Computer Systems Organization

Memory Hierarchy

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Programmer’s Wish List

- Private
- Infinitely large
- Infinitely fast
- Non-volatile
- Inexpensive

Programs are getting bigger faster than memories.
Example Memory Hierarchy

- **L0:** CPU registers
  - Hold words retrieved from the L1 cache.

- **L1:** L1 cache (SRAM)
  - Hold cache lines retrieved from the L2 cache.

- **L2:** L2 cache (SRAM)
  - Hold cache lines retrieved from L3 cache.

- **L3:** L3 cache (SRAM)
  - Hold cache lines retrieved from main memory.

- **L4:** Main memory (DRAM)
  - Hold disk blocks retrieved from local disks.

- **L5:** Local secondary storage (local disks)
  - Hold files retrieved from disks on remote servers.

- **L6:** Remote secondary storage (e.g., Web servers)
  - Larger, slower, and cheaper (per byte) storage devices.
## SRAM vs DRAM Summary

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>4 or 6</td>
<td>1X</td>
<td>No</td>
<td>Maybe</td>
<td>100x</td>
<td>Cache memories</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>10X</td>
<td>Yes</td>
<td>Yes</td>
<td>1X</td>
<td>Main memories, frame buffers</td>
</tr>
</tbody>
</table>
Question: Who Cares About the Memory Hierarchy?

“Moore’s Law”

Processor-Memory Performance Gap:
(grows 50% / year)

CPU-DRAM Gap

μProc
60%/yr.

DRAM
7%/yr.
Disk
Funny facts:
• It took 51 years to reach 1TB and 2 years to reach 2TB!
• IBM introduced the first hard disk drive to break the 1 GB barrier in 1980.

IBM Disk 350, size 5MB, circa 50s

Hard Disks

- spinning platter of special material
- mechanical arm with read/write head must be close to the platter to read/write data
- data is stored magnetically
- storage capacity is commonly between 100GB - 3TB
- disks are random access meaning data can be read/written anywhere on the disk

The disk surface spins at a fixed rotational rate

The read/write head is attached to the end of the arm and flies over the disk surface on a thin cushion of air

By moving radially, the arm can position the read/write head over any track
What's Inside A Disk Drive?

- Spindle
- Arm
- Actuator
- Platters
- Electronics (including a processor and memory!)
- SCSI connector

Image courtesy of Seagate Technology
Disk Drives

• To access data:
  — seek time: position head over the proper track
  — rotational latency: wait for desired sector
  — transfer time: grab the data (one or more sectors)
A Conventional Hard Disk Structure
Hard Disk Architecture

- **Surface** = group of tracks
- **Track** = group of sectors
- **Sector** = group of bytes
- **Cylinder**: several tracks on corresponding surfaces
Disk Sectors and Access

• Each sector records
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - Error correcting code (ECC)
    • Used to hide defects and recording errors
  - Synchronization fields and gaps

• Access to a sector involves
  - Queuing delay if other accesses are pending
  - Seek: move the heads
  - Rotational latency
  - Data transfer
  - Controller overhead
Disks: Other Issues

- Average seek and rotation times are helped by locality.
- Disk performance improves about 10%/year.
- Capacity increases about 60%/year.
- Example of disk controllers:
  - SCSI, ATA, SATA
Flash Storage

• Nonvolatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)
Flash Types

• NOR flash: bit cell like a NOR gate
  - Random read/write access
  - Used for instruction memory in embedded systems

• NAND flash: bit cell like a NAND gate
  - Denser (bits/area), but block-at-a-time access
  - Cheaper per GB
  - Used for USB keys, media storage, …

• Flash bits wears out after 1000’s of accesses
  - Not suitable for direct RAM or disk replacement
  - Wear leveling: remap data to less used blocks
Solid-State Disk

Requests to read and write logical disk blocks

I/O bus

Solid State Disk (SSD)

Flash translation layer

Flash memory

Block 0
Page 0  Page 1  ...  Page P-1

...  ...  ...

Block B-1
Page 0  Page 1  ...  Page P-1

Typically:
• pages are 512–4KB in size
• a block consists of 32–128 pages
• A blocks wears out after roughly 100,000 repeated writes.
• Once a block wears out it can no longer be used.
Main Memory
(DRAM ... For now!)
DRAM

• packaged in **memory modules** that plug into expansion slots on the main system board (motherboard)

• Example package: 168-pin dual inline memory module (DIMM)
  – transfers data to and from the memory controller in 64-bit chunks
addr (row = i, col = j)

64 MB memory module consisting of 8 8Mx8 DRAMs

Memory controller

64-bit double word at main memory address A

64-bit doubleword to CPU chip
Conventional DRAM Organization

- $d \times w$ DRAM:
  - $dw$ total bits organized as $d$ supercells of size $w$ bits
Reading DRAM Supercell (2,1)

Step 1(a): Row access strobe (RAS) selects row 2.

Step 1(b): Row 2 copied from DRAM array to row buffer.
Reading DRAM Supercell (2,1)

Step 2(a): Column access strobe (CAS) selects column 1.
Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.
Memory Modules

64 MB memory module consisting of eight 8Mx8 DRAMs

addr (row = i, col = j)

Memory controller

64-bit word main memory address A

64-bit word
Enhanced DRAMs

• Basic DRAM cell has not changed since its invention in 1966.
  – Commercialized by Intel in 1970.
• DRAM cores with better interface logic and faster I/O:
  – Synchronous DRAM (SDRAM)
    • Uses a conventional clock signal instead of asynchronous control
    • Allows reuse of the row addresses (e.g., RAS, CAS, CAS, CAS)
  – Double data-rate synchronous DRAM (DDR SDRAM)
    • Double edge clocking sends two bits per cycle per pin
    • Different types distinguished by size of small prefetch buffer:
      – DDR (2 bits), DDR2 (4 bits), DDR3 (8 bits)
    • By 2010, standard for most server and desktop systems
    • Intel Core i7 supports only DDR3 SDRAM
# Storage Trends

## SRAM

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$/MB</td>
<td>2,900</td>
<td>320</td>
<td>256</td>
<td>100</td>
<td>75</td>
<td>60</td>
<td>25</td>
<td>116</td>
</tr>
<tr>
<td>access (ns)</td>
<td>150</td>
<td>35</td>
<td>15</td>
<td>3</td>
<td>2</td>
<td>1.5</td>
<td>1.3</td>
<td>115</td>
</tr>
</tbody>
</table>

## DRAM

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$/MB</td>
<td>880</td>
<td>100</td>
<td>30</td>
<td>1</td>
<td>0.1</td>
<td>0.06</td>
<td>0.02</td>
<td>44,000</td>
</tr>
<tr>
<td>access (ns)</td>
<td>200</td>
<td>100</td>
<td>70</td>
<td>60</td>
<td>50</td>
<td>40</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>typical size (MB)</td>
<td>0.256</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>2,000</td>
<td>8,000</td>
<td>16,000</td>
<td>62,500</td>
</tr>
</tbody>
</table>

## Disk

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$/GB</td>
<td>100,000</td>
<td>8,000</td>
<td>300</td>
<td>10</td>
<td>5</td>
<td>0.3</td>
<td>0.03</td>
<td>3,333,333</td>
</tr>
<tr>
<td>access (ms)</td>
<td>75</td>
<td>28</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>typical size (GB)</td>
<td>0.01</td>
<td>0.16</td>
<td>1</td>
<td>20</td>
<td>160</td>
<td>1,500</td>
<td>3,000</td>
<td>300,000</td>
</tr>
</tbody>
</table>
## CPU Clock Rates

![Inflection point in computer history when designers hit the “Power Wall”]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>80286</td>
<td>80386</td>
<td>Pentium</td>
<td>P-4</td>
<td>Core 2</td>
<td>Core i7(n)</td>
<td>Core i7(h)</td>
<td></td>
</tr>
<tr>
<td>Clock rate (MHz)</td>
<td>6</td>
<td>20</td>
<td>150</td>
<td>3,300</td>
<td>2,000</td>
<td>2,500</td>
<td>3,000</td>
<td>500</td>
</tr>
<tr>
<td>Cycle time (ns)</td>
<td>166</td>
<td>50</td>
<td>6</td>
<td>0.30</td>
<td>0.50</td>
<td>0.4</td>
<td>0.33</td>
<td>500</td>
</tr>
<tr>
<td>Cores</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Effective cycle time (ns)</td>
<td>166</td>
<td>50</td>
<td>6</td>
<td>0.30</td>
<td>0.25</td>
<td>0.10</td>
<td>0.08</td>
<td>2,075</td>
</tr>
</tbody>
</table>

(n) Nehalem processor
(h) Haswell processor
Cache Memory
Large gap between processor speed and memory speed
A...B...C of Cache
Cache Analogy

• **Hungry! must eat!**
  - **Option 1: go to refrigerator**
    • Found → eat!
    • Latency = 1 minute
  - **Option 2: go to store**
    • Found → purchase, take home, eat!
    • Latency = 20-30 minutes
  - **Option 3: grow food!**
    • Plant, wait ... wait ... wait ..., harvest, eat!
    • Latency = ~250,000 minutes (~ 6 months)
What Do We Gain?

Let \( m = \) cache access time, \( M: \) main memory access time
\( p = \) probability that we find the data in the cache

\[
\text{Average access time} = p \cdot m + (1-p)(m+M) = m + (1-p) M
\]

We need to increase \( p \)
Problem

Given the following:
   Cache: 1 cycle access time
   Main memory: 100 cycle access time

What is the average access time for 100 memory references if you measure that 90% of the cache accesses are hits
Class Problem

For application X, you measure that 40% of the operations access memory. The non-memory access operations take one cycle. You also measure that 90% of the memory references hit in the cache. Whenever a cache miss occurs, the processor is stalled for 20 cycles to transfer the block from memory into the cache. A cache hit takes one cycle. What is the average operations time?
Cache Organization

Fundamental Unit: CACHE BLOCK
Purpose: HOLD DATA

A Simple Cache
A More Complex Cache
(similar to having several caches)
A Single CACHE BLOCK (or LINE)
Parameter: BLOCK SIZE (or LINE SIZE)
A Single CACHE SET (equivalence class)
Parameter: ASSOCIATIVITY
Given 8 cache blocks ...

Direct Mapped  
2-Way Set Associative  
4-Way Set Associative  

8-Way Set Associative  
(Fully Associative, or Content-Addressable Memory)
Basic Cache Design

- Cache memory can copy data from any part of main memory
  - It has 2 parts:
    - The **TAG (CAM)** holds the memory address
    - The **BLOCK (SRAM)** holds the memory data

- Accessing the cache:
  - Compare the reference address with the tag
    - If they match, get the data from the cache block
    - If they don’t match, get the data from main memory
Direct Mapped Cache

Example: 32-bit address
Direct Mapped Cache
So…What is a cache?

- Small, fast storage used to improve average access time to slow memory.
- Exploits **spatial** and **temporal** locality
- In computer architecture, almost everything is a cache!
  - Registers a cache on variables
  - First-level cache a cache on second-level cache
  - Second-level cache a cache on memory
  - Memory a cache on disk (virtual memory)
  - etc…

![Diagram of cache hierarchy](image-url)

- Slower - Cheaper - Bigger - Faster
- More Expensive - Smaller
Localities:
Why Cache Is a Good Idea?

• **Spatial locality**: If block $k$ is accessed, it is likely that block $k+1$ will be accessed

• **Temporal locality**: If block $k$ is accessed, it is likely that it will be accessed again
### Set 0:

- **Valid**
- **Tag**
- 0 1 \( \ldots \) \( B-1 \)
- \( E \) lines per set

### Set 1:

- **Valid**
- **Tag**
- 0 1 \( \ldots \) \( B-1 \)
- \( E \) lines per set

### Set \( S-1 \):

- **Valid**
- **Tag**
- 0 1 \( \ldots \) \( B-1 \)
- \( E \) lines per set

### Cache size:

\[ C = B \times E \times S \text{ data bytes} \]
Problem

Show the breakdown of the address for the following cache configuration:

- 32 bit address
- 16K cache
- Direct-mapped cache
- 32-byte blocks

| tag | set index | block offset |
Problem

Show the breakdown of the address for the following cache configuration:

- 32 bit address
- 32K cache
- 4-way set associative cache
- 32-byte blocks

| tag | set index | block offset |
Cache

- **Associativity**: (DM, 2-way, 4-way,...FA)
- **Block size**
- **Replacement strategy**: (LRU, FIFO, LFU, RANDOM)
- **Size**
Design Issues

- What to do in case of hit/miss?
- Block size
- Associativity
- Replacement algorithm
- Improving performance
Hits vs. Misses

• Read hits
  - this is what we want!

• Read misses
  - stall the CPU, fetch block from memory, deliver to cache, restart

• Write hits:
  - can replace data in cache and memory (*write-through*)
  - write the data only into the cache (*write-back* the cache later)

• Write misses:
  - read the entire block into the cache, then write the word
Improving Cache Performance

1. Reduce the miss rate,
2. Reduce the miss penalty,
3. Reduce power consumption (won’t be discussed here)
Reducing Misses

• Classifying Misses: 3 Cs
  - **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. Also called *cold start misses* or *first reference misses*. (Misses in even an Infinite Cache)
  - **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, **capacity misses** will occur due to blocks being discarded and later retrieved.
  - **Conflict**—If block-placement strategy is set associative or direct mapped, **conflict misses** (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called *collision misses* or *interference misses*. 
How Can We Reduce Misses?

1) **Change Block Size:**

2) **Change Associativity:**

3) **Increase Cache Size**
Increasing the block size tends to decrease miss rate:
Decreasing miss ratio with associativity

One-way set associative
(direct mapped)

<table>
<thead>
<tr>
<th>Block</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two-way set associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Four-way set associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Eight-way set associative (fully associative)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
</table>
Implementation of 4-way set associative
Effect of Associativity on Miss Rate

![Graph showing the effect of associativity on miss rate. The graph plots the miss rate against different associativities: One-way, Two-way, Four-way, and Eight-way. The y-axis represents the miss rate in percentages (0% to 15%). The x-axis represents the associativity levels: 1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, and 128 KB. The graph shows a decreasing trend in miss rate as the associativity increases.](image)
Reducing Miss Penalty

Write Policy 1:
Write-Through vs Write-Back

- **Write-through:** all writes update cache and underlying memory/cache
  - Can always discard cached data - most up-to-date data is in memory
  - Cache control bit: only a valid bit
- **Write-back:** all writes simply update cache
  - Can’t just discard cached data - may have to write it back to memory
  - Cache control bits: both valid and dirty bits

**Other Advantages:**
- **Write-through:**
  - memory (or other processors) always have latest data
  - Simpler management of cache
- **Write-back:**
  - much lower bandwidth, since data often overwritten multiple times
  - Better tolerance to long-latency memory?
Reducing Miss Penalty
Write Policy 2:
Write Allocate vs Non-Allocate
(What happens on write-miss)

• **Write allocate**: allocate new cache line in cache
  – Usually means that you have to do a “read miss” to fill in rest of the cache-line!

• **Write non-allocate** (or “write-around”):
  – Simply send write data through to underlying memory/cache - don’t allocate new cache line!
Decreasing miss penalty with multilevel caches

• Add a second (and third) level cache:
  - often primary cache is on the same chip as the processor
  - use SRAMs to add another cache above primary memory (DRAM)
  - miss penalty goes down if data is in 2nd level cache

• Using multilevel caches:
  - try and optimize the hit time on the 1st level cache
  - try and optimize the miss rate on the 2nd level cache
What about Replacement Algorithm?

- LRU
- LFU
- FIFO
- Random
How to write cache friendly code?
Is The Following Code Cache Friendly?

```c
1    int sumvec(int v[N])
2    {
3        int i, sum = 0;
4
5        for (i = 0; i < N; i++)
6            sum += v[i];
7        return sum;
8    }
```
Matrix Multiplication Example

- **Description:**
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - O(N^3) total operations
  - 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;
    }
}
```

Variable `sum` held in register

---

```c
matmult/mm.c
```
Miss Rate Analysis for Matrix Multiply

• Assume:
  – Block size = 32B (big enough for four doubles)
  – Matrix dimension (N) is very large
  – 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

- **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) > `sizeof(a_{ij})` bytes, exploit spatial locality
- **Stepping through rows in one column:**
  - `for (i = 0; i < n; i++)`
    - `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - `miss rate = 1` (i.e. 100%)
Matrix Multiplication (ijk)

```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;
    }
}
```

Misses per inner loop iteration:

<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
    }
}
```

Misses per inner loop iteration:

<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];
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Inner loop:

- **Fixed**
- **Row-wise**

Diagram:

- $(i,k)$
- $(k,*)$
- $(i,*)$
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];
    }
}

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Inner loop:

A \rightarrow B \rightarrow C

Fixed \hspace{1cm} Row-wise \hspace{1cm} Row-wise
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:

\[
\begin{array}{ccc}
A & B & C \\
1.0 & 0.0 & 1.0 \\
\end{array}
\]
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:

Misses per inner loop iteration:

<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>
Conclusions

• The computer system’s storage is organized as a hierarchy.
• The reason of this hierarchy is to try to get an memory that is very fast, cheap, and almost infinite.
• A good programmer must try to make the code cache friendly → make the common case cache friendly → locality