

More about Caches

Randal E. Bryant
CS347 Lecture 8
Feb. 5, 1998

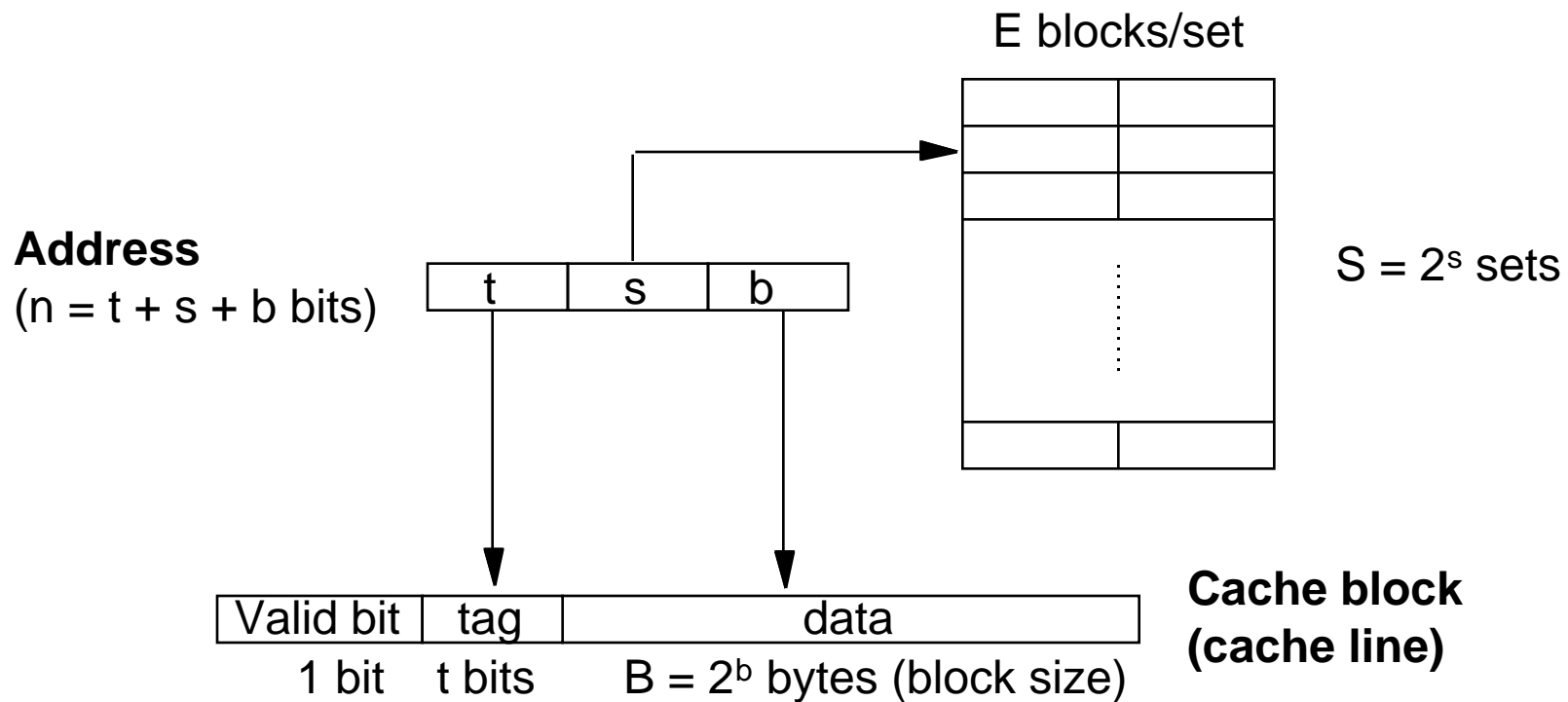
Topics

- **Performance factors**
- **Access patterns & program performance**
 - Use matrix multiply as case study
- **Memory hierarchy bandwidths**

Basic Cache Organization

Address space ($N = 2^n$ bytes)

Cache ($C = S \times E \times B$ bytes)



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

		read	modify	write																
write buffer block	<table border="1"><tr><td></td><td></td><td>17</td><td></td></tr></table>			17		<table border="1"><tr><td></td><td></td><td>17</td><td></td></tr></table>			17		<table border="1"><tr><td></td><td></td><td>17</td><td></td></tr></table>			17		<table border="1"><tr><td></td><td></td><td>17</td><td></td></tr></table>			17	
		17																		
		17																		
		17																		
		17																		
temporary buffer	<table border="1"><tr><td></td><td></td><td></td><td></td></tr></table>					<table border="1"><tr><td>5</td><td>7</td><td>11</td><td>13</td></tr></table>	5	7	11	13	<table border="1"><tr><td>5</td><td>7</td><td>17</td><td>13</td></tr></table>	5	7	17	13	<table border="1"><tr><td>5</td><td>7</td><td>17</td><td>13</td></tr></table>	5	7	17	13
5	7	11	13																	
5	7	17	13																	
5	7	17	13																	
memory block	<table border="1"><tr><td>5</td><td>7</td><td>11</td><td>13</td></tr></table>	5	7	11	13	<table border="1"><tr><td>5</td><td>7</td><td>11</td><td>13</td></tr></table>	5	7	11	13	<table border="1"><tr><td>5</td><td>7</td><td>11</td><td>13</td></tr></table>	5	7	11	13	<table border="1"><tr><td>5</td><td>7</td><td>17</td><td>13</td></tr></table>	5	7	17	13
5	7	11	13																	
5	7	11	13																	
5	7	11	13																	
5	7	17	13																	

- 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

Compulsory Misses

- First access to line not in cache
- Also called “Cold start” misses

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
- Not our concern right now

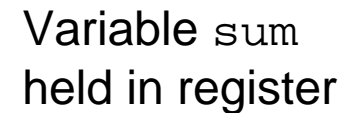
Interactions Between Program & Cache

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

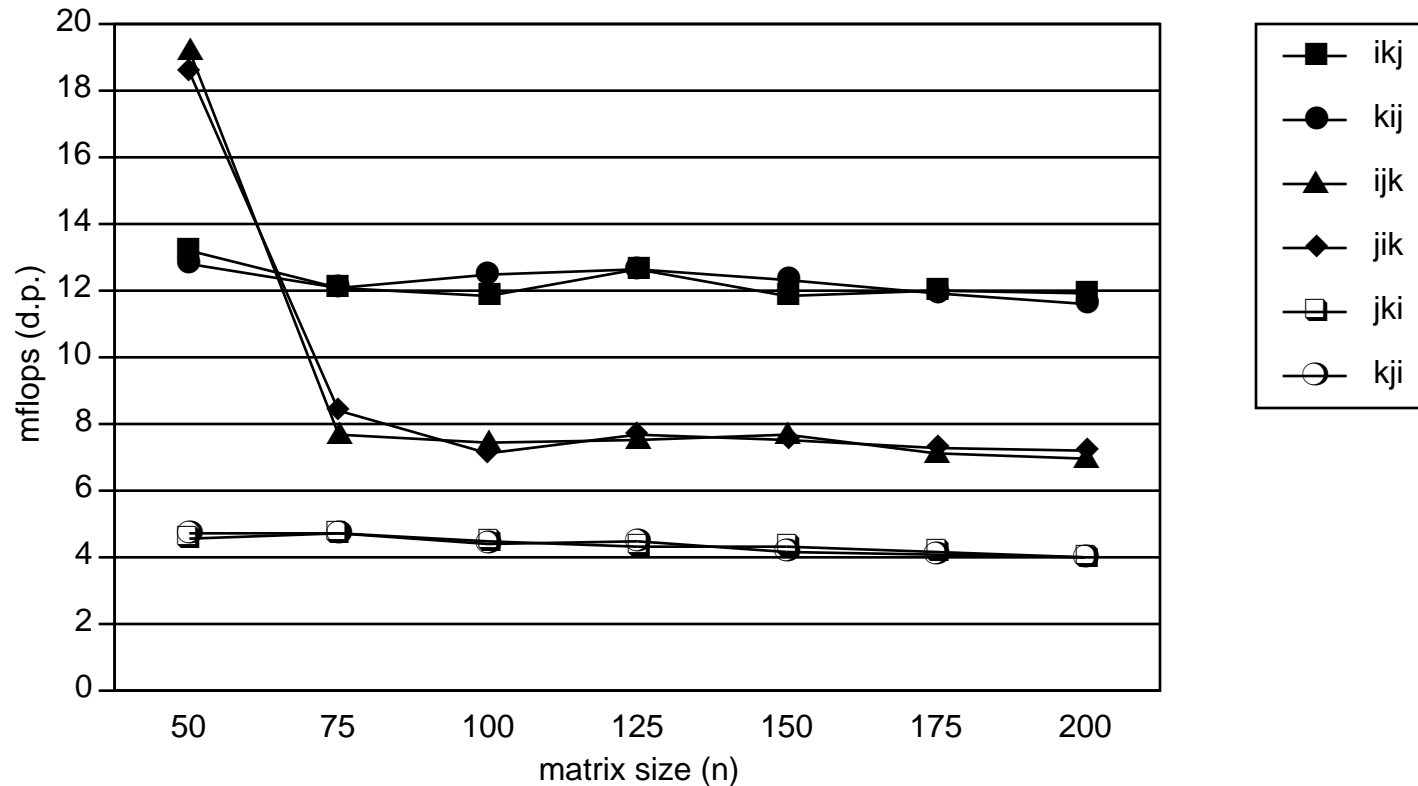
```
/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```



Example Application

- **Multiply $n \times n$ 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 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 Column

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

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

Memory Layout

0x80000	a[0][0]
0x80008	a[0][1]
0x80010	a[0][2]
0x80018	a[0][3]
	...
0x807F8	a[0][255]
0x80800	a[1][0]
0x80808	a[1][1]
0x80810	a[1][2]
0x80818	a[1][3]
	...
0x80FF8	a[1][255]
	...
	...
0xFFFF8	a[255][255]

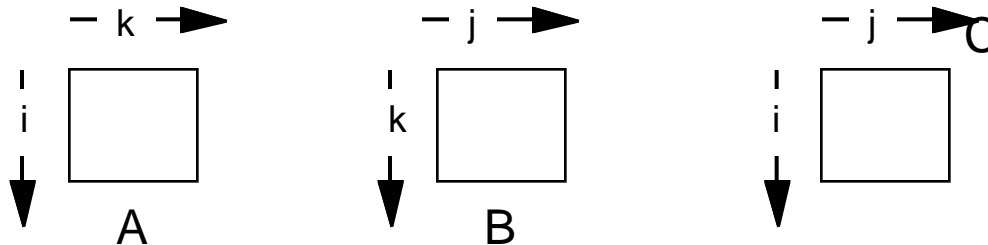
Miss Rate Analysis

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

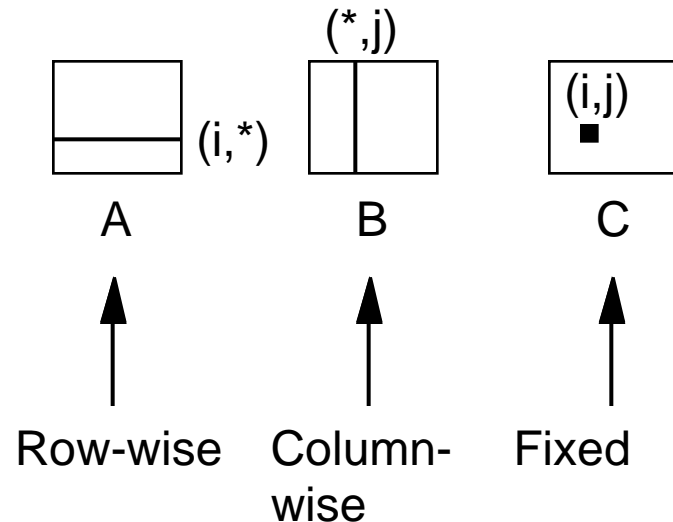
- Look at access pattern by inner loop



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;
  }
}
```

Inner loop:



Approx. Miss Rates

a	b	c
0.25	1.0	0.0

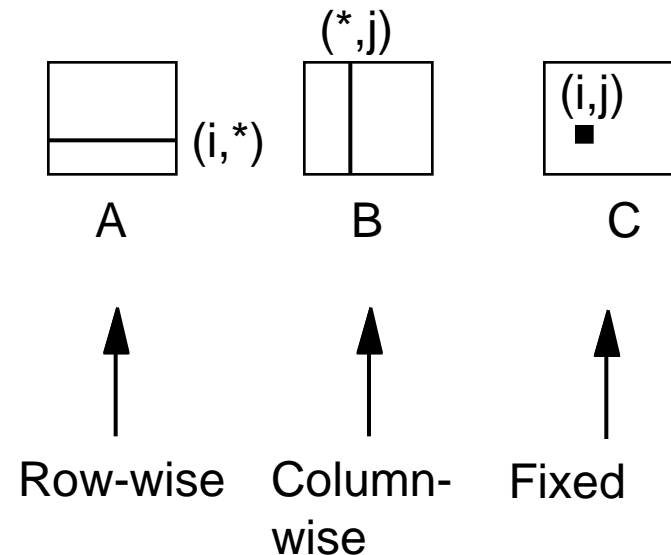
Matrix multiplication (jik)

```
/* 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

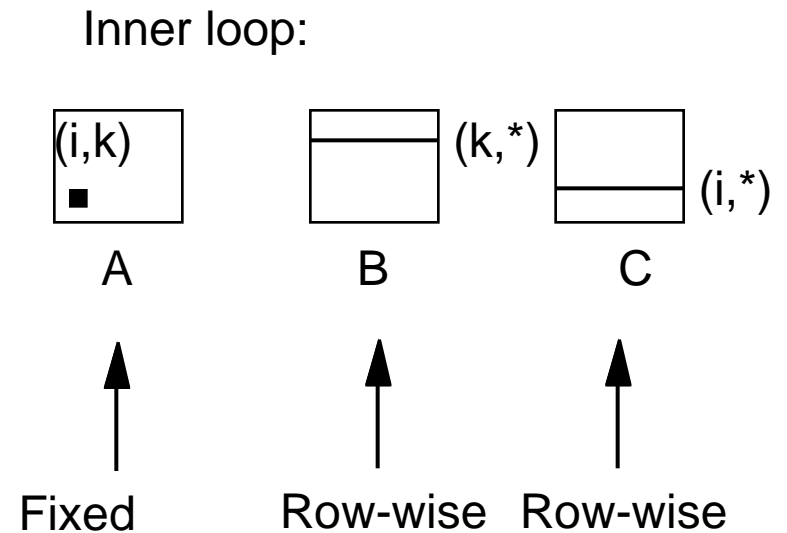
a	b	c
0.25	1.0	0.0

Inner loop:



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];
  }
}
```

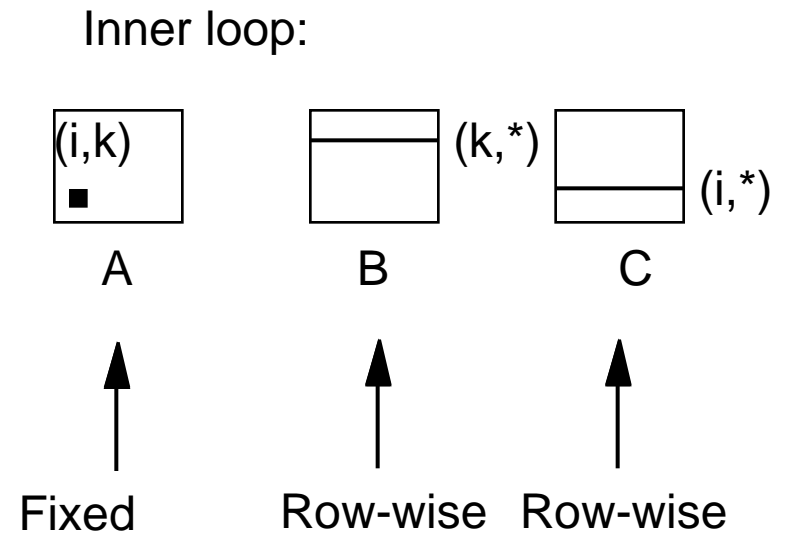


Approx. Miss Rates

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];
  }
}
```



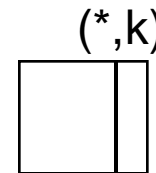
Approx. Miss Rates

a	b	c
0.0	0.25	0.25

Matrix multiplication (jki)

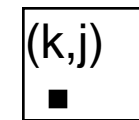
```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:



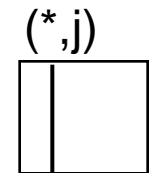
A

↑
Column -
wise



B

↑
Fixed



C

↑
Column-
wise

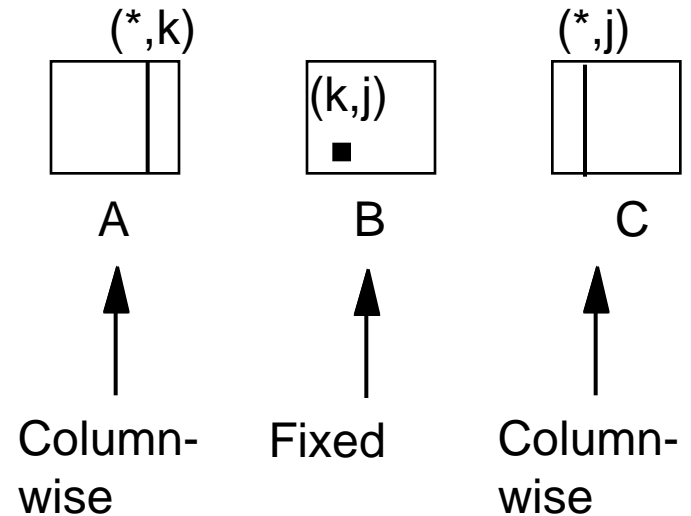
Approx. Miss Rates

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;  
  }  
}
```

Inner loop:



Approx. Miss Rates

a	b	c
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (L=2, S=0, MR=1.25) jik (L=2, S=0, MR=1.25) kij (L=2, S=1, MR=0.5)

```
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;  
  }  
}
```

```
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;  
  }  
}
```

```
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];  
  }  
}
```

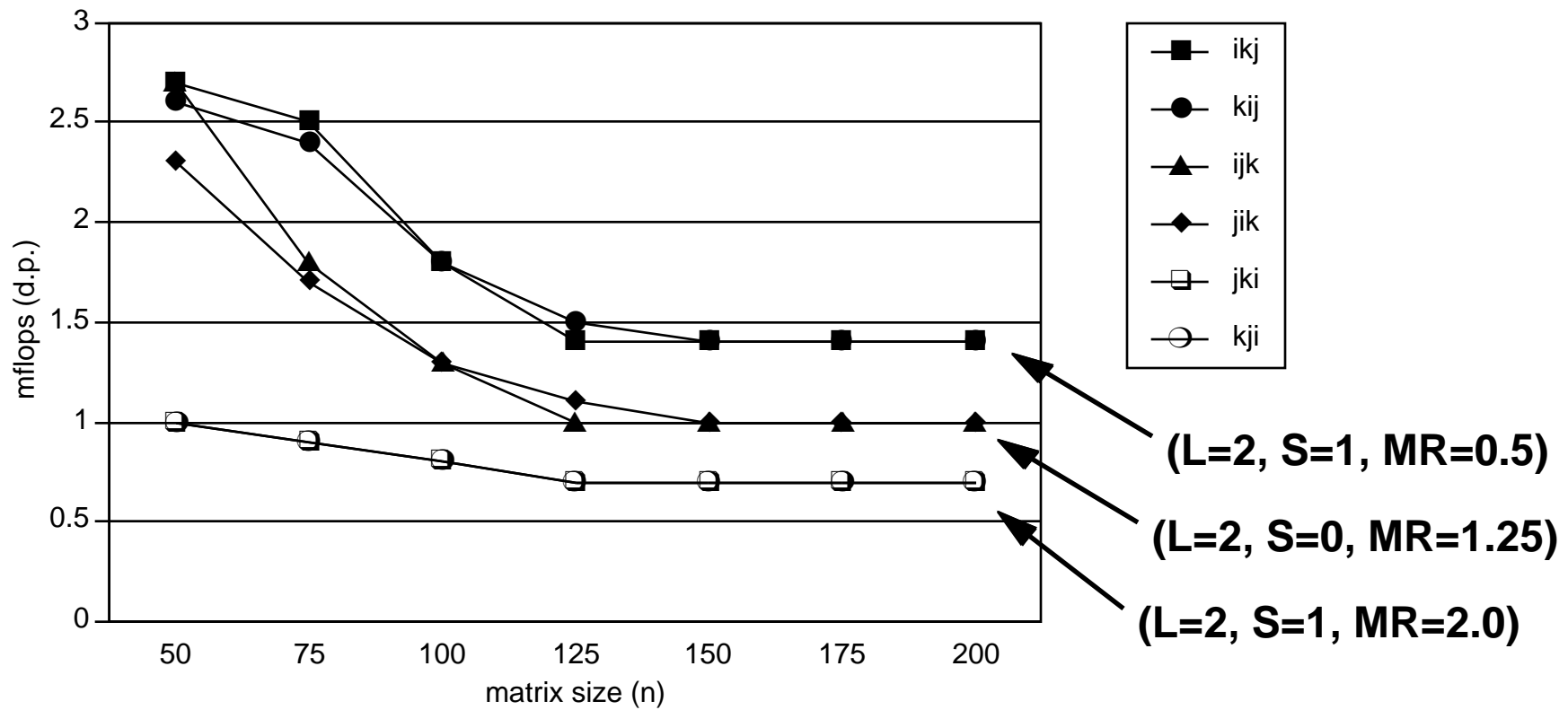
jki (L=2, S=1, MR=2.0)

```
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;  
  }  
}
```

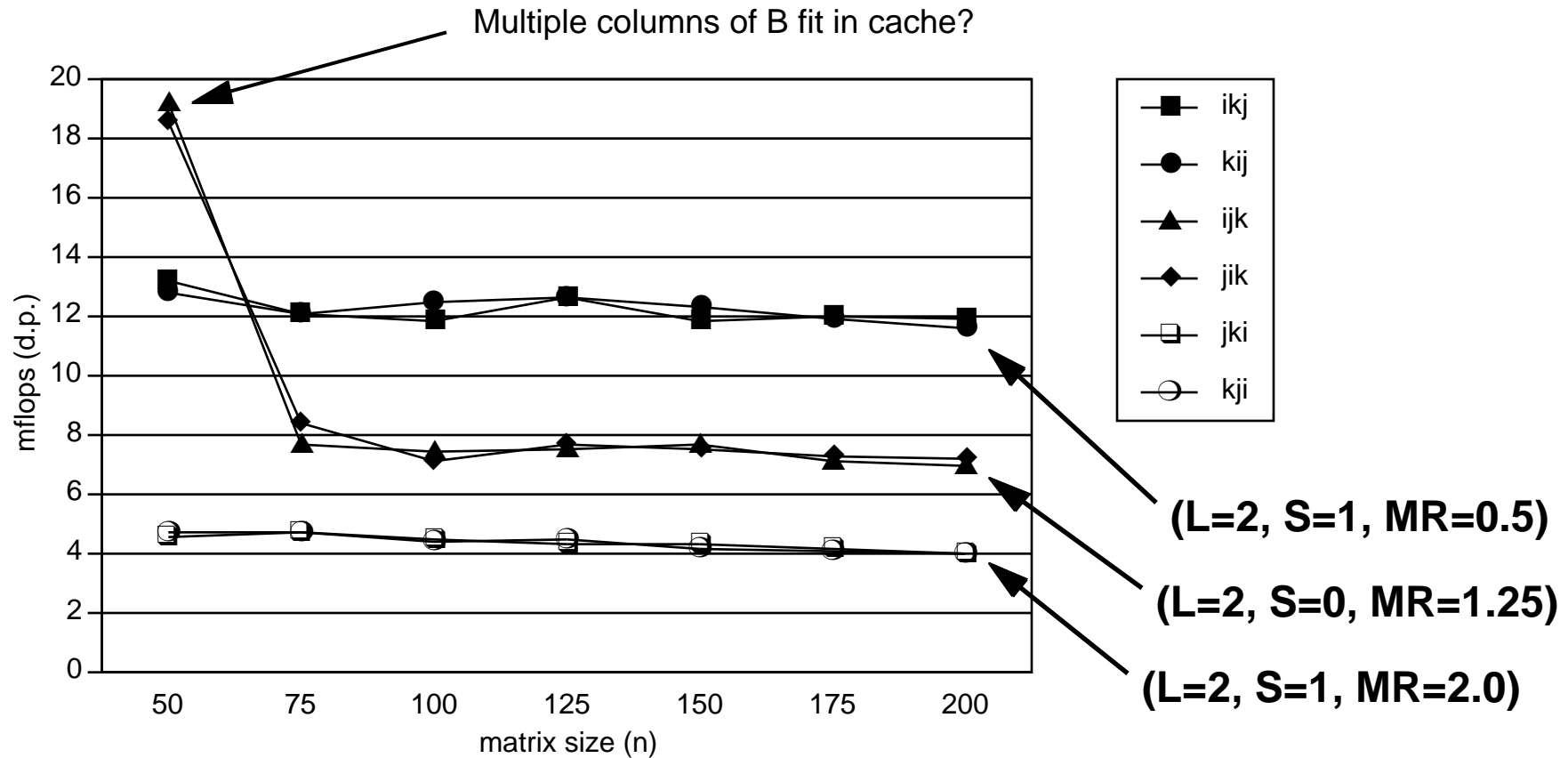
kji (L=2, S=1, MR=2.0)

```
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;  
  }  
}
```

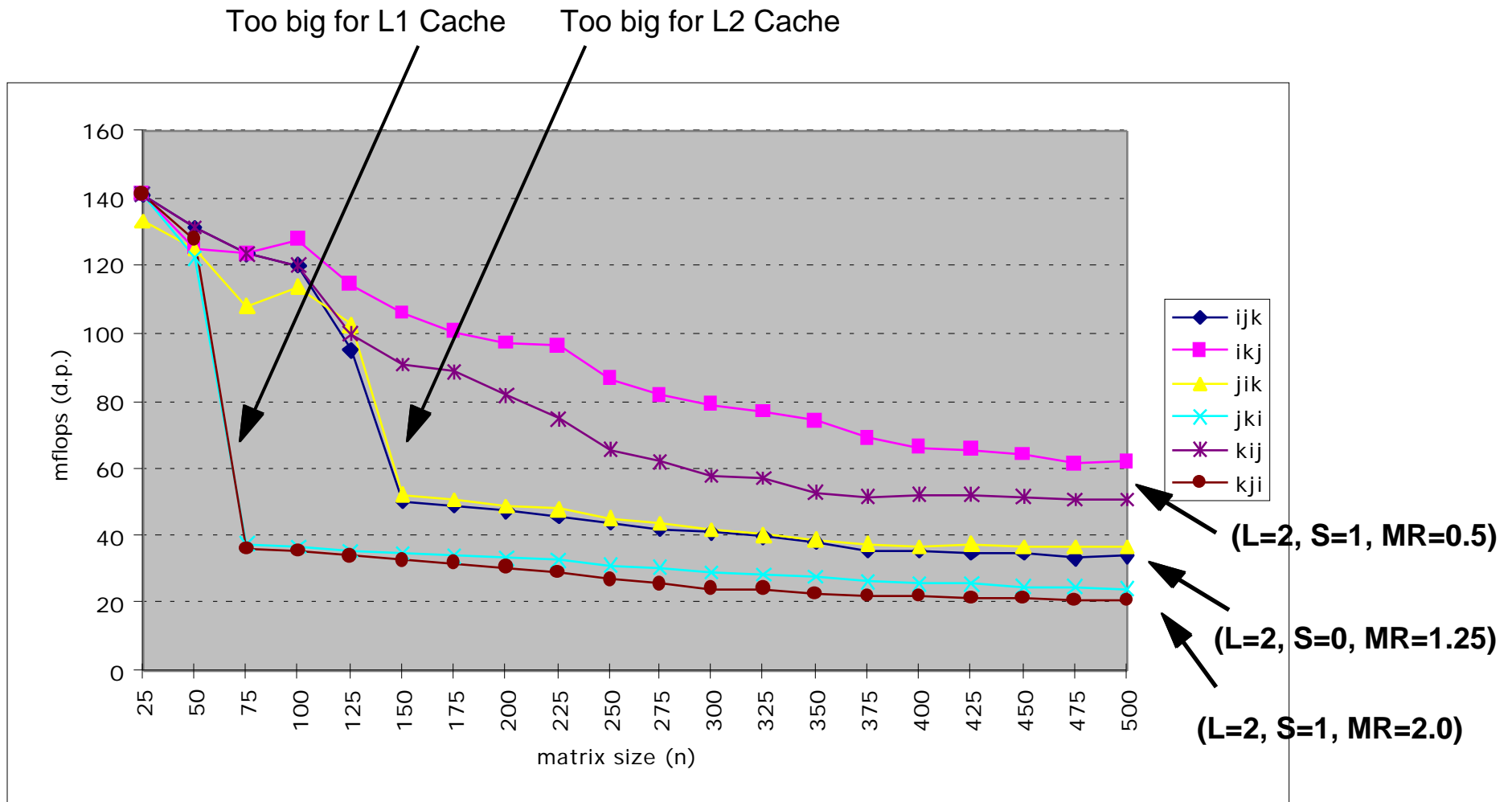
Matmult performance (DEC5000)



Matmult Performance (Sparc20)



Matmult Performance (Alpha 21164)



Block Matrix Multiplication

Example $n=8$, $B = 4$:

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

Key idea: Sub-blocks (i.e., A_{ij}) 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-1,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

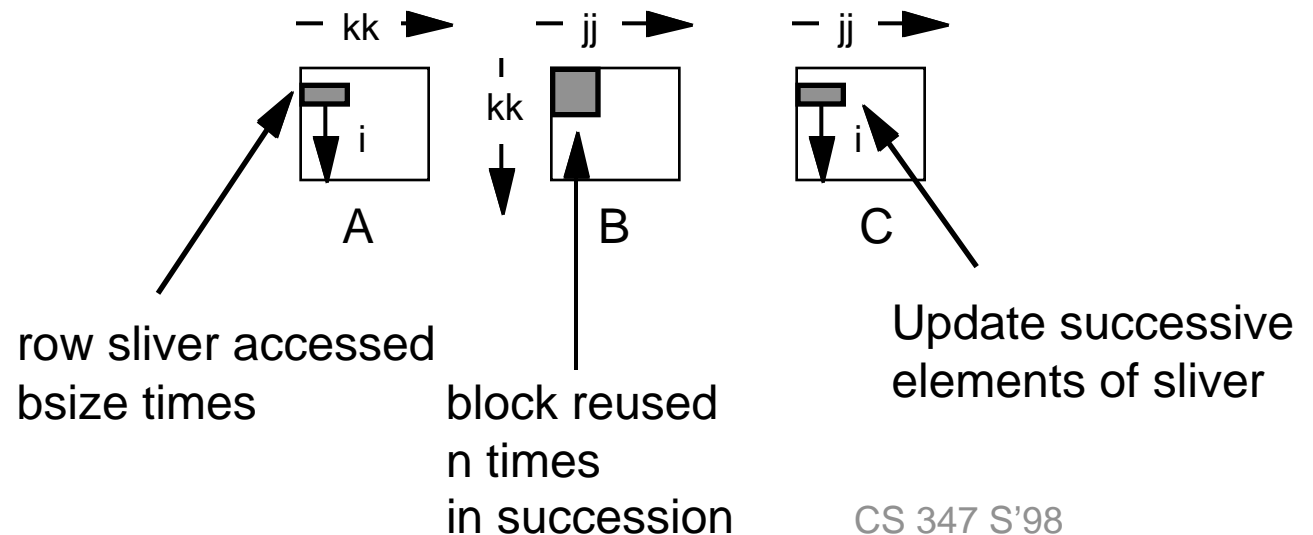
- Innermost loop pair multiplies 1 X bsize sliver of A times bsize X bsize block of B and accumulates into 1 X bsize sliver of C
- Loop over i steps through n row slivers of A & C, using same B

```

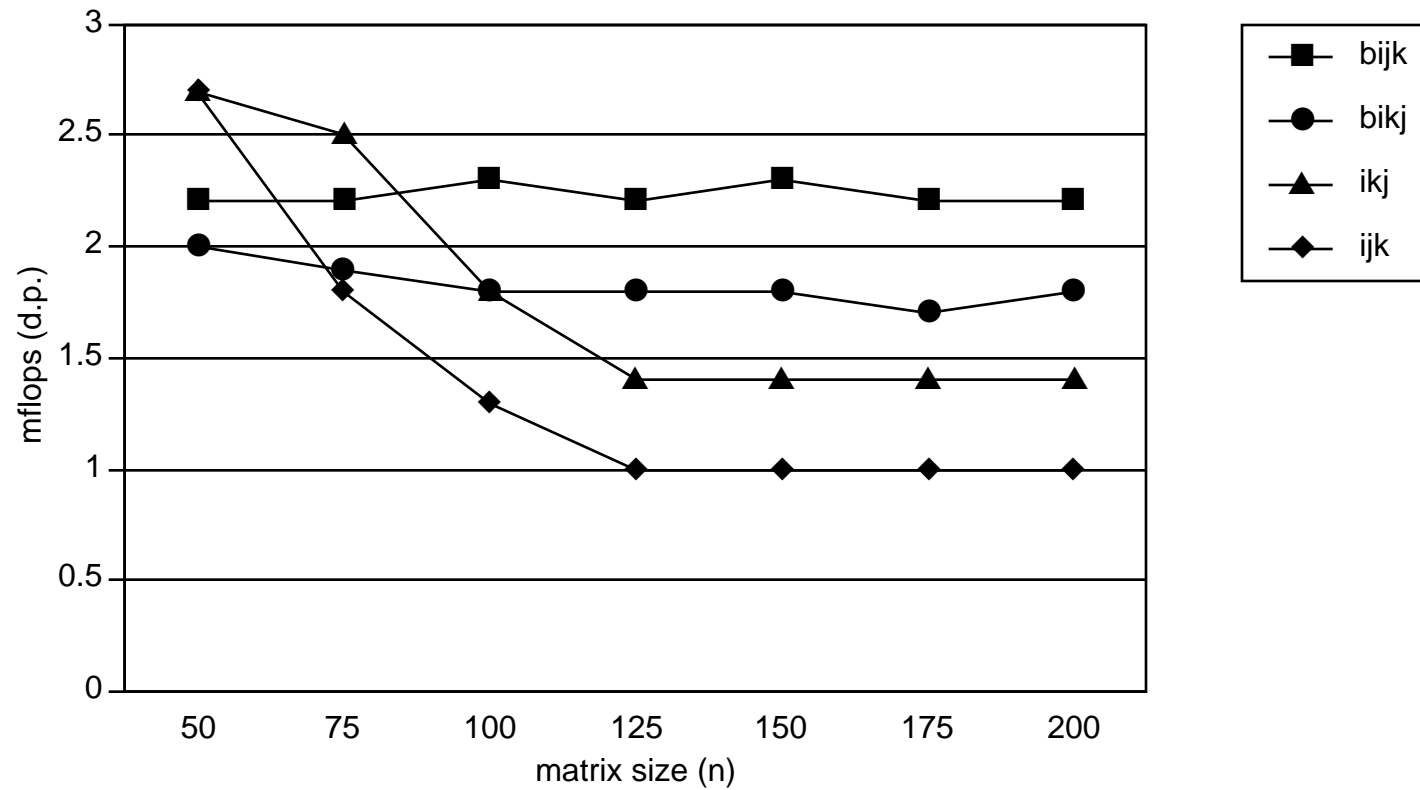
for (i=0; i<n; i++) {
  for (j=jj; j < min(jj+bsize,n); j++) {
    sum = 0.0
    for (k=kk; k < min(kk+bsize,n); k++) {
      sum += a[i][k] * b[k][j];
    }
    c[i][j] += sum;
  }
}

```

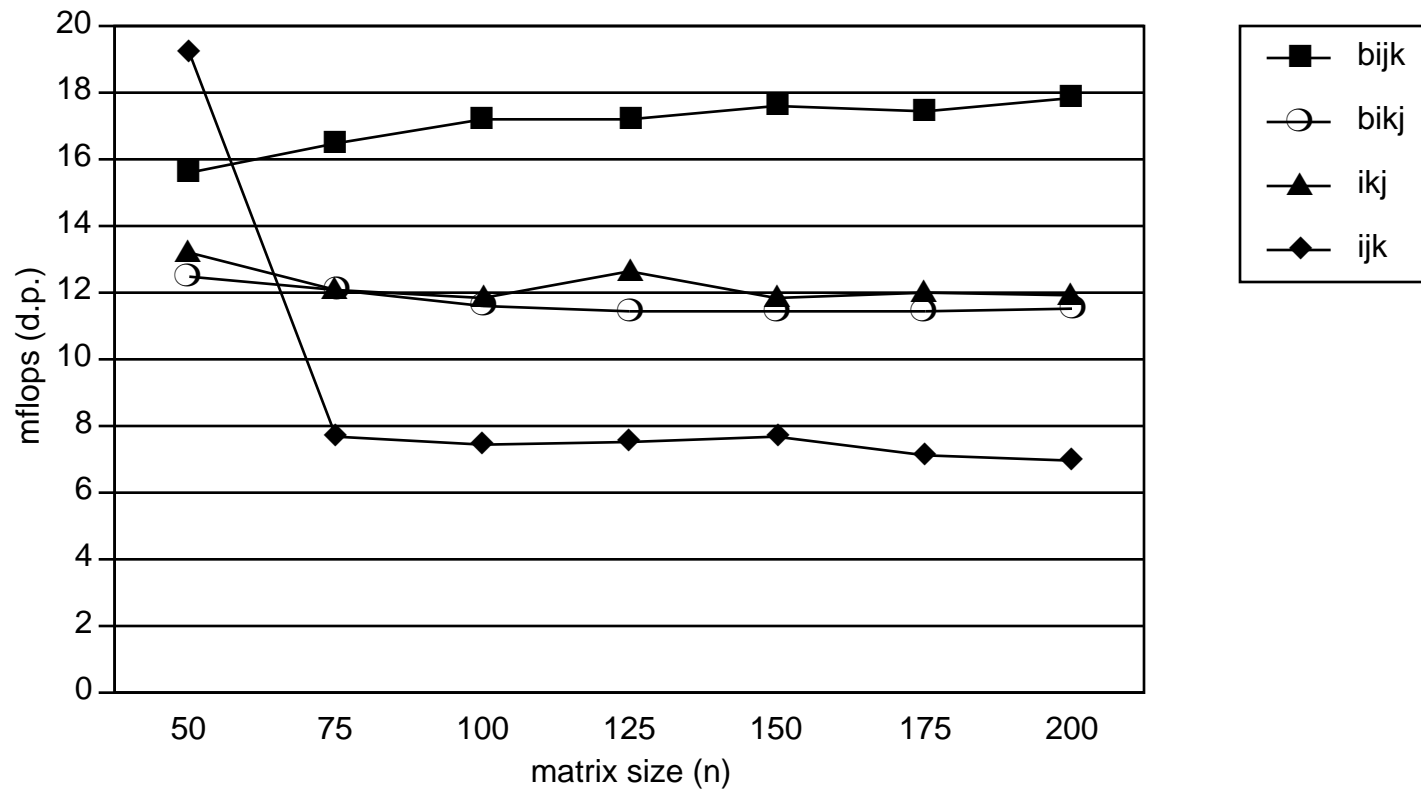
Innermost
Loop Pair



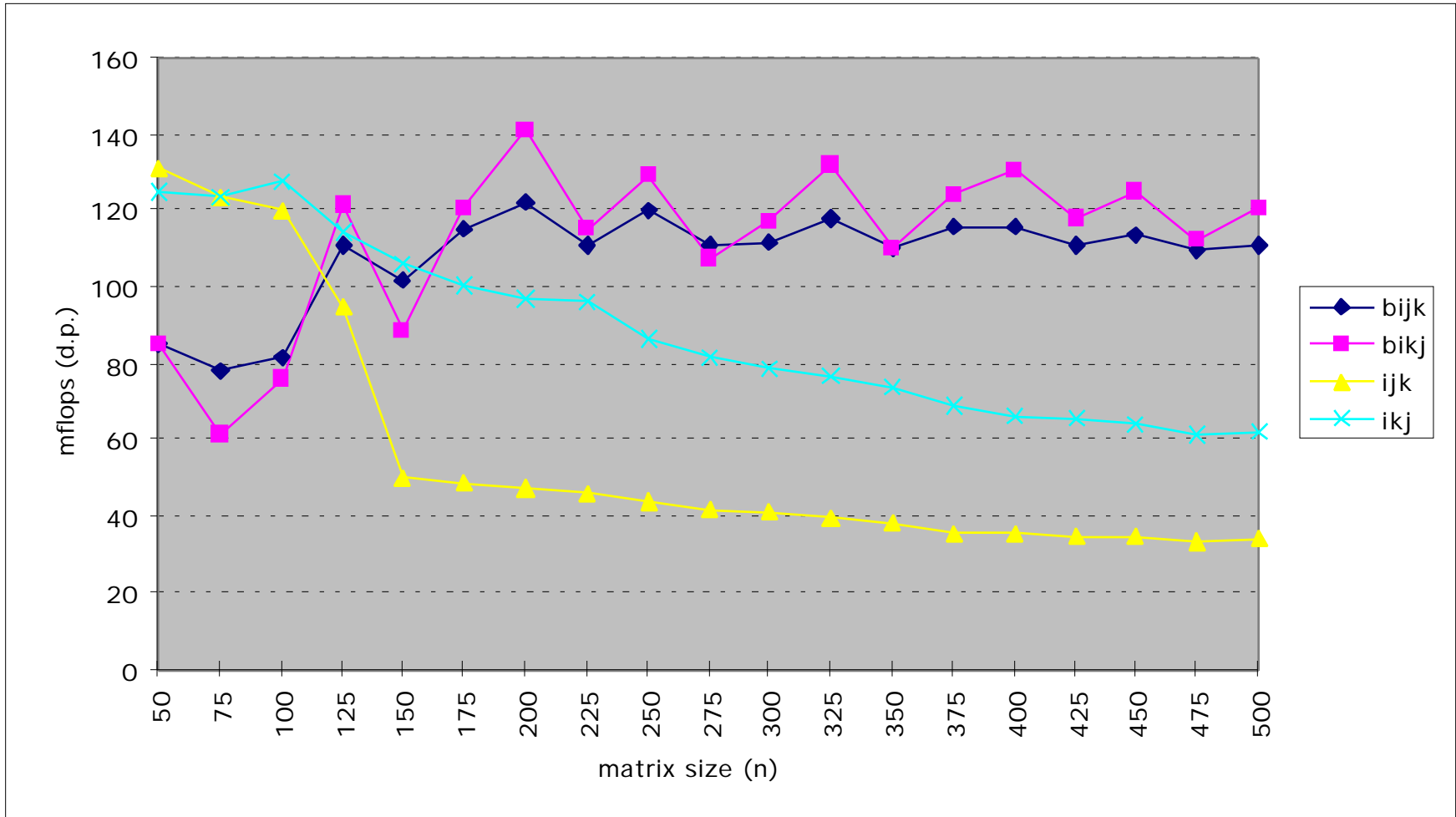
Blocked matmult perf (DEC5000)



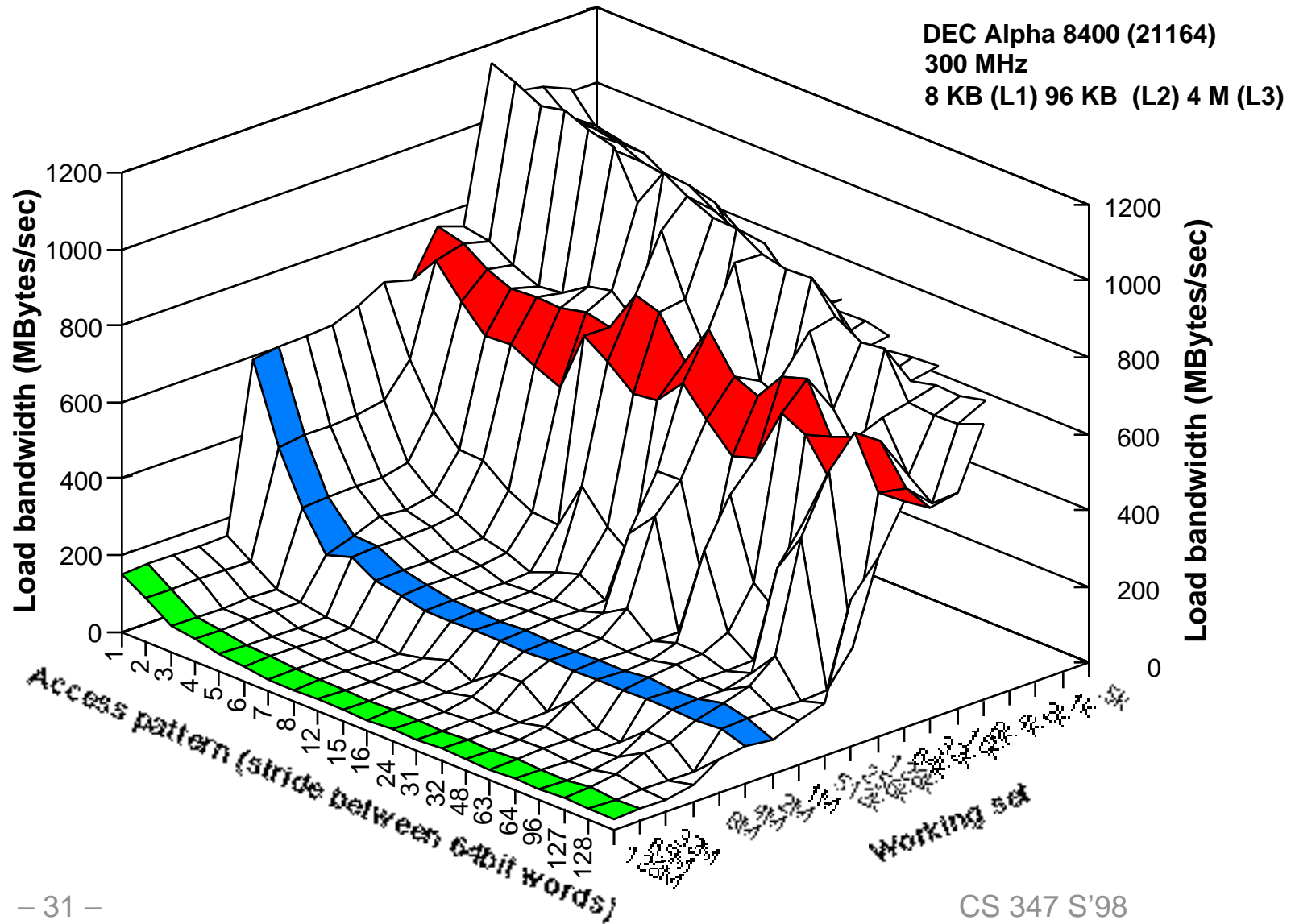
Blocked matmult perf (Sparc20)



Blocked matmult perf (Alpha 21164)



The Memory Mountain Range



Effects Seen in Mountain Range

Cache Capacity

- See sudden drops as increase working set size

Cache Block Effects

- Performance degrades as increase stride
 - Less spatial locality
- Levels off
 - When reach single access per block

Bandwidth Matching

Challenge

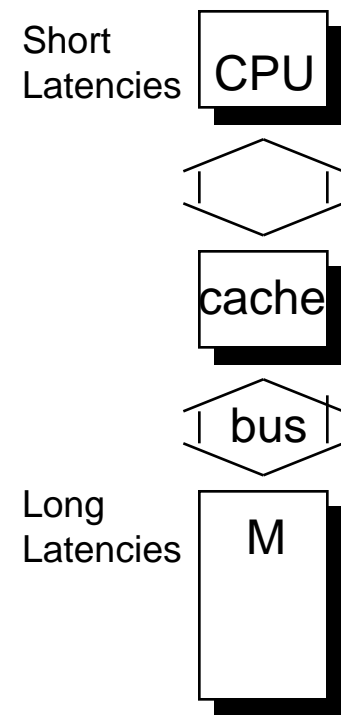
- CPU works with short cycle times
- DRAM (relatively) long cycle times
- *How can we provide enough bandwidth between processor & memory?*

Effect of Caching

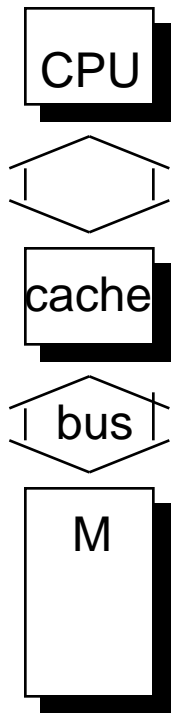
- Caching greatly reduces amount of traffic to main memory
- But, sometimes need to move large amounts of data from memory into cache

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

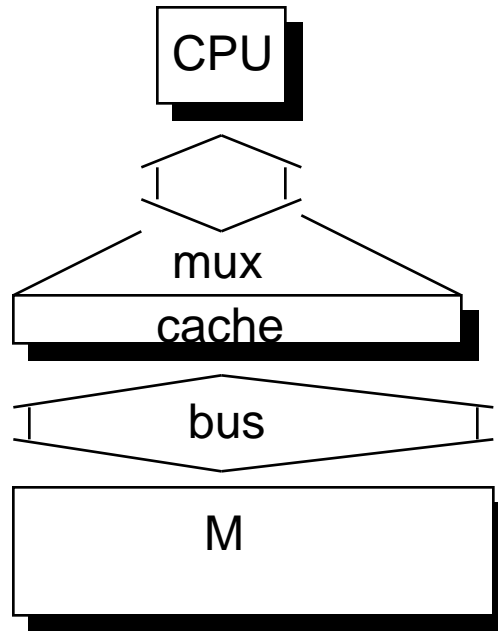


High Bandwidth Memory Systems



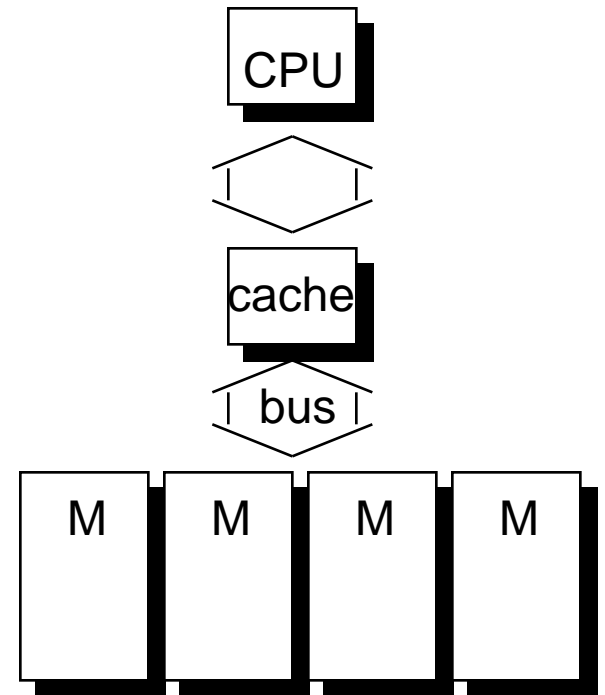
Solution 1
High BW DRAM

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