module

Cache Performance Metrics

Miss Rate

- **fraction of memory references not found in cache (misses/references)**
- **Typical numbers:**

  5-10% for L1

  1-2% for L2

Hit Time

- **time to deliver a block in the cache to the processor (includes time to determine whether the block 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 10-30 cycles for main memory

Impact of Cache and Block Size

Cache Size

- **Effect on miss rate**
  - Larger is better
- **Effect on hit time**
  - Smaller is faster

Block Size

- **Effect on miss rate**
  - Big blocks help exploit spatial locality
  - For given cache size, can hold fewer big blocks than little ones, though
- **Effect on miss penalty**
  - Longer transfer time

Impact of Associativity

- **Direct-mapped, set associative, or fully associative?**

Total Cache Size (tags+data)

- **Higher associativity requires more tag bits, LRU state machine bits**
- **Additional read/write logic, multiplexors**

Miss rate

- **Higher associativity decreases miss rate**

Hit time

- **Higher associativity increases hit time**
  - Direct mapped allows test and data transfer at the same time for read hits.

Miss Penalty

- **Higher associativity requires additional delays to select victim**

Impact of Replacement Strategy

- RAND, FIFO, or LRU?

Total Cache Size (tags+data)

- LRU requires complex state machine for each set
- FIFO requires simpler state machine for each set
- RAND very simple

Miss Rate

- LRU has up to ~10% lower miss rate than FIFO
- RAND does much worse

Miss Penalty

- LRU takes more time to select victim

Impact of Write Strategy

- **Write-through or write-back?**

Advantages of Write Through

- **Read misses are cheaper. Why?**
- **Simpler to implement.**
- **Requires a write buffer to pipeline writes**

Advantages of Write Back

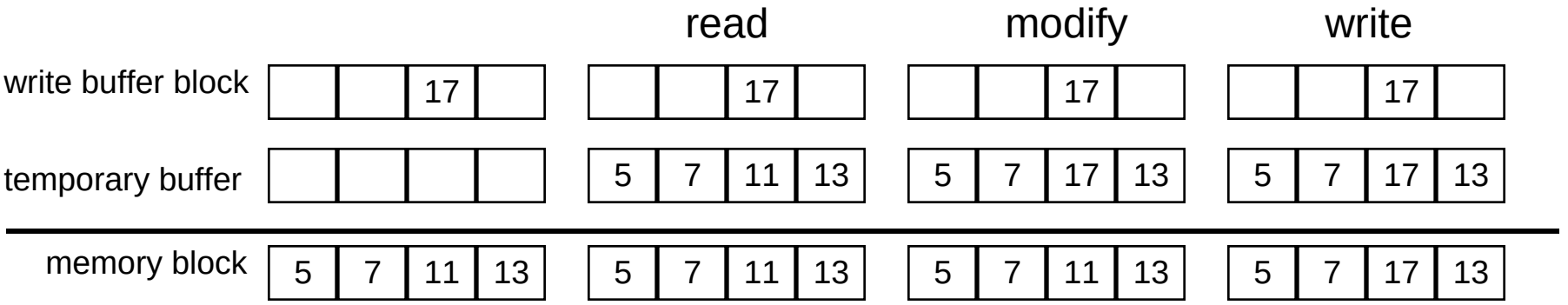
- **Reduced traffic to memory**
  - Especially if bus used to connect multiple processors or I/O devices
- **Individual writes performed at the processor rate**

Allocation Strategies

- **On a write miss, is the block loaded from memory into the cache?**

Write Allocate:

- **Block is loaded into cache on a write miss.**
- **Usually used with write back**
- **Otherwise, write-back requires read-modify-write to replace word within block**



- **But if you've gone to the trouble of reading the entire block, why not load it in cache?**

Allocation Strategies (Cont.)

- **On a write miss, is the block loaded from memory into the cache?**

No-Write Allocate (Write Around):

- **Block is not loaded into cache on a write miss**
- **Usually used with write through**
  - Memory system directly handles word-level writes

Qualitative Cache Performance Model

Miss Types

- **Compulsory ("Cold Start") Misses**
  - First access to line not in cache
- **Capacity Misses**
  - Active portion of memory exceeds cache size
- **Conflict Misses**
  - Active portion of address space fits in cache, but too many lines map to same cache entry
  - Direct mapped and set associative placement only
- **Validation Misses**
  - Block invalidated by multiprocessor cache coherence mechanism

Hit Types

- **Reuse hit**
  - Accessing same word that previously accessed
- **Line hit**
  - Accessing word spatially near previously accessed word

Interactions Between Program & Cache

Major Cache Effects to Consider

- **Total cache size**
  - Try to keep heavily used data in highest level cache
- **Block size (sometimes referred to "line size")**
  - Exploit spatial locality

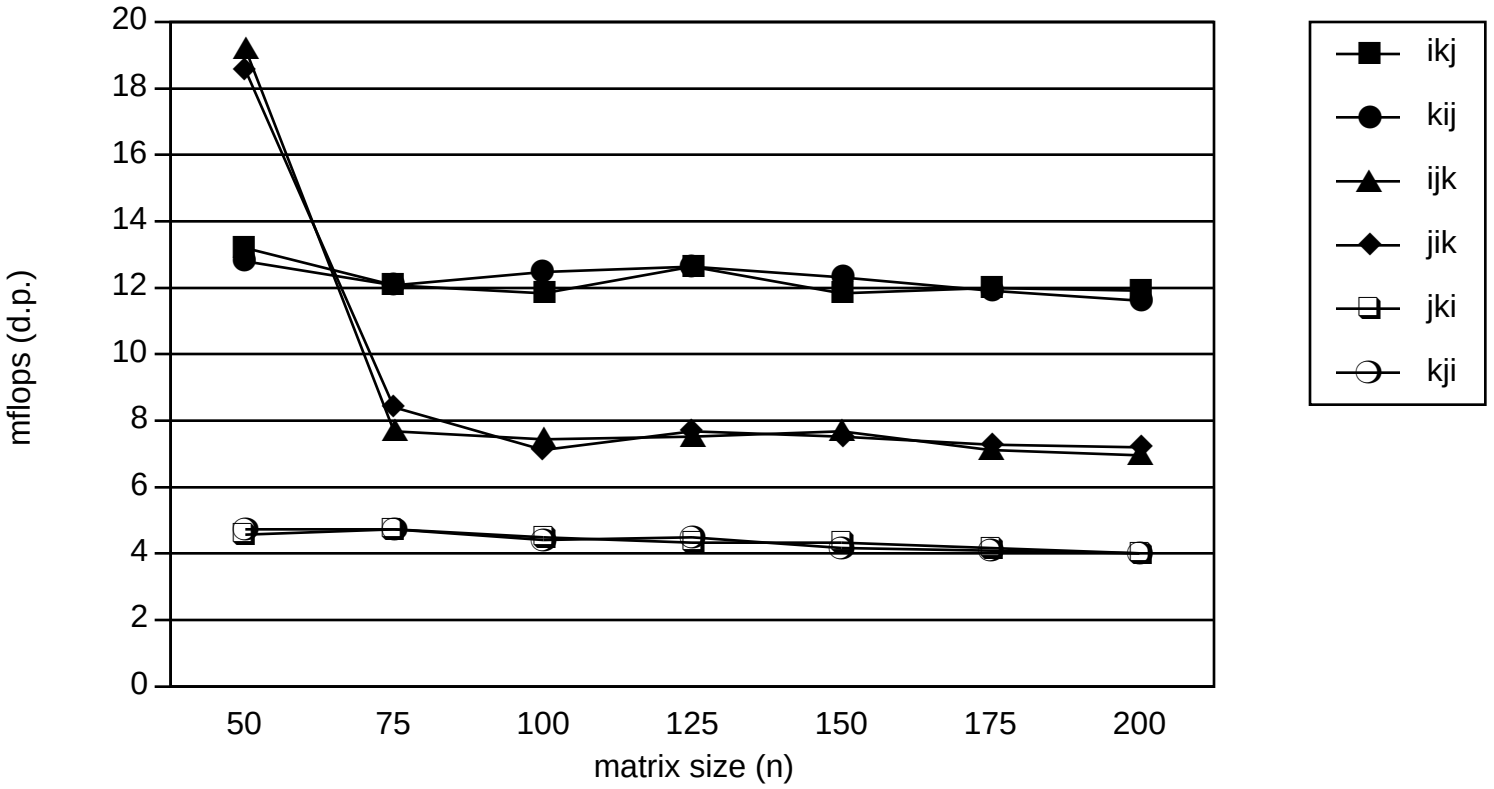
Variable sum held in register

```
/* 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;
  }
}
```

Example Application

- **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

Matmult Performance (Sparc20)



- **As matrices grow in size, exceed cache capacity**
- **Different loop orderings give different performance**
  - Cache effects
  - Whether or not can accumulate in register

Layout of Arrays in Memory

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];
```

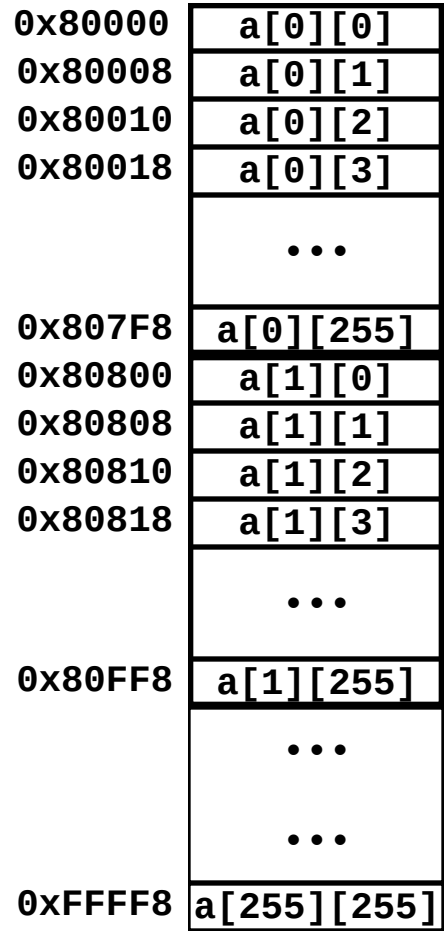
- **Accesses successive elements**
- **For block size > 8, get spatial locality**
  - Cold Start Miss Rate = 8/B

Stepping Through Rows in One Column

```
for (i = 0; i < n; i++)
   sum += a[i][0];
```

- **Accesses distant elements**
- **No spatial locality**
  - Cold Start Miss rate = 1

Memory Layout



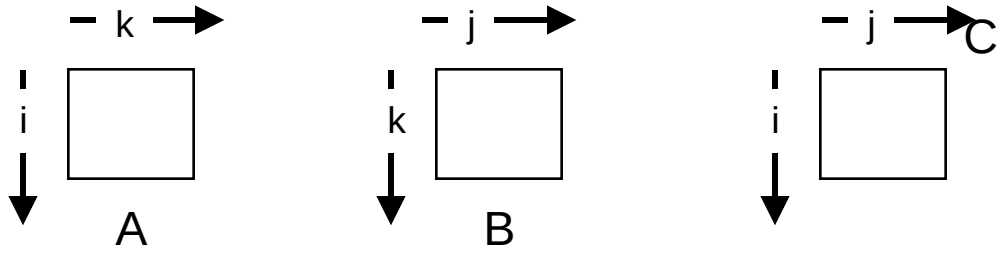
Miss Rate Analysis

Assume

- **Block size = 32B (big enough for 4 double's)**
- **n is very large**
  - Approximate 1/n as 0.0
- **Cache not even big enough to hold multiple rows**

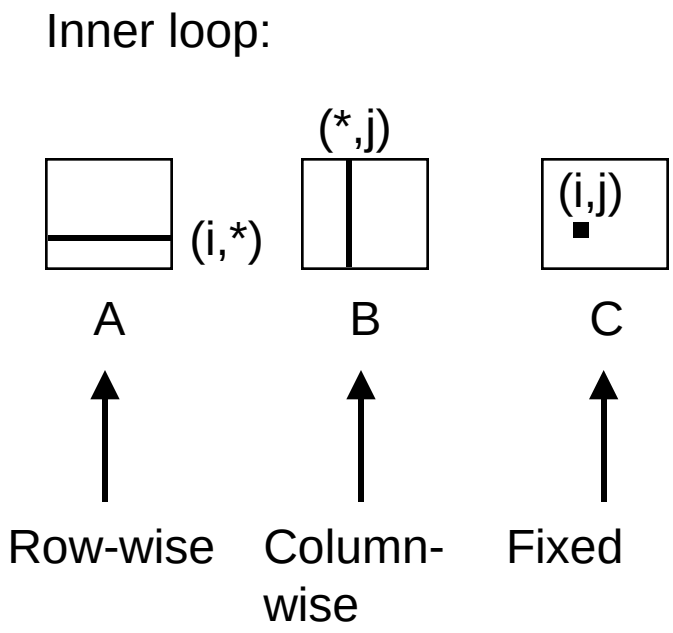
Analysis Method

- **Look at access pattern by 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;
  }
}
```

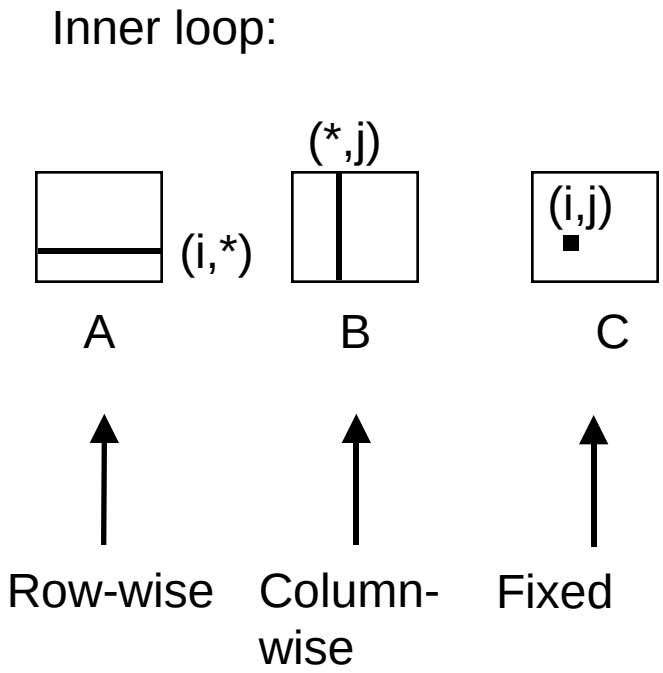


Approx. Miss Rates

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
  }
}
```

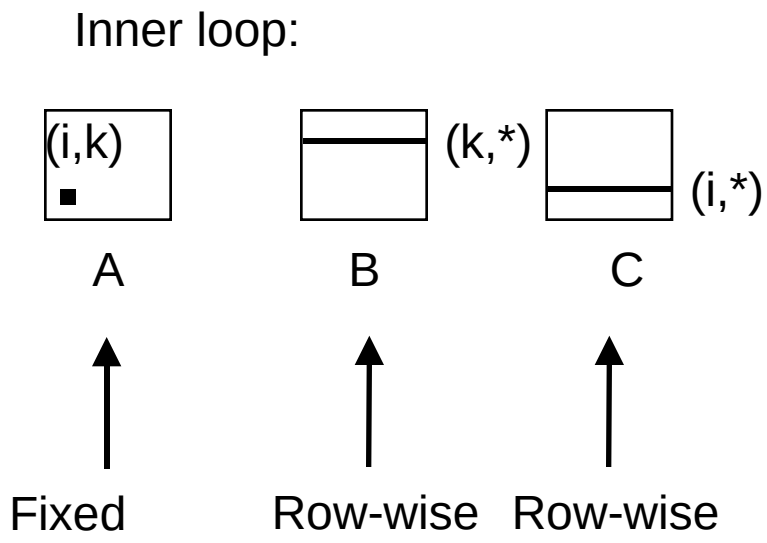


Approx. Miss Rates

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];
  }
}
```

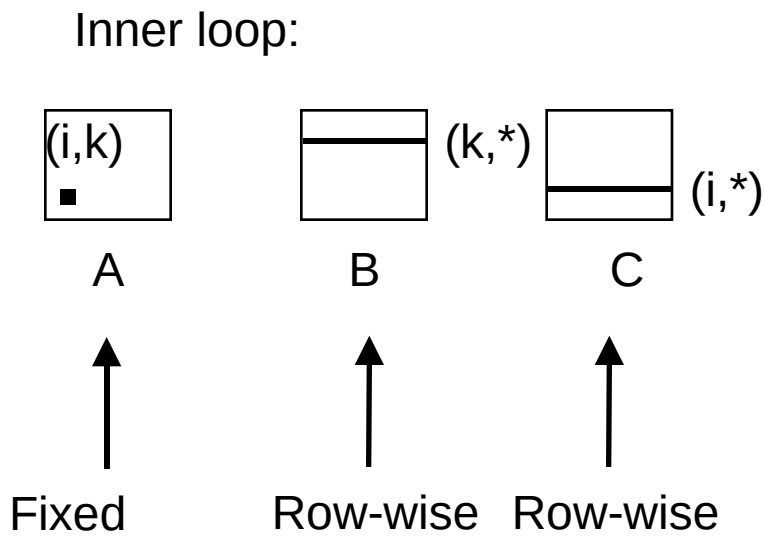


Approx. Miss Rates

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];
  }
}
```

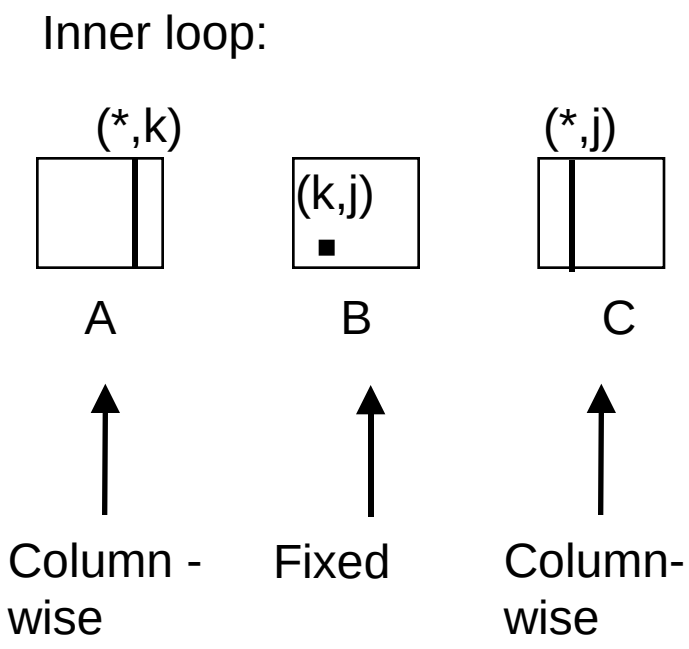


Approx. Miss Rates

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;
  }
}
```

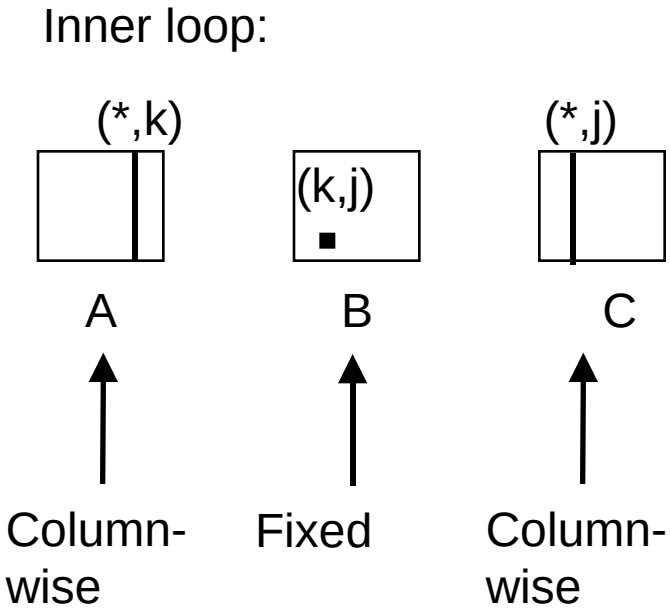


Approx. Miss Rates

a	b	c
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;
  }
}
```



Approx. Miss Rates

a	b	c
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (L=2, S=0, MR=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;
   }
}
```

jik (L=2, S=0, MR=1.25)

```
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
    }
}
```

kij (L=2, S=1, MR=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];
   }
}
```

ikj (L=2, S=1, MR=0.5)

```
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];
   }
}
```

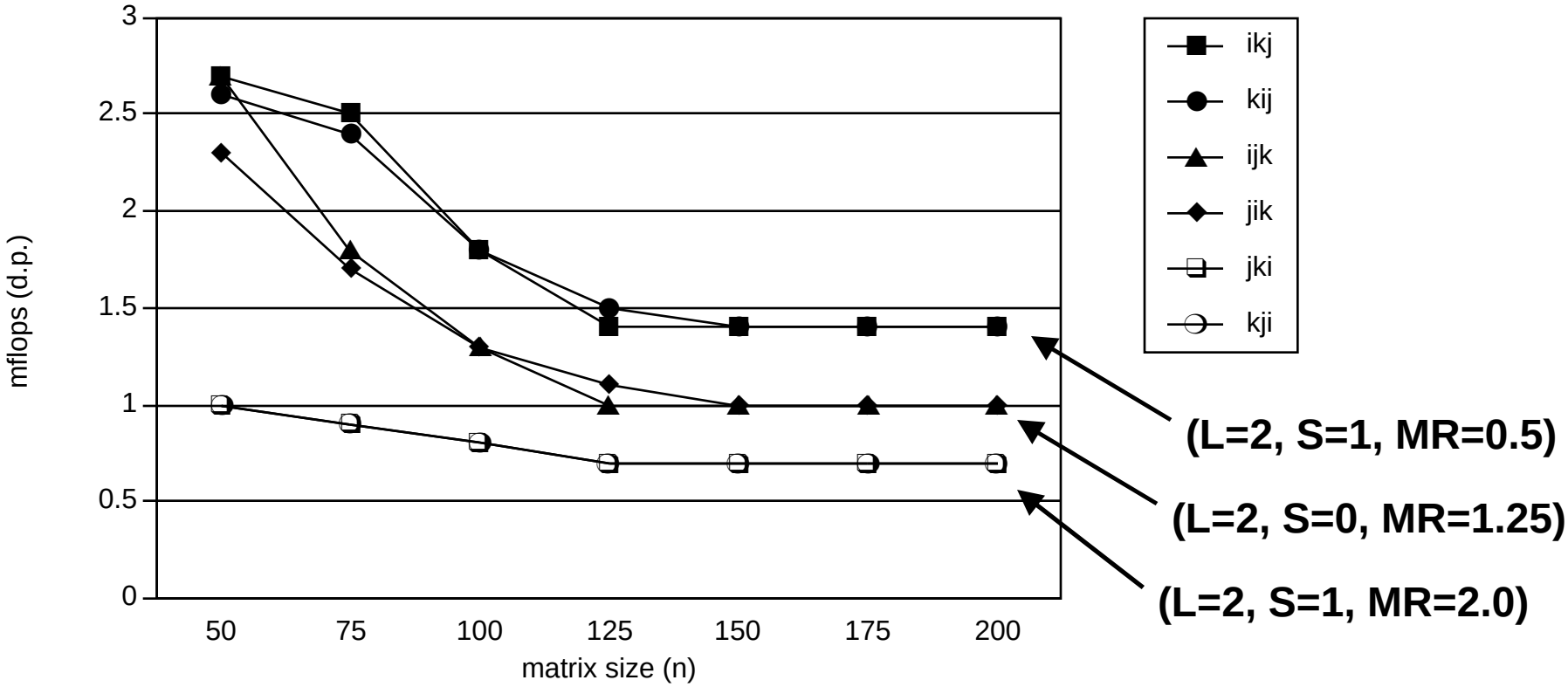
jki (L=2, S=1, MR=2.0)

```
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;
   }
}
```

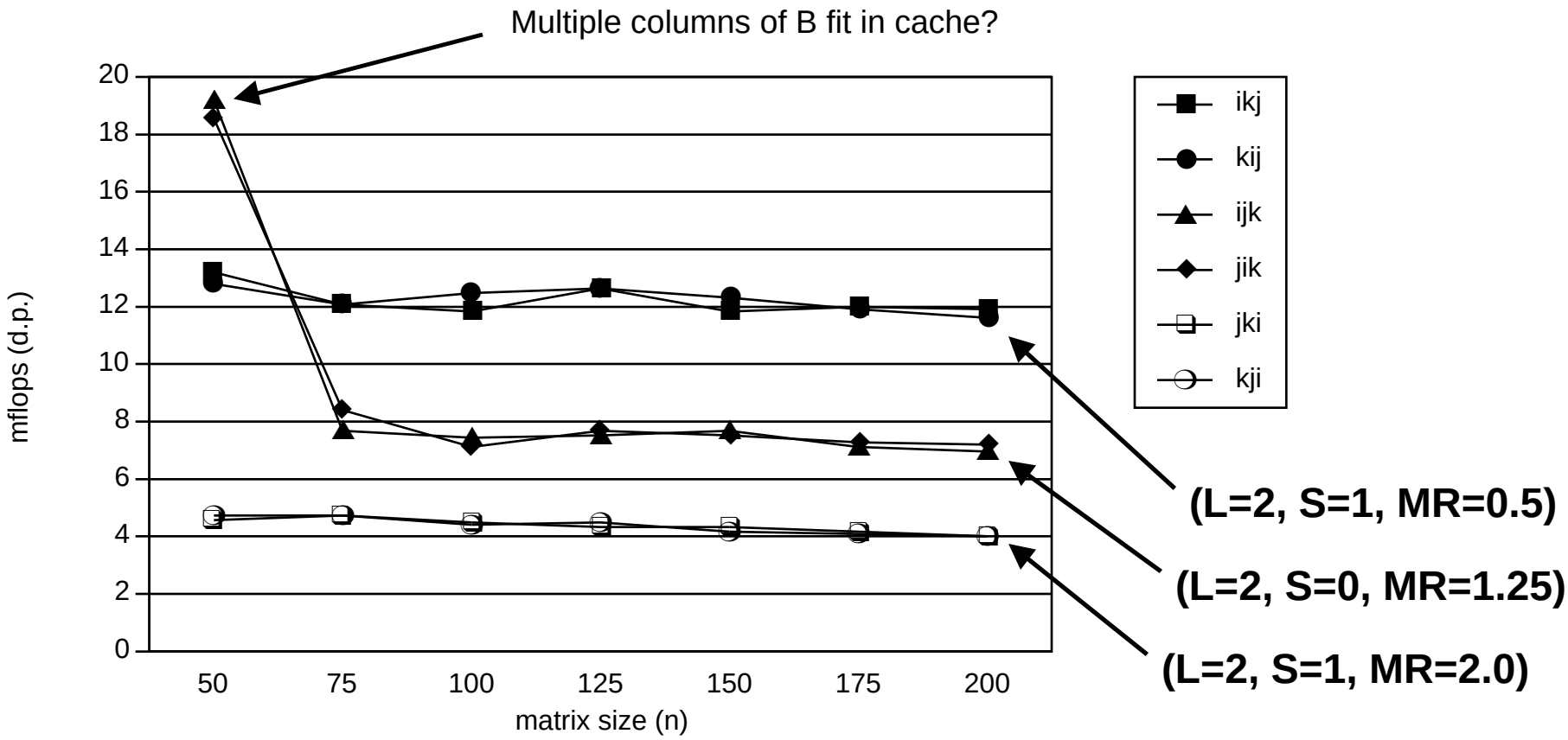
kji (L=2, S=1, MR=2.0)

```
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;
   }
}
```

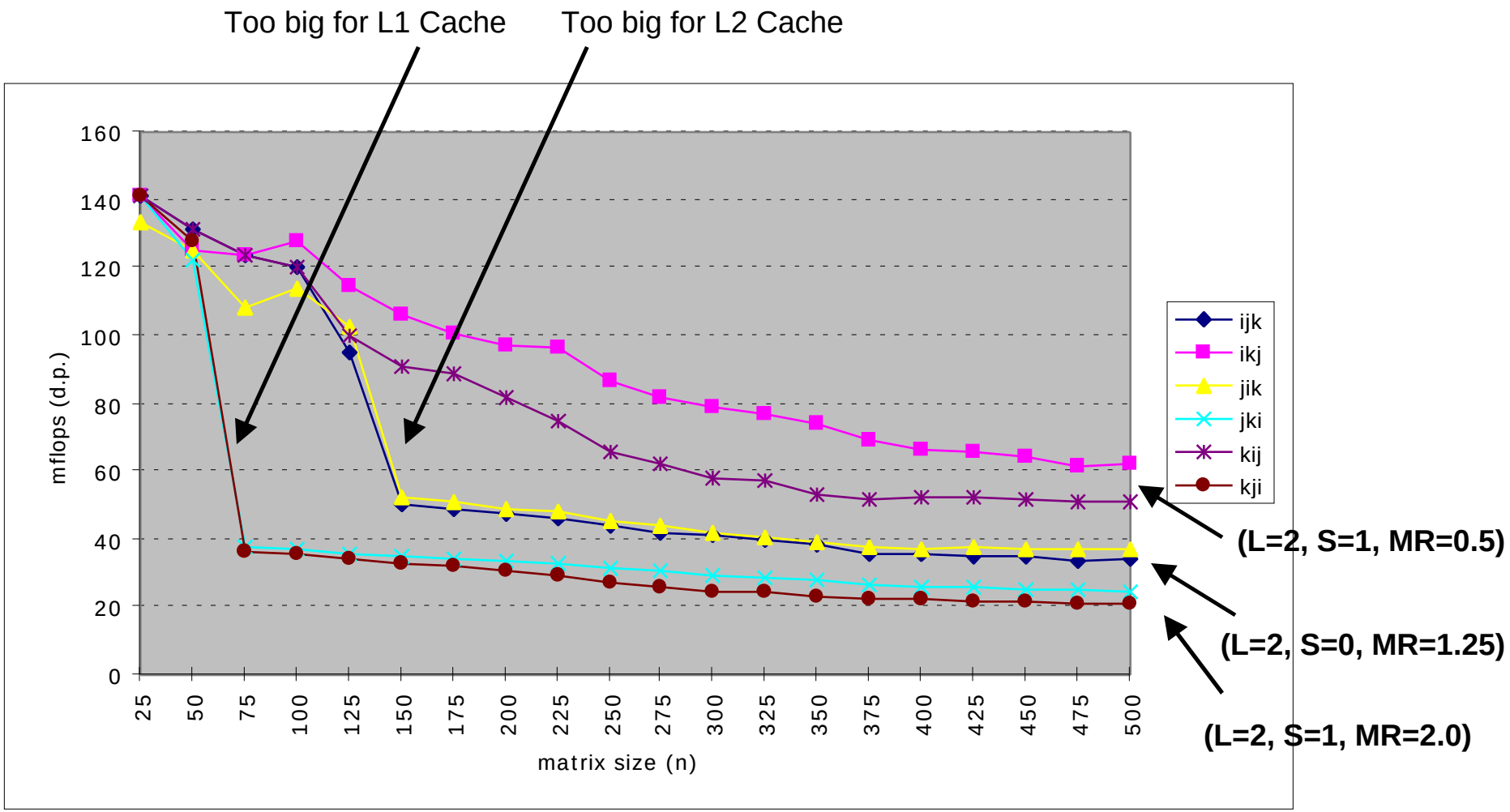
Matmult performance (DEC5000)



Matmult Performance (Sparc20)



Matmult Performance (Alpha 21164)



Block Matrix Multiplication

Example n=8, B = 4:

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \times \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$$

Key idea: Sub-blocks (i.e., A_{ij}) 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;
      }
    }
  }
}
```

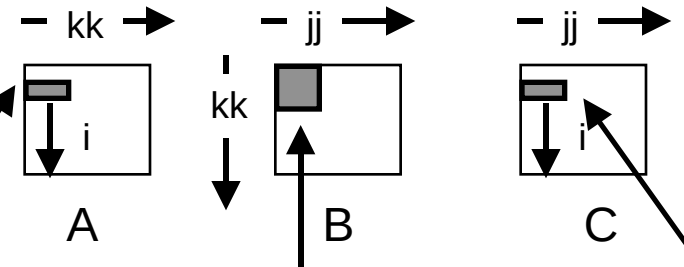
***Warning*: Code in H&P (p. 409) has bugs!**

Blocked Matrix Multiply Analysis

- **Innermost loop pair multiplies 1 X bsize sliver of A times 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

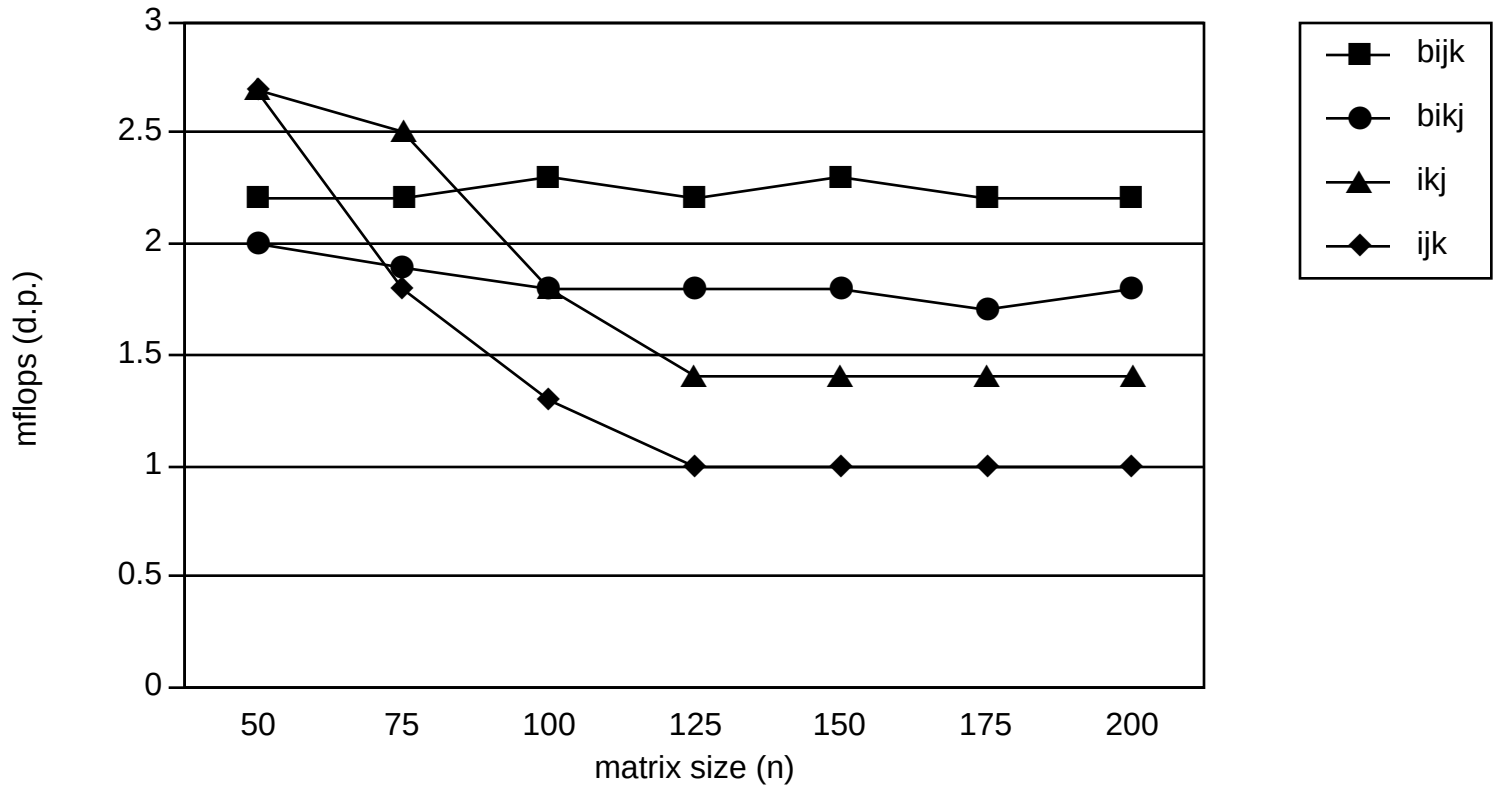


row sliver accessed bsize times

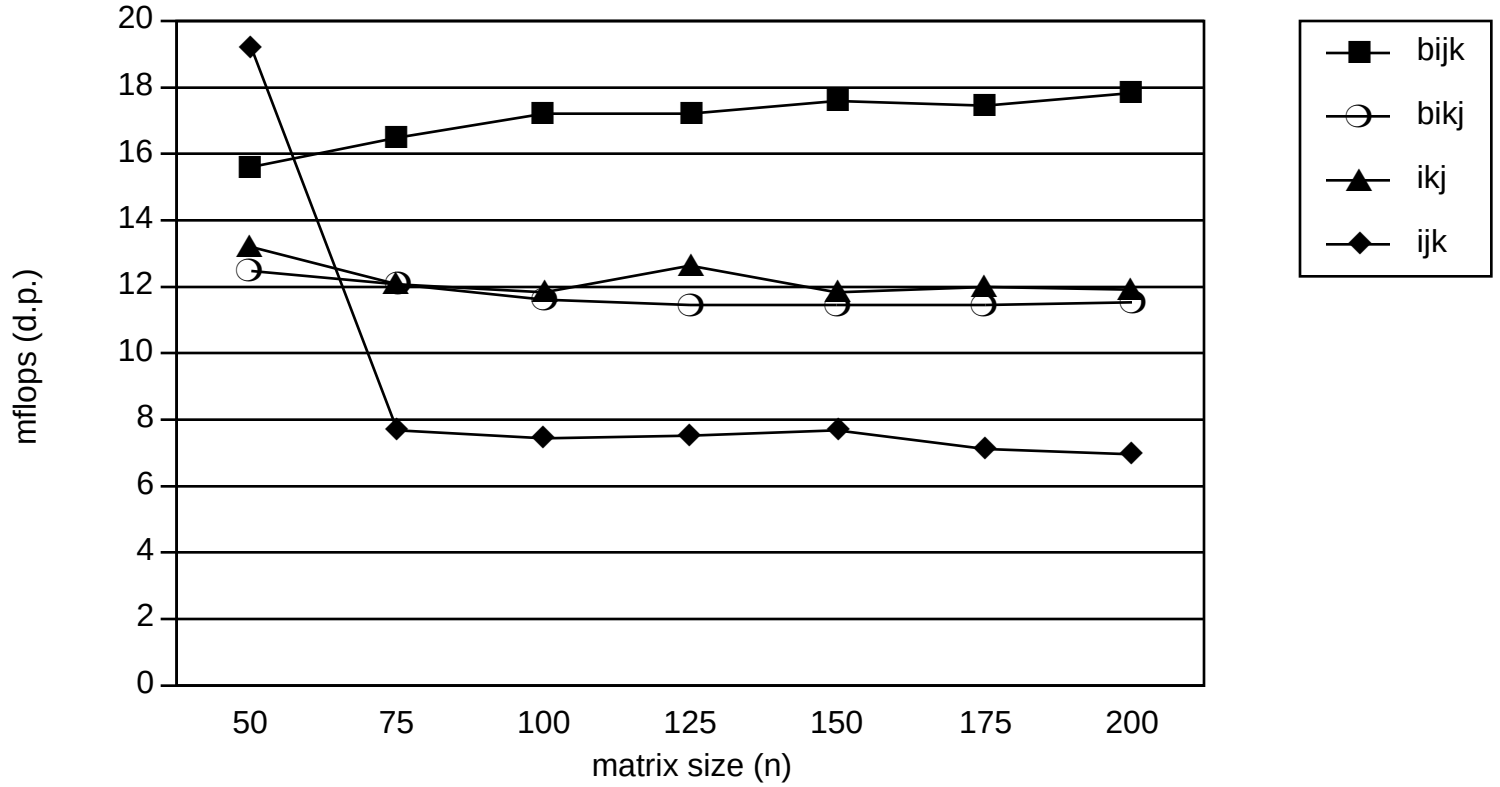
block reused n times in succession

Update successive elements of sliver

Blocked matmult perf (DEC5000)



Blocked matmult perf (Sparc20)



Blocked matmult perf (Alpha 21164)