Topics

• Memory Hierarchy
• Cache Design
• Optimizing Program Performance
Computer System

Processor

Cache

Memory-I/O bus

Memory

I/O controller

Disk

I/O controller

Display

I/O controller

Network

interrupts
Levels in a typical memory hierarchy

- CPU (register reference) to Cache (cache reference)
- Cache to Memory (memory reference)
- Memory to Disk (disk memory reference)

- Size: 200 B to 32KB -- 4MB to 128 MB to 10GB
- Speed: 5 ns to 6 ns to 100 ns to 10 ms
- $/Mbyte: $256/MB to $2/MB to $0.10/MB
- Block size: 4 B to 16 B to 4 KB

larger, slower, cheaper
Caches:
- L1 data
- L1 instruction
- L2 unified
- TLB
- Branch history
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.

Access word \( w \) in block \( a \) (hit)

Access word \( v \) in block \( b \) (miss)

High Level

\[
\begin{array}{c}
\text{w} \\
\text{a} \\
\text{a} \\
\end{array}
\]

Low Level

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{a} \\
\text{b} \\
\end{array}
\]

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{a} \\
\text{b} \\
\end{array}
\]
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

• **Data**
  – Reference array elements in succession (spatial)

• **Instruction**
  – Reference instructions in sequence (spatial)
  – Cycle through loop repeatedly (temporal)

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```
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: $0, \ldots, 2^{n-1}$.
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.,

\[
\text{address} = \begin{array}{ccc}
\text{t} & \text{s} & \text{b} \\
\text{tag} & \text{set index} & \text{offset}
\end{array}
\]

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

\[ t = 1 \quad s = 3 \quad b = 1 \]

- \( 2^s = 8 \) sets of blocks
- \( 2^t = 2 \) blocks/set
- \( 2^b = 2 \) addresses/block.
Partitioning address spaces

\[ t=1 \quad s=3 \quad b=1 \]

\[ \begin{array}{c|c|c}
0 & 110 & 1 \\
\end{array} \]

\[ 2^s = 8 \text{ sets of blocks} \]
\[ 2^t = 2 \text{ blocks/set} \]
\[ 2^b = 2 \text{ addresses/block}. \]

Set 110

Block 0

Offset 1
Partitioning address spaces

\[ t = 2 \quad s = 2 \quad b = 1 \]

\[ \begin{array}{ccc}
10 & 11 & 0 \\
\end{array} \]

\[ 2^s = 4 \text{ sets of blocks} \]
\[ 2^t = 4 \text{ blocks/set} \]
\[ 2^b = 2 \text{ addresses/block.} \]
Partitioning address spaces

\[2^s = 4 \text{ sets of blocks}\]
\[2^t = 4 \text{ blocks/set}\]
\[2^b = 2 \text{ addresses/block}.\]
Partitioning address spaces

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

\[ t=4 \quad s=0 \quad b=1 \]

- \[ 2^s = 1 \text{ set of blocks} \]
- \[ 2^t = 16 \text{ blocks/set} \]
- \[ 2^b = 2 \text{ addresses/block} \]
Basic cache organization

**Address space** ($N = 2^n$ bytes)  
**Cache** ($C = S \times E \times B$ bytes)

### Address
($n = t + s + b$ bits)

- **t** bits: tag
- **s** bits: set
- **b** bits: block

### Cache block
(cache line)

- **Valid bit**: 1 bit
- **tag**: **t** bits
- **data**: $B = 2^b$ bytes (block size)

**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)

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xx</td>
<td>x</td>
</tr>
</tbody>
</table>

- **cache size:**
  - C = 8 data bytes
- **line size:**
  - B = \(2^b\) = 2 bytes/line

- **direct mapped cache**
  - E = 1 entry/set
  - \(S = 2^s = 4\) sets

1. Determine set from middle bits
2. If something already there, knock it out
3. Put new block in cache
### Direct Mapped Cache Simulation

- 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

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>XX</td>
<td>X</td>
</tr>
</tbody>
</table>

**0 [0000] (miss)**

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>m[1] m[0]</td>
</tr>
</tbody>
</table>

**13 [1101] (miss)**

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>m[1] m[0]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>m[13] m[12]</td>
</tr>
</tbody>
</table>

**8 [1000] (miss)**

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>m[9] m[8]</td>
</tr>
</tbody>
</table>

**0 [0000] (miss)**

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>m[1] m[0]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>m[13] m[12]</td>
</tr>
</tbody>
</table>
Direct Mapped Cache Implementation (DECStation 3100)

31 30 29 ................. 19 18 17 16 15 14 13 ................. 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>tag</th>
<th>set</th>
<th>byte offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid</td>
<td>tag (16 bits)</td>
<td>data (32 bits)</td>
</tr>
</tbody>
</table>

? sets

hit

data
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 cache

E = 2 entries/set
S = 2^l = 2 sets
# 2-Way Set Associative Simulation

- **t=2**, **s=1**, **b=1**
- **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]

### Cache State:

<table>
<thead>
<tr>
<th>Address</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
<td>00</td>
<td>m[1]</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>00</td>
<td>m[0]</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>0100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td>0101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
<td>10</td>
<td>m[9]</td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td>0111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td>1001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>10</td>
<td>m[8]</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td>1011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>00</td>
<td>m[1]</td>
<td>1110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 0 (miss)
- 13 (miss)
- 8 (miss) (LRU replacement)
- 0 (miss) (LRU replacement)
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)

\[ \begin{array}{ccc}
    t=3 & s=0 & b=1 \\
    xxx & x & \\
\end{array} \]

Fully associative cache

\[ E = 4 \text{ entries/set} \]

\[ S = 2^s = 1 \text{ set} \]

cache size:
\[ C = 8 \text{ data bytes} \]

line size:
\[ B = 2^b = 2 \text{ bytes/line} \]
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]

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

\[
\begin{array}{|c|c|c|}
  \hline
  v & \text{tag} & \text{data} \\
  \hline
  1 & 00 & m[1] m[0] \\
  \hline
  \end{array}
\]

\(t=3\) s=0 b=1

\[
\begin{array}{|c|c|c|}
  \hline
  v & \text{tag} & \text{data} \\
  \hline
  1 & 00 & m[1] m[0] \\
  \hline
  \end{array}
\]

\(1\) (miss) set \(\varnothing\)

\[
\begin{array}{|c|c|c|}
  \hline
  v & \text{tag} & \text{data} \\
  \hline
  \hline
  \end{array}
\]

\(2\) (miss)

\[
\begin{array}{|c|c|c|}
  \hline
  v & \text{tag} & \text{data} \\
  \hline
  1 & 000 & m[1] m[0] \\
  \hline
  \end{array}
\]

\(3\) (miss)

\[
\begin{array}{|c|c|c|}
  \hline
  v & \text{tag} & \text{data} \\
  \hline
  \hline
  \end{array}
\]

\[
\begin{array}{|c|c|c|}
  \hline
  v & \text{tag} & \text{data} \\
  \hline
  1 & 100 & m[9] m[8] \\
  \hline
  \end{array}
\]

\(8\) (miss)

CS 740 F'98
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 $\Rightarrow \sim E \log E$ 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

<table>
<thead>
<tr>
<th></th>
<th>size:</th>
<th>speed:</th>
<th>$/Mbyte:</th>
<th>block size:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>200 B</td>
<td>5 ns</td>
<td>$256/MB</td>
<td>4 B</td>
</tr>
<tr>
<td></td>
<td>8 KB</td>
<td>5 ns</td>
<td>$2/MB</td>
<td>16 B</td>
</tr>
<tr>
<td>Memory</td>
<td>1M SRAM</td>
<td>6 ns</td>
<td>$256/MB</td>
<td>32 KB</td>
</tr>
<tr>
<td></td>
<td>128 MB DRAM</td>
<td>100 ns</td>
<td>$2/MB</td>
<td>4 KB</td>
</tr>
<tr>
<td></td>
<td>10 GB</td>
<td>10 ms</td>
<td>$0.10/MB</td>
<td></td>
</tr>
</tbody>
</table>

larger, slower, cheaper

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

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

```c
/* 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;
    }
}
```
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80000</td>
<td>a[0][0]</td>
</tr>
<tr>
<td>0x80008</td>
<td>a[0][1]</td>
</tr>
<tr>
<td>0x80010</td>
<td>a[0][2]</td>
</tr>
<tr>
<td>0x80018</td>
<td>a[0][3]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x807F8</td>
<td>a[0][255]</td>
</tr>
<tr>
<td>0x80800</td>
<td>a[1][0]</td>
</tr>
<tr>
<td>0x80808</td>
<td>a[1][1]</td>
</tr>
<tr>
<td>0x80810</td>
<td>a[1][2]</td>
</tr>
<tr>
<td>0x80818</td>
<td>a[1][3]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x80FF8</td>
<td>a[1][255]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFFFF8</td>
<td>a[255][255]</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix multiplication (jik)

```c
/* 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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix multiplication (kij)

```c
/* 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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Inner loop:
- (i,k) - Fixed
- (k,*) - Row-wise
- (i,*) - Row-wise
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:

Approx. Miss Rates

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix multiplication (jki)

```c
/* 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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Inner loop:

- Column-wise
- Fixed
- Column-wise
Matrix multiplication (kji)

```c
/* 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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Inner loop:
- (i,k) Column-wise
- (k,j) Fixed
- (j,k) Column-wise
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;
    }
}
```

**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)

![Graph showing performance vs. matrix size for different matrix multiplications.]

- (L=2, S=1, MR=0.5)
- (L=2, S=0, MR=1.25)
- (L=2, S=1, MR=2.0)
Matmult Performance (Sparc20)

Multiple columns of B fit in cache?

- ikj
- kij
- ijk
- jik
- jki
- kji

(L=2, S=1, MR=0.5)
(L=2, S=0, MR=1.25)
(L=2, S=1, MR=2.0)
Matmult Performance (Alpha 21164)

Too big for L1 Cache
Too big for L2 Cache

(L=2, S=1, MR=0.5)
(L=2, S=0, MR=1.25)
(L=2, S=1, MR=2.0)
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.

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \\
C_{12} = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} = A_{21}B_{11} + A_{22}B_{21} \\
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
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

```c
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)