The Memory Hierarchy

2015 Fall

Yusung Kim
yskim525@gmail.com
Motivation

■ A simple model of a computer system
  - CPU executes instructions, and a memory system is a linear array of bytes
  - CPU can access each memory location in a constant amount of time

■ In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times.
  - CPU registers
  - Cache memories near by the CPU
  - Main memory
  - Local and remote disks
Motivation

- **Understand the memory hierarchy**
  - A big impact on the performance of your programs

- **To access data**
  - If in a CPU register, zero cycles
  - If in a cache, 1 to 30 cycles
  - If in a main memory, 50 to 200 cycles
  - If in disk, tens of millions of cycles!

- **Learn about basic storage technologies, and how to analyze your C programs for performance.**
Storage technologies and trends

- Early computers had a few kilobytes of random-access memory.

- IBM PC in 1982, with 10 Megabyte disk.

- The amount of storage was increasing by a factor of 2 every couple of years.

- Currently terabyte disks in PC
Random-Access Memory (RAM)

- **SRAM (Static RAM)**
  - Faster, more expensive than DRAM
  - Used for cache memories

- **DRAM (Dynamic RAM)**
  - Used for main memory

- Typically, a desktop system has no more than a few megabytes of SRAM, but hundreds or thousands of megabytes of DRAM
## SRAM vs DRAM Summary

<table>
<thead>
<tr>
<th></th>
<th>Transistors per bit</th>
<th>Relative access time</th>
<th>Persistent?</th>
<th>Sensitive?</th>
<th>Relative cost</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>6</td>
<td>1×</td>
<td>Yes</td>
<td>No</td>
<td>100×</td>
<td>Cache memory</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>10×</td>
<td>No</td>
<td>Yes</td>
<td>1×</td>
<td>Main mem, frame buffers</td>
</tr>
</tbody>
</table>
Nonvolatile Memories

- DRAM and SRAM are volatile memories
  - Lose information if powered off.

- Nonvolatile memories retain value even if powered off
  - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Erasable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically erasable PROM (EEPROM): electronic erase capability
  - Flash memory: EEPROMs. with partial (block-level) erase capability
    - Wears out after about 100,000 reprograms

- Programs stored in ROM devices are often referred to as firmware.
  - BIOS (basic input output system)
  - Graphics cards and disk drive controllers
Traditional Bus Structure
Connecting CPU and Memory

- A **bus** is a collection of parallel wires that carry address, data, and control signals.
- **Buses** are typically shared by multiple devices.
Memory Read Transaction (1)

- CPU places address A on the system bus. The I/O bridge passes the signal to the memory bus.

Load operation: `movl A, %rax`
Memory Read Transaction (2)

- Main memory reads A from the memory bus, retrieves word \( x \), and places it on the bus.

Load operation: `movq A, %rax`
CPU senses the data on the system bus, reads it from the bus and copies it into register %rax.

Load operation: `movq A, %rax`
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.

\[
\text{Store operation: movl } \%\text{rax}, A
\]
Memory Write Transaction (2)

- CPU places data word $y$ on the bus.

Store operation: $\text{movq } %rax, A$
Memory Write Transaction (3)

- Main memory reads data word $y$ from the bus and stores it at address $A$.

Store operation: $\text{movq} \ %rax, A$

Diagram:
- Register file
  - %rax
  - $y$
- ALU
- Bus interface
- I/O bridge
- Main memory
  - 0
  - $y$
  - A
Disk Storage

- The order of hundreds to thousands of gigabytes, as opposed to the hundreds or thousands of megabytes in a RAM based memory.

- It takes on the order of milliseconds to read information from a disk:
  - a hundred thousand times longer than from DRAM
  - a million times longer than from SRAM
What’s Inside A Disk Drive?

Image courtesy of Seagate Technology
Disk Geometry

- Disks consist of **platters**, each with two sides (**surfaces**).
- Each surface consists of concentric rings called **tracks**.
- Each track consists of **sectors** separated by gaps.
Disk Geometry (Multiple-Platter View)

- Aligned tracks form a cylinder.

Cylinder $k$ is the collection of the six instances of track $k$
Disk Capacity

- **Capacity**: maximum number of bits that can be stored.
  - Vendors express capacity in units of gigabytes (GB), where 1 GB = $10^9$ Bytes.

- Capacity is determined by these technology factors:
  - **Track density** (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
  - **Recording density** (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
  - **Areal density** (bits/in$^2$): product of recording and track density.
Recording zones

- Modern disks partition tracks into disjoint subsets called **recording zones**
  
  ✓ Each track in a zone has the same number of sectors.
  
  ✓ Each zone has a different number of sectors/track, outer zones have more sectors/track than inner zones.
  
  ✓ So we use **average** number of sectors/track when computing capacity.
Computing Disk Capacity

Capacity = (# bytes/sector) x (avg. # sectors/track) x
          (# tracks/surface) x (# surfaces/platter) x
          (# platters/disk)

Example:
- 512 bytes/sector
- 300 sectors/track (on average)
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

Capacity = 512 x 300 x 20000 x 2 x 5
          = 30,720,000,000
          = 30.72 GB
Disk Operation (Single-Platter View)

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.

By moving radially, the arm can position the read/write head over any track.
Disk Operation (Multi-Platter View)

Read/write heads move in unison from cylinder to cylinder

- Spindle
- Arm
Disk Access

Head in position above a track
Disk Access

Rotation is counter-clockwise
Disk Access – Read

About to read blue sector
Disk Access – Read

After BLUE read

After reading blue sector
Disk Access – Read

After BLUE read

Red request scheduled next
Disk Access – Seek

Seek to red’s track
Disk Access – Rotational Latency

After **BLUE** read  
Seek for **RED**  
Rotational latency

Wait for red sector to rotate around
Disk Access – Read

After BLUE read  
Seek for RED  
Rotational latency  
After RED read

Complete read of red
Disk Access – Service Time Components

After BLUE read

Seek for RED

Rotational latency

After RED read

Data transfer

Seek

Rotational latency

Data transfer
Disk Access Time

Average time to access some target sector approximated by:

time for access = time for the average seek time
  + time for the average rotational latency
  + time for the average transfer time

\[
T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}
\]
Disk Access Time

- **T_access** = T_avg seek + T_avg rotation + T_avg transfer

- **Seek time** (T_avg seek)
  - Time to position heads over cylinder containing target sector.
  - Typical T_avg seek is 3—9 ms

- **Rotational latency** (T_avg rotation)
  - Time waiting for first bit of target sector to pass under r/w head.
  - T_avg rotation = \( \frac{1}{2} \times \frac{1}{\text{RPMs}} \times 60 \ \text{sec/1 min} \)
  - Typical T_avg rotation = 7200 RPMs

- **Transfer time** (T_avg transfer)
  - Time to read the bits in the target sector.
  - T_avg transfer = \( \frac{1}{\text{RPM}} \times \frac{1}{(\text{avg # sectors/track})} \times 60 \ \text{secs/1 min} \)
Disk Access Time Example

- **Given:**
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms
  - Avg # sectors/track = 400

- **Derived:**
  - \( T_{\text{avg rotation}} = \frac{1}{2} \times (\text{60 secs/7200 RPM}) \times 1000 \text{ ms/sec} = 4 \text{ ms.} \)
  - \( T_{\text{avg transfer}} = \frac{60}{7200} \text{ RPM} \times \frac{1}{400} \text{ secs/track} \times 1000 \text{ ms/sec} = 0.02 \text{ ms} \)
  - \( T_{\text{access}} = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms} \)

- **Important points:**
  - Access time dominated by **seek time** and **rotational latency**.
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
  - Disk is about 2,500 times slower than DRAM, and 40,000 times slower than SRAM !!!
Logical Disk Blocks

- Modern disks present a simpler abstract view of the complex sector geometry:
  - The set of available sectors is modeled as a sequence of **B-sized logical blocks** (0, 1, 2, ... B-1)

- Mapping between logical blocks and actual (physical) sectors
  - Maintained by hardware/firmware device called disk controller.
  - Converts requests for logical blocks into actual (physical) sectors.

- Disk controller also performs some intelligent functions
  - Caching, prefetching, scheduling, etc.
I/O Bus

Expansion slots for other devices such as network adapters.
CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) of the disk.
Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.
When the DMA transfer completes, the disk controller notifies the CPU with an interrupt.
Solid State Disks (SSDs)

- Pages: 512B to 4KB, Blocks: 32 to 128 pages
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- A block wears out after about 100,000 repeated writes.
SSD Performance Characteristics

<table>
<thead>
<tr>
<th></th>
<th>Sequential read tput</th>
<th>550 MB/s</th>
<th>Sequential write tput</th>
<th>470 MB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random read tput</td>
<td>365 MB/s</td>
<td></td>
<td>Random write tput</td>
<td>303 MB/s</td>
</tr>
<tr>
<td>Avg seq read time</td>
<td>50 us</td>
<td></td>
<td>Avg seq write time</td>
<td>60 us</td>
</tr>
</tbody>
</table>

- **Sequential access faster than random access**
  - Common theme in the memory system

- **Random writes are somewhat slower**
  - Erasing a block takes a long time (~1 ms)
  - Modifying a block page requires all other pages to be copied to new block
  - In earlier SSDs, the read/write gap was much larger.

Source: Intel SSD 730 product specification.
SSD Tradeoffs vs Rotating Disks

- **Advantages**
  - No moving parts $\rightarrow$ faster, less power, more rugged (durable)

- **Disadvantages**
  - Have the potential to wear out
    - Mitigated by “wear leveling logic” in flash translation layer
    - E.g. Intel SSD 730 guarantees 128 petabyte ($128 \times 10^{15}$ bytes) of writes before they wear out
  - In 2015, about 30 times more expensive per byte

- **Applications**
  - MP3 players, smart phones, laptops
  - Beginning to appear in desktops and servers
The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.
Locality to the Rescue!

- Different storage technologies have different price and performance trade-offs.

- DRAM and disk performance are lagging behind CPU performance.

- The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as **locality**
Locality

- **Principle of Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently.

- **Temporal locality**:  
  - Recently referenced items are likely to be referenced again in the near future.

- **Spatial locality**:  
  - Items with nearby addresses tend to be referenced close together in time.
Locality Example

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

- **Data references**
  - Reference array elements in succession (stride-1 reference pattern).
  - Reference variable `sum` each iteration.

- **Instruction references**
  - Reference instructions in sequence.
  - Cycle through loop repeatedly.
Qualitative Estimates of Locality

- **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

- **Question:** Does this function have good locality with respect to array `a`?

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum;
}
```
Locality Example

**Question:** Does this function have good locality with respect to array \( a \)?

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>( a_{00} )</td>
<td>( a_{01} )</td>
<td>( a_{02} )</td>
<td>( a_{10} )</td>
<td>( a_{11} )</td>
<td>( a_{12} )</td>
</tr>
<tr>
<td>Access order</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
Summary of Locality

- Referencing the same variables enjoys good temporal locality

- With stride-k reference patterns, the smaller the stride the better the spatial locality.

- Loops have good temporal and spatial locality with respect to instruction fetches.

- Let’s learn about caching with respect to good locality.
Memory Hierarchies

- Some fundamental properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.

- These fundamental properties complement each other beautifully.

- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.
**Example Memory Hierarchy**

- **L0:** CPU registers hold words retrieved from the L1 cache.
- **L1:** L1 cache holds cache lines retrieved from the L2 cache.
- **L2:** L2 cache holds cache lines retrieved from L3 cache.
- **L3:** L3 cache holds cache lines retrieved from main memory.
- **L4:** Main memory holds disk blocks retrieved from local disks.
- **L5:** Local disks hold files retrieved from disks on remote servers.
- **L6:** Remote secondary storage (e.g., Web servers).

Smaller, faster, and costlier (per byte) storage devices:
- Local secondary storage (local disks)
- Remote secondary storage (e.g., Web servers)

Larger, slower, and cheaper (per byte) storage devices:
- Main memory (DRAM)
Caches

- **Cache**: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

- **Fundamental idea of a memory hierarchy:**
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
  
  e.g. The main memory serves as a cache for data on the local disk.

- **Why do memory hierarchies work?**
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
General Cache Concepts

Level $k$

Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Level $k+1$

Larger, slower, cheaper memory viewed as partitioned into “blocks”
General Cache Concepts: Hit

Data in block b is needed

Block b is at level $k$:

Hit!
General Cache Concepts: Miss

Data in block b is needed

Block b is not at level k: Miss!

Block b is fetched from level $k+1$

Block b is stored in cache
- **Placement policy:** determines where b goes
- **Replacement policy:** determines which block gets evicted (victim)
General Caching Concepts:
Types of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty.

- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - When multiple level k+1 blocks map to the same level k block,
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
Cache Memories

- In early computer systems, only three level memory hierarchies; registers, main memory, and disk.
- Because of the gap between CPU and main memory, systems require a small, fast SRAM cache memory, called L1 & L2 cache.
- CPU looks first for data in L1, then in L2, then in main memory.
# Examples of Caching in the Mem. Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What is Cached?</th>
<th>Where is it Cached?</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-8 bytes words</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware MMU</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-byte blocks</td>
<td>On-Chip L1</td>
<td>4</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-byte blocks</td>
<td>On-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB pages</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware + OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Disk cache</td>
<td>Disk sectors</td>
<td>Disk controller</td>
<td>100,000</td>
<td>Disk firmware</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>
Anatomy of a Real Cache Hierarchy

Intel Core i7

Processor package

Core 0

- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

Core 3

- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

8 MB (L3 unified cache, shared by all cores)

Main memory

Each 32 KB

256 KB
Cache Performance Metrics

- **Miss rate**
  - Fraction of memory references not found in cache (misses/references)
  - Typically 3-10% for L1
  - Hit rate is computed as 1 – miss rate.

- **Hit time**
  - The time to deliver a word in the cache to the CPU.
  - Hit time is on the order of several clock cycles for L1 caches.

- **Miss penalty**
  - Additional time required because of a miss.
  - At L1 cache, 10 cycles from L2
  - At L3 cache, 25-100 cycles from main memory
Issues with a Writ Hit

■ When writing a word $w$ that is already cached (a write hit), update its copy of $w$ at the cache, and what?

- Write-through: immediately write $w$’s cache block to the next lower level.

  Simple, but causing bus traffic with every write

- Write-back: update at level $k$ but defer the update at the next lower level ($k+1$) until it is evicted from level $k$.

  Reducing bus traffic, but need an additional bit to check whether the cache block has been modified.
Issues with a Write Miss

When writing a word $w$ that is not cached, how to deal with the write miss?

- Write-allocate: loads corresponding block from the next lower level into the cache and then updates the cache block.

  Can exploit temporal and spatial locality of writes, but every miss results in a block transfer.

- No-write-allocate: directly write the word to the next lower level.

  Simple, but..?
Writing Cache Friendly Code

- Programs with better locality will tend to have lower miss rates, and programs with lower miss rates will tend to run faster than programs with higher miss rates.

- Thus, good programmers should always try to write code that is cache friendly, in the sense that it has good locality.
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)
Examples:

- cold cache, 4-byte words, 4-word cache blocks

```c
int sumvec(int v[N])
{
    int i, sum = 0;
    for (i = 0; i < N; i++)
        sum += v[i];
    return sum;
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>v[i]</th>
<th>Access order, [h]it or [m]iss</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[m]</td>
<td>1 [m]</td>
</tr>
<tr>
<td>1</td>
<td>[h]</td>
<td>2 [h]</td>
</tr>
<tr>
<td>2</td>
<td>[h]</td>
<td>3 [h]</td>
</tr>
<tr>
<td>3</td>
<td>[h]</td>
<td>4 [h]</td>
</tr>
<tr>
<td>4</td>
<td>[m]</td>
<td>5 [m]</td>
</tr>
<tr>
<td>5</td>
<td>[h]</td>
<td>6 [h]</td>
</tr>
<tr>
<td>6</td>
<td>[h]</td>
<td>7 [h]</td>
</tr>
<tr>
<td>7</td>
<td>[h]</td>
<td>8 [h]</td>
</tr>
</tbody>
</table>

**Miss rate = \( \frac{1}{4} = 25\% \)**
```c
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
            sum += a[i][j];
    }
    return sum;
}
```

**Miss rate = \( \frac{1}{4} = 25% \)**

<table>
<thead>
<tr>
<th>a[i][j]</th>
<th>j = 0</th>
<th>j = 1</th>
<th>j = 2</th>
<th>j = 3</th>
<th>j = 4</th>
<th>j = 5</th>
<th>j = 6</th>
<th>j = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0</td>
<td>1 [m]</td>
<td>2 [h]</td>
<td>3 [h]</td>
<td>4 [h]</td>
<td>5 [m]</td>
<td>6 [h]</td>
<td>7 [h]</td>
<td>8 [h]</td>
</tr>
<tr>
<td>i = 1</td>
<td>9 [m]</td>
<td>10 [h]</td>
<td>11 [h]</td>
<td>12 [h]</td>
<td>13 [m]</td>
<td>14 [h]</td>
<td>15 [h]</td>
<td>16 [h]</td>
</tr>
</tbody>
</table>
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}

Miss rate = 100%

<table>
<thead>
<tr>
<th>a[i][j]</th>
<th>j = 0</th>
<th>j = 1</th>
<th>j = 2</th>
<th>j = 3</th>
<th>j = 4</th>
<th>j = 5</th>
<th>j = 6</th>
<th>j = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0</td>
<td>1 [m]</td>
<td>5 [m]</td>
<td>9 [m]</td>
<td>13 [m]</td>
<td>17 [m]</td>
<td>21 [m]</td>
<td>25 [m]</td>
<td>29 [m]</td>
</tr>
<tr>
<td>i = 1</td>
<td>2 [m]</td>
<td>6 [m]</td>
<td>10 [m]</td>
<td>14 [m]</td>
<td>18 [m]</td>
<td>22 [m]</td>
<td>26 [m]</td>
<td>30 [m]</td>
</tr>
<tr>
<td>i = 2</td>
<td>3 [m]</td>
<td>7 [m]</td>
<td>11 [m]</td>
<td>15 [m]</td>
<td>19 [m]</td>
<td>23 [m]</td>
<td>27 [m]</td>
<td>31 [m]</td>
</tr>
<tr>
<td>i = 3</td>
<td>4 [m]</td>
<td>8 [m]</td>
<td>12 [m]</td>
<td>16 [m]</td>
<td>20 [m]</td>
<td>24 [m]</td>
<td>28 [m]</td>
<td>32 [m]</td>
</tr>
</tbody>
</table>
Exploiting Locality in Your Programs

- Understand the nature of the memory hierarchy.

- Focus your attention on the inner loops, where the bulk of the computations and memory accesses occur.

- Try to maximize the spatial locality in your programs by reading data objects sequentially in the order they are stored in memory.

- Try to maximize the temporal locality in your programs by using a data object as often as possible once it has been read from memory.
Summary

- Faster storage technologies are more expensive per bit, and have smaller capacities.

- The speed gap between CPU, memory and mass storage continues to widen.

- Memory hierarchies based on caching narrow the gap by exploiting locality.

- Well-written programs exhibit a property called locality, and optimize cache performance.
  - Temporal locality
  - Spatial locality