Memory Hierarchy

CSE 2431: Introduction to Operating Systems
Reading: Chap. 6, [CSAPP]
Motivation

- Up to this point we have relied on a simple model of a computer system - a CPU with a simple memory which holds instructions and data for the CPU.
- In reality, a computer system contains a hierarchy of storage devices with different costs, capacities, and access times.
- With a memory hierarchy, a faster storage device at one level of the hierarchy acts as a staging area for a slower storage device at the next lower level.
- Software that is well-written takes advantage of the hierarchy accessing the faster storage device at a particular level more frequently than the storage at the next level.
- As a programmer, understanding the memory hierarchy will result in better performance of applications.
Outline

• Storage Technologies
• Locality
• Memory Hierarchy
• Cache Memories
• Writing Cache-friendly Code
• Impact of Caches on Program Performance
Storage Technologies

- Random-Access Memory
- Disk Storage
- Solid State Disks
- Storage Technology Trends
Random-Access Memory (RAM)

Features
- Basic storage unit is usually a cell (one bit per cell)
- RAM is traditionally packaged as a chip
- Multiple chips form memory
- Static RAM (SRAM)
  - Each cell implemented with a six-transistor circuit
  - Holds value as long as power is maintained - volatile
  - Relatively insensitive to disturbances such as electrical noise, radiation, etc.
  - Faster and more expensive than DRAM
- Dynamic RAM (DRAM)
  - Each bit stored as charge on a capacitor
  - Value must be refreshed every 10–100 ms: volatile
  - Sensitive to disturbances
  - Slower and cheaper than SRAM
## SRAM vs DRAM

<table>
<thead>
<tr>
<th></th>
<th>Trans. per bit</th>
<th>Access time</th>
<th>Needs refresh?</th>
<th>Sensitive?</th>
<th>Cost</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SRAM</strong></td>
<td>4 or 6</td>
<td>1X</td>
<td>No</td>
<td>No</td>
<td>100x</td>
<td>Cache memories</td>
</tr>
<tr>
<td><strong>DRAM</strong></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>
Conventional DRAM Organization

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

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 8M×8 DRAMs

addr (row = i, col = j)

Memory controller

64-bit doubleword at main memory address A

64-bit doubleword
Enhanced DRAMs

• Enhanced DRAMs have optimizations that improve the speed with which the basic DRAM cells can be accessed.

• Examples:
  – Fast page mode DRAM (FPM DRAM)
  – Extended data out DRAM (EDO DRAM)
  – Synchronous DRAM (SDRAM)
  – Double Data-Rate Synchronous DRAM (DDR SDRAM)
  – Rambus DRAM (RDRAM)
  – Video RAM (VRAM)
Nonvolatile Memory (1)

Features

• Information retained if supply voltage is turned off
• Collectively referred to as read-only memories (ROM) although some may be written to as well as read
• Distinguishable by the number of times they can be reprogrammed (written to) and by the mechanism for reprogramming them
• Used for firmware programs (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems…), solid state disks, disk caches
Nonvolatile Memory (2)

- **Read-only memory (ROM)**
  - Programmed during production
- **Programmable ROM (PROM)**
  - Fuse associated with cell that is blown once by zapping with current
  - Can be programmed once
- **Eraseable PROM (EPROM)**
  - Cells cleared by shining ultraviolet light, special device used to write 1’s
  - Can be erased and reprogrammed about 1000 times
- **Electrically eraseable PROM (EEPROM)**
  - Similar to EPROM but does not require a physically separate programming device, can be re-programmed in place on printed circuit cards
  - Can be reprogrammed about 100,000 times
- **Flash Memory**
  - Based on EEPROM technology
  - Wears out after about 100,000 repeated writes
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 memory bus.

Load operation: `movl A, %eax`

Diagram: CPU (Register file: %eax) → ALU → Bus interface → I/O bridge → Main memory
Memory Read Transaction (2)

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

Load operation: `movl A, %eax`
Memory Read Transaction (3)

- CPU read word \( x \) from the bus and copies it into register \( \%eax \).
Memory Write Transaction (1)

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

Store operation: \texttt{movl} \%eax, \( A \)
Memory Write Transaction (2)

- CPU places data word \( y \) on the bus.

Store operation: `movl %eax, A`
Memory Write Transaction (3)

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

\[
\text{Store operation: } \text{movl } \%eax, \ A
\]
Disk Storage

• **Disks hold enormous amount of data** – on the order of hundreds to thousands of gigabytes compared to hundreds to thousands of megabytes in memory.

• **Disks are slower than RAM-based memory** – on the order of milliseconds to read information on a disk, a hundred thousand times longer than from DRAM and a million times longer than SRAM.
Anatomy of A Disk Drive

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

Image courtesy of Seagate Technology
Disk Geometry

- Disks consist of **platters**, each with two **surfaces**.
- Each surface consists of concentric rings called **tracks**.
- Each track consists of **sectors** separated by **gaps**.

![Diagram of disk geometry]
Disk Geometry (Multiple-Platter View)

- Aligned tracks form a cylinder.
Disk Capacity (1)

Capacity defined to be the maximum number of bits that can be recorded on a disk. Determined by the following factors:

- **Recording density (bits/in):** The number of bits on a 1-inch segment of a track.
- **Track density (tracks/in):** The number of tracks on a 1-inch segment of radius extending from the center of the platter.
- **Areal density (bits/in²):** Product of recording density and track density
Disk Capacity (2)

Determination of areal density:

• Original disks partitioned every track into the same number of sectors, which was determined by the innermost track. Resulted in sectors being spaced further apart on outer tracks.

• Modern disks partition into disjoint subsets called **recording zones**.
  • Each track within zone same number of sectors, determined by the innermost track.
  • Each zone has a different number of sectors/track.
Computing Disk Capacity

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

Example:
- 512 bytes/sector
- Average of 300 sectors/track
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

Capacity = 512 × 300 × 20,000 × 2 × 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 on a thin cushion of air.

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.
Disk Structure: Top View of Single Platter

Surface organized into tracks
Tracks divided into sectors
Disk Access (1)

Head in position above a track
Disk Access (2)

Rotation is counter-clockwise
Disk Access – Read (1.1)

About to read blue sector
Disk Access – Read (1.2)

After BLUE read

After reading blue sector
Disk Access – Read (1.3)

After BLUE read

Red request scheduled next
Disk Access – Seek

After BLUE read
Seek for RED

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 (2.1)

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
Calculating Access Time (1)

Average access time for a sector:

\[ T_{access} = T_{avg\_seek} + T_{avg\_rotation} + T_{avg\_transfer} \]

Seek time \((T_{avg\_seek})\):

- Time to position heads over cylinder
- Typical \(T_{avg\_seek}\) is 3–9 ms, max can be as high as 20 ms

Rotational latency \((T_{avg\_rotation})\):

- Once head is positioned over track, the time it takes for the first bit of the sector to pass under the head.
- In the worst case, the head just misses the sector and waits for the disk to make a full rotation.
  \[ T_{max\_rotation} = \left(\frac{1}{\text{RPM}}\right) \times \left(\frac{60\ \text{secs}}{1\ \text{min}}\right) \]
- Average case is \(\frac{1}{2}\) of worst case:
  \[ T_{avg\_rotation} = \left(\frac{1}{2}\right) \times \left(\frac{1}{\text{RPM}}\right) \times \left(\frac{60\ \text{secs}}{1\ \text{min}}\right) \]
- Typical \(T_{avg\_rotation} = 7200\ \text{RPMs.}\)
Calculating Access Time (2)

Transfer time ($T_{avg\_transfer}$):
- Time to read bits in the sector
- Time depends on the rotational speed and the number of sectors per track.
- Estimate of the average transfer time;
  - $T_{avg\_transfer} = \frac{1}{\text{RPM}} \times \frac{1}{(\text{avg \#sectors/tracks})} \times (60 \text{ secs/1 min})$

Example:
- Rotational rate = 7200 RPM
- Average seek time = 9 ms
- Avg \#sectors/track = 400

$T_{avg\_rotation} = \frac{1}{2} \times (60 \text{ secs}/7200 \text{ RPM}) \times (1000 \text{ ms/sec}) = 4 \text{ ms}$

$T_{avg\_transfer} = (60/7200 \text{ RPM}) \times (1/400 \text{ secs/track}) \times (1000 \text{ ms/sec}) = 0.02 \text{ ms}$

$T_{access} = 9 \text{ ms} + 4\text{ms} + 0.02 \text{ ms}$
Access Time

Time to access the 512 bytes in a disk sector is dominated by the seek time (9 ms) and rotational latency (4 ms).

Accessing the sector takes a long time but transferring bits are basically free.

Since seek time and rotational latency are roughly the same, at least same order of magnitude, doubling the seek time is a reasonable estimate for access time.

Comparison of access times of various storage devices when reading a comparable 512-byte sector sized block:

- SRAM: 256 ns
- DRAM: 5000 ns
- Disk: 10 ms
- Disk is about 40,000 times slower than SRAM, 2,500 times slower than DRAM.
Logical Disk Blocks

- Although modern disks have complex geometries they present a simpler abstract view as a sequence of \( B \) sector-sized logical blocks, numbered 0, 1, 2, … \( B-1 \).

- Disk controller maintains the mapping between the logical and actual (physical) disk sectors and converts requests for block into a surface, track and sector by doing a fast table lookup.

Formatted Disk Capacity

- Before disks can be used for the first time they must be formatted by the disk controller.

- Gaps between sectors filled in with info to identify sectors.

- Finds surface defects and sets aside cylinders to be used for spares.

- Formatted capacity is less than the maximum capacity.
Connecting I/O Devices

• I/O devices such as disks, graphics cards, monitors, mice, and keyboards are connected to the CPU and main memory using an I/O bus.

• Unlike the system bus and memory bus which are CPU specific, the I/O bus is independent of the underlying CPU.

• The I/O bus is slower than the system and memory buses but can accommodate a wide variety of third-party I/O devices. For instance, USB, graphics card or adapter, host bus adapter (SCSI/SATA).

• Network adapters can be connected to the I/O bus by plugging the adapter into an empty expansion slot on the motherboard.
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) associated with disk controller.
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* (i.e., asserts a special “interrupt” pin on the CPU).
Solid State Disks (SSDs)

- Pages: 512KB 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 100,000 repeated writes.
SSD Performance Characteristics

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential read throughput</td>
<td>250 MB/s</td>
<td>Sequential write throughput</td>
<td>170 MB/s</td>
</tr>
<tr>
<td>Random read throughput</td>
<td>140 MB/s</td>
<td>Random write throughput</td>
<td>14 MB/s</td>
</tr>
<tr>
<td>Random read access</td>
<td>30 μs</td>
<td>Random write access</td>
<td>300 μs</td>
</tr>
</tbody>
</table>

- Why are random writes so slow?
  - Erasing a block is slow (around 1 ms)
  - Write to a page triggers a copy of all useful pages in the block
    - Find an used block (new block) and erase it
    - Write the page into the new block
    - Copy other pages from old block to the new block
Advantages of SSD over Rotating Disks

• No moving parts – semiconductor memory – more rugged
• Much faster random access times
• Use less power

Disadvantages of SSD over Rotating Disks

• SSDs wear out with usage
• 100 times more expensive than disks
## Storage Technology 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>19,200</td>
<td>2,900</td>
<td>320</td>
<td>256</td>
<td>100</td>
<td>75</td>
<td>60</td>
<td>320</td>
</tr>
<tr>
<td>Access (ns)</td>
<td>300</td>
<td>150</td>
<td>35</td>
<td>15</td>
<td>3</td>
<td>2</td>
<td>1.5</td>
<td>200</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>8,000</td>
<td>880</td>
<td>100</td>
<td>30</td>
<td>1</td>
<td>0.1</td>
<td>0.06</td>
<td>130,000</td>
</tr>
<tr>
<td>Access (ns)</td>
<td>375</td>
<td>200</td>
<td>100</td>
<td>70</td>
<td>60</td>
<td>50</td>
<td>40</td>
<td>9</td>
</tr>
<tr>
<td>Size (MB)</td>
<td>0.065</td>
<td>0.256</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>2,000</td>
<td>8,000</td>
<td>125,000</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>$/MB</td>
<td>500</td>
<td>100</td>
<td>8</td>
<td>0.30</td>
<td>0.01</td>
<td>0.005</td>
<td>0.0003</td>
<td>1,600,000</td>
</tr>
<tr>
<td>Access (ms)</td>
<td>87</td>
<td>75</td>
<td>28</td>
<td>10</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>29</td>
</tr>
<tr>
<td>Size (MB)</td>
<td>1</td>
<td>10</td>
<td>160</td>
<td>1,000</td>
<td>20,000</td>
<td>160,000</td>
<td>1,500,000</td>
<td>1,500,000</td>
</tr>
</tbody>
</table>
# CPU Trends

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>8080</td>
<td>386</td>
<td>Pentium</td>
<td>P-III</td>
<td>P-4</td>
<td>Core 2</td>
<td>Core i7</td>
<td>—</td>
</tr>
<tr>
<td>Clock rate (MHz)</td>
<td>1</td>
<td>20</td>
<td>150</td>
<td>600</td>
<td>3300</td>
<td>2000</td>
<td>2500</td>
<td>2500</td>
</tr>
<tr>
<td>Cycle time (ns)</td>
<td>1000</td>
<td>50</td>
<td>6</td>
<td>1.6</td>
<td>0.3</td>
<td>0.50</td>
<td>0.4</td>
<td>2500</td>
</tr>
<tr>
<td>Cores</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Effective cycle time (ns)</td>
<td>1000</td>
<td>50</td>
<td>6</td>
<td>1.6</td>
<td>0.3</td>
<td>0.25</td>
<td>0.1</td>
<td>10,000</td>
</tr>
</tbody>
</table>

- Around 2003, system designers reached a limit regarding the exploitation of *instruction-level parallelism (ILP)* in sequential programs.
- Since 2000, processor speed has not significantly increased.
- Instead, we see *multicore processors*. 
The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.
Outline

• Storage Technologies
• Locality
• Memory Hierarchy
• Cache Memories
• Writing Cache-friendly Code
• Impact of Caches on Program Performance
**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.

- Principle of locality has an enormous impact on the design and performance of hardware and software systems. *In general, programs with good locality run faster than programs with poor locality.*

- This principle is used by all levels of a modern computer system from hardware, to operating systems, to application programs.
Locality Example

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

• Data references
  – Reference variable `sum` each iteration.

• Instruction references
  – Reference instructions in sequence.
  – Cycle through loop repeatedly.

Spatial locality

Temporal 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 of Reference to Program Data (2)

- 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;
}
```
Locality of Instruction Fetches

• Question: Does this function have good locality with respect to instructions?

```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;
}
```
Summary of Locality

Simple rules for evaluating the locality in a program:

• Programs that repeatedly reference the same variables enjoy good temporal locality.

• For programs with stride-\(k\) reference patterns, the smaller the stride the better for spatial locality. Programs with stride-1 reference patterns have good spatial locality. Programs that hop around memory with large strides have poor spatial locality.

• Loops have good temporal and spatial locality with respect to instruction fetches. The smaller the loop body and the greater the number of loop iterations, the better the locality.
Outline

• Storage Technologies
• Locality
• Memory Hierarchy
• Cache Memories
• Writing Cache-friendly Code
• Impact of Caches on Program Performance
The Memory Hierarchy

Fundamental properties of storage technology and computer software:

- **Storage technology:** Different storage technologies have widely different access times. Faster technologies cost more per byte than slower ones and have less capacity. The gap between CPU and main memory speed is widening.

- **Computer software:** Well-written programs tend to exhibit good locality.

The complementary nature of these properties suggest an approach for organizing memory systems, known as a **memory hierarchy**.
An Example Memory Hierarchy

- **L0:** Registers
  - CPU registers hold words retrieved from L1 cache

- **L1:** L1 cache (SRAM)
  - L1 cache holds cache lines retrieved from L2 cache

- **L2:** L2 cache (SRAM)
  - L2 cache holds cache lines retrieved from main memory

- **L3:** Main memory (DRAM)
  - Main memory holds disk blocks retrieved from local disks

- **L4:** Local secondary storage (local disks)
  - Local disks hold files retrieved from disks on remote network servers

- **L5:** Remote secondary storage (tapes, distributed file systems, Web servers)

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte
Caching in the Memory Hierarchy

• A cache is a small, fast storage device that acts as a staging area for the data objects stored in a larger, slower device.

• The central idea is that for each level $k$ in the memory hierarchy, the faster and larger storage device serves as a cache for the larger and slower storage devices at level $k+1$.

• If a program finds a needed data object from level $k+1$ in level $k$ then we have a cache hit. Otherwise we have a cache miss and the data must be brought to level $k$ from level $k+1$. 
General Cache Concepts

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

Data is copied in block-sized transfer units.

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

Data in block b is needed

Block b is in cache: Hit!
General Cache Concepts: Miss

Data in block $b$ is needed

Block $b$ is not in cache: Miss!

Block $b$ is fetched from memory

Block $b$ is stored in cache
- Placement policy: determines where $b$ goes
- Replacement policy: determines which block gets evicted (victim)
Kinds of Cache Misses

• **Cold miss:**
  
  – Cache at level $k$ is empty. Temporary situation that resolves itself when repeated accesses cause the cache to ‘warm up’

• **Conflict miss:**
  
  – Most caches limit the blocks at $k+1$ to a small subset (possibly only one) position at level $k$, for instance, block $i$ restricted to $(i \mod 4)$
  
  – Cache at level $k$ is large enough but needed blocks map to the same position, for instance, blocks 0, 4, 8, 12, 16, … mapping to 0 using $(i \mod 4)$

• **Capacity miss:**
  
  – Set of active blocks at $k+1$ larger than cache.
## Examples of Caching in the 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 byte 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</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-byte blocks</td>
<td>On-Chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-byte blocks</td>
<td>On/Off-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>4 KB page</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</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>AFS/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>
Outline

• Storage Technologies
• Locality
• Memory Hierarchy
• Cache Memories
• Writing Cache-friendly Code
• Impact of Caches on Program Performance
Cache Memories

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:
General Cache Organization (S, E, B)

- \( S = 2^s \) sets
- \( E = 2^e \) lines per set
- \( B = 2^b \) bytes per cache block (the data)

Cache size:
\[ C = S \times E \times B \] data bytes
Steps of a Cache Request

Given a request for the word $w$, the address for $w$ is used to determine the following;

1. **Set Selection**: Determine set within cache
2. **Line Matching**: Determine line within specific set
3. **Word Extraction**: Extract word from cache and return it to CPU
An ‘Aside’: Some Terminology

**Block:** fixed size packet of info that moves back and forth between a cache and main memory (or a lower-level cache)

**Line:** a container in a cache that stores a block as well as other info such as the valid bit and the tag bits

**Set:** collection of one of more lines. Sets in direct-mapped caches consist of a single line. Sets in set associative and fully associative caches consist of multiple lines.
Cache Read

- $E = 2^e$ lines per set
- $S = 2^s$ sets

- Locate set
- Check if any line in set has matching tag
- ‘Yes’ + line valid: hit
- Locate data starting at offset

Address of word:
- $t$ bits
- $s$ bits
- $b$ bits

- $m-1$ tag
- set index
- block offset

data begins at this offset

valid bit

$B = 2^b$ bytes per cache block (the data)
**Example: Direct Mapped Cache (E = 1)**

**Direct mapped:** One line per set  
Assume: cache block size 8 bytes

![Diagram of direct mapped cache with 8 bytes per line and 4 sets]

Address of int:  
\[
\begin{array}{c}
\text{t bits} \\
\hline
0...01 \\
100 \\
\end{array}
\]

find set
Example: Direct Mapped Cache (E = 1) (2)

Direct mapped: One line per set
Assume: cache block size 8 bytes

Address of int:

| t bits | 0...01 | 100 |

valid? + match: assume yes = hit

block offset
Example: Direct Mapped Cache (E = 1) (3)

Direct mapped: One line per set
Assume: cache block size 8 bytes

No match: old line is evicted and replaced
Direct-Mapped Cache Simulation

M=16 byte addresses, B=2 bytes/block, S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Block</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂],</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂],</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂],</td>
<td>miss</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>
A Higher Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];

    return sum;
}
```

Ignore the variables sum, i, j
assume: cold (empty) cache, a[0][0] goes here
E-way Set Associative Cache (Here: $E = 2$) (1)

$E = 2$: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

```
t bits | 0...01 | 100
```

```
find set
```
E-way Set Associative Cache (Here: E = 2) (2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

valid? + match: yes = hit

compare both

block offset

v tag 0 1 2 3 4 5 6 7
v tag 0 1 2 3 4 5 6 7
E-way Set Associative Cache (Here: E = 2) (3)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:
0...01 100

t bits

No match:
• One line in set is selected for eviction and replacement
• Replacement policies: random, least recently used (LRU), …
2-Way Set Associative Cache Simulation

M=16 byte addresses, B=2 bytes/block, S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>M[8-9]</td>
</tr>
<tr>
<td>8</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
</tbody>
</table>
A Higher Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];

    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here

32 B = 4 doubles

blackboard
How do we handle writes?

• **Hit**
  - *Write through:* write immediately to memory
  - *Write-back:* wait and write to memory when line is replaced (need a *dirty bit* to see if line is different from memory or not)
    - Nested loop structure

• **Miss**
  - *Write-allocate:* load into cache, update line in cache (good if more writes to the location follow)
  - *No-write-allocate:* writes immediately to memory

• **Typical**
  - Write-through + No-write-allocate
  - *Write-back* + *Write-allocate*
Intel Core i7 Cache Hierarchy

Processor package

Core 0
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)

Core 3
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache

L1 i-cache and d-cache: 32 KB, 8-way, Access: 4 cycles
L2 unified cache: 256 KB, 8-way, Access: 11 cycles
L3 unified cache: 8 MB, 16-way, Access: 30-40 cycles

Block size: 64 bytes for all caches.

Main memory
Performance Metrics

Cache performance is evaluated with a number of metrics;

• **Miss Rate:**
  – The fraction of memory references during the execution of a program, or a part of a program, that miss ($\#\text{misses}/\#\text{references}) = 1 – \text{hit rate}.
  – Typical numbers (in %s): 3-10% for L1, can be quite small (<1%) for L2, depending on size, etc.

• **Hit Rate:**
  – The fraction of memory references that hit. = 1 – miss rate

• **Hit Time:**
  – The time to deliver a word in the cache to the CPU, including time for set selection, line identification, and word selection.
  – Hit time is on the order of several clock cycles for L1, 5-20 for L2.

• **Miss Penalty:**
  – Any additional time required because of a miss.
  – Penalty for L1 served from L2 is on the order of 10 cycles; from L3, 40 cycles; and from main memory, 100 cycles.
Some Insights… (1)

99% Hits is Twice as Good as 97%:
– Consider: each hit time of 1 cycle and miss penalty of 100 cycles
– Average access time:
  – 97% hits: 1 cycle +0.03*100 cycles = 4 cycles
  – 99% hits: 1 cycle+0.01* 100 cycles = 2 cycles.

Impact of Cache Size:
– On the one hand, a larger cache will tend to increase the hit rate. On the other hand, it is always harder to make larger memories run faster. As a result, larger caches tend to increase the hit time.

Impact of Block Size:
– Larger blocks can help increase hit rate by exploiting spatial locality however larger blocks imply a smaller number of cache lines which hurt hit rate with more temporal locality than spatial. Usually have blocks of 32-64 bytes.
Some Insights… (2)

Impact of Associativity:

• Higher associativity (larger values of E) decrease the vulnerability of the cache to thrashing due to conflict misses.

• Higher associativity expensive to implement and hard to make fast. It requires more tag bits per line, additional LRU state bits per line, and additional control logic. It can increase hit time and increase miss penalty because of increased complexity.

• Choice usually trade-off between hit time and miss penalty. Intel Core i7 systems: L1 and L2 are 8-way, L3 is 16-way.
Outline

- Storage Technologies
- Locality
- Memory Hierarchy
- Cache Memories
- **Writing Cache-friendly Code**
- Impact of Caches on Program Performance
Writing Cache-Friendly Code

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

• Good programmers should always try to write code that is cache friendly, in the sense that it has good locality.
Approach to Cache Friendly Code

• *Make the common case go fast.* Programs often spend most of their time in a few core functions. These functions often spend most of their time in a few loops. So focus on the inner loops of the core function and ignore the rest.

• *Minimize the number of cache misses for each inner loop.* Good programmers should always try to write code that is cache friendly, in the sense that it has good locality.
Outline

• Storage Technologies
• Locality
• Memory Hierarchy
• Cache Memories
• Writing Cache-friendly Code
• Putting it Together: Impact of Caches on Program Performance
Putting it Together: Impact of Caches on Program Performance

- The Memory Mountain
- Rearranging loops to improve spatial locality
- Using blocking to improve temporal locality
The Memory Mountain

Every computer has a unique memory mountain that characterizes the capabilities of its memory.

Read throughput (read bandwidth):

- Number of bytes read from memory per second (MB/s)

Memory Mountain: Measured read throughput as a function of spatial and temporal locality.

- Compact way to characterize memory system performance.
Memory Mountain Test Function

```c
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);    /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0); /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```
The Memory Mountain (1)

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip
The Memory Mountain (3)

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip

Read throughput (MB/s)

Stride (x8 bytes)

Working set size (bytes)

Slopes of spatial locality

Ridges of Temporal locality

L1

L2

L3

Mem
Memory Mountain Summary

The performance of the memory mountain is not characterized by a single number. Instead, it is a mountain of temporal and spatial locality whose elevations can vary by over an order of magnitude.

Wise programmers try to structure their programs so that they run in the peaks instead in the valleys.

The aim is to exploit temporal locality so that heavily used words are fetched from the L1 cache, and to exploit spatial locality so that as many words as possible are accessed from a single L1 cache line.
Programming Example: Matrix Multiplication

Consider the problem of multiplying a pair of $N \times N$ matrices: $C = AB$.

A matrix multiplying function is usually implemented using three nested loops, which are identified with indexes $i, j,$ and $k$.

If we permute the loops and make some minor code changes, we can create six functionally equivalent versions. Each version is uniquely identified by the ordering of its loops.
Miss Rate Analysis for Matrix Multiplication

- **Assume:**
  - Line size = 32B (big enough for four 64-bit words)
  - Matrix dimension \((N)\) is very large
    - Approximate \(1/N\) as 0.0
  - Cache is not even big enough to hold multiple rows

- **Analysis Method:**
  - Look at access pattern of inner loop
Matrix Multiplication Example

- **Description:**
  - Multiply $N \times N$ matrices
  - $O(N^3)$ 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
Layout of C Arrays in Memory (review)

- **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) > 4 bytes, exploit spatial locality
    - Compulsory miss rate = 4 bytes / B
- **Stepping through rows in one column:**
  - for (i = 0; i < n; i++)
    
    ```
    sum += a[i][0];
    ```
  - Accesses distant elements
  - No spatial locality!
    - Compulsory miss rate = 1 (i.e. 100%)
Matrix Multiplication ($ijk$)

```c
/* $ijk$ */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        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
    }
}
```

Inner loop:

```
\[ \text{Row-wise} \quad \text{Column-wise} \quad \text{Fixed} \]
```

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>0.0</td>
<td>0.25</td>
<td>0.25</td>
<td></td>
</tr>
</tbody>
</table>

Inner loop:

- Fixed
- Row-wise
- Row-wise

Diagram:

- $(i,k)$
- $(k,*)$
- $(i,*)$
Matrix Multiplication (ikj)

```c
/* 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:

- **A**: Fixed
- **B**: Row-wise
- **C**: Row-wise

| Misses per inner loop iteration: |
|---|---|---|
| **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

B

C

Column-wise

Fixed

Column-wise

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>
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;
  }
}
```

**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>
Summary of Matrix Multiplication

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 (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];
  }
}

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;
}
Core i7 Matrix Multiply Performance

Cycles per inner loop iteration vs. Array size (n)

- jki / kji
- ijk / jik
- kij / ikj
Example: Matrix Multiplication

c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n+k]*b[k*n + j];
}
Cache Miss Analysis (1)

• Assume:
  – Matrix elements are doubles
  – Cache block = 8 doubles
  – Cache size C << n (much smaller than n)

• First iteration:
  – $\frac{n}{8} + n = \frac{9n}{8}$ misses

  – Afterwards *in cache*: (schematic)
Cache Miss Analysis (2)

• Assume:
  – Matrix elements are doubles
  – Cache block = 8 doubles
  – Cache size $C \ll n$ (much smaller than $n$)

• Second iteration:
  – Again:
    $\frac{n}{8} + n = \frac{9n}{8}$ misses

• Total misses:
  – $\frac{9n}{8} \times n^2 = \left(\frac{9}{8}\right) \times n^3$
Blocked Matrix Multiplication

c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
Cache Miss Analysis (1)

• Assume:
  – Cache block = 8 doubles
  – Cache size $C \ll n$ (much smaller than $n$)
  – Three blocks fit into cache: $3B^2 < C$

• First (block) iteration:
  – $B^2/8$ misses for each block
  – $2n/B \times B^2/8 = nB/4$
    (omitting matrix $C$)

  – Afterwards in cache (schematic)
Cache Miss Analysis (2)

• Assume:
  – Cache block = 8 doubles
  – Cache size $C << n$ (much smaller than $n$)
  – Three blocks fit into cache: $3B^2 < C$

• Second (block) iteration:
  – Same as first iteration
  – $2n/B \times B^2/8 = nB/4$

• Total misses:
  – $nB/4 \times (n/B)^2 = n^3/(4B)$
Summary

• No blocking: \( (9/8) \ast n^3 \)
• Blocking: \( 1/(4B) \ast n^3 \)

• Suggest largest possible block size B, but limit \( 3B^2 < C! \)

• Reason for dramatic difference:
  – Matrix multiplication has inherent temporal locality:
    • Input data: \( 3n^2 \), computation \( 2n^3 \)
    • Every array elements used \( O(n) \) times!
  – But program has to be written properly
Concluding Observations

- **Programmer can optimize for cache performance**
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique

- **All systems favor “cache friendly code”**
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)