CS 31: Intro to Systems Storage and Memory

Martin Gagne
Swarthmore College
March 16, 2016
Transition

• First half of course: hardware focus
  – How the hardware is constructed
  – How the hardware works
  – How to interact with hardware / ISA

• Up next: performance and software systems
  – Memory performance
  – Operating systems
  – Standard libraries (strings, threads, etc.)
Efficiency

• How to Efficiently Run Programs

• Good algorithm is critical (so you should take CS35 and CS41)

• Hardware matters

• Many systems concerns to account for too!
  – The Memory Hierarchy and its effect on program performance
  – OS abstractions for running programs efficiently
  – Support for Parallel programming
Efficiency

• How to Efficiently Run Programs

• Good algorithm is critical (so you should take CS35 and CS41)

• Hardware matters

• Many systems concerns to account for too!
  – The Memory Hierarchy and its effect on program performance
  – OS abstractions for running programs efficiently
  – Support for Parallel programming
Suppose you’re designing a new computer architecture. Which type of memory would you use? **Why?**

A. low-capacity (~1 MB), fast, expensive

B. medium-capacity (a few GB), medium-speed, moderate cost

C. high-capacity (100’s of GB), slow, cheap

D. something else (it must exist)
Classifying Memory

• Broadly, two types of memory:
  1. Random access: CPU instructions can access any location at any time (assuming OS permission)
  2. Secondary storage: CPU can’t access this directly
Memory Technologies

Volatile
(loses data without power)

Latches
(SRAM: registers, cache)

Capacitors
(DRAM)

Non-Volatile
(maintains data when computer is turned off)

Magnetic
(hard drives)

Flash
(SSDs)

$$$

$$$

$$$
Memory Technologies

• Static RAM (SRAM)
  – 0.25ns – 2ns, about $500 per GB

• Dynamic RAM (DRAM)
  – 20ns – 50ns, $5 - $8 per GB
    (Main memory, “RAM”)

We’ve talked a lot about registers (SRAM) and we’ll cover caches (SRAM) soon. Let’s look at main memory (DRAM) now.
Dynamic Random Access Memory (DRAM)

Capacitor based:
- cheaper and slower than SRAM
- capacitors are leaky (lose charge over time)
- **Dynamic**: value needs to be refreshed (every 10-100ms)

Example: DIMM (Dual In-line Memory Module):
Connecting CPU and Memory

- Components are connected by a **bus**:
  - A bus is a collection of parallel wires that carry address, data, and control signals.
  - Buses are typically shared by multiple devices.
How Memory Read Works

(1) CPU places address A on the memory bus.

Load operation: \texttt{movl (A), %eax}
Read (cont.)

(2) Main Memory reads Address A from Memory Bus, fetches x at that address and puts it on the bus

Load operation: \texttt{movl (A), %eax}
Read (cont.)

(3) CPU reads \( X \) from the bus, and copies it into register \( \%eax \), a copy also goes into the on-chip cache memory

Load operation: `movl (A), \%eax`
Write
1. CPU writes A to bus, Memory Reads it
2. CPU writes y to bus, Memory Reads it
3. Memory stores read value, y, at address A

Store operation: movl %eax, (A)
Secondary Storage

• Disk, Tape Drives, Flash Solid State Drives, …

• Non-volatile: retains data without a charge

• Instructions **CANNOT** directly access data on secondary storage
  – No way to specify a disk location in an instruction
  – Operating System moves data to/from memory
I/O Bus: connects Devices & Memory

OS moves data between Main Memory & Devices

Expansion slots for other devices such as network controller.

CPU chip

Register file

ALU

Cache

Bus interface

System bus

Memory bus

Main memory

I/O bridge

I/O bus

USB controller

Graphics controller

Disk controller

Mouse

Keyboard

Monitor

Disk
Device Driver: OS device-specific code

OS driver code running on CPU makes read & write requests to Device Controller via I/O Bridge
What’s Inside A Disk Drive?

- **Spindle**
- **Arm**
- **Actuator**
- **Platters**
- **Controller Electronics** (includes processor & memory)
- **R/W head**
- **bus connector**

Data Encoded as points of magnetism on Platter surfaces

Device Driver (part of OS code) interacts with Controller to R/W to disk

*Image from Seagate Technology*
Reading and Writing to Disk

Data blocks located in some **Sector** of some **Track** on some **Surface**

1. Disk Arm moves to correct **track** (seek time)
2. Wait for **sector** spins under R/W head (rotational latency)
3. As sector spins under head, data are Read or Written (transfer time)

![Diagram of disk arm and sector](image)

- Disk arm sweeps across surface to position read/write head over a specific track.
- Disk surface spins at a fixed rotational rate
  - ~7200 rotations/min
What about SSDs?

- They are just big USB keys (with better connection)
- They have no moving parts

*Images from Wikipedia*
Memory Technology

• Static RAM (SRAM)
  – 0.25ns – 2ns, about $500 per GB

• Dynamic RAM (DRAM)
  – 20ns – 50ns, $5 – $8 per GB

• Solid-State Disks
  – 100us – 1ms, $0.25 – $0.40 per GB

• Magnetic disk
  – 5ms – 20ms, $0.02 – $0.05 per GB

  $1 ms = 1,000,000 ns
Where does accessing the network belong?

- **A:** Here
- **B:** Here
- **C:** Somewhere else

The diagram shows a hierarchy of memory access times:

- **Registers:** 1 cycle to access
- **Cache(s) (SRAM):** ~10’s of cycles to access
- **Main memory (DRAM):** ~100 cycles to access
- **Local secondary storage (disk):** ~100 M cycles to access

The diagram also indicates:

- **Smaller:** Faster
- **Larger:** Slower
- **Faster:** Costlier per byte
- **Slower:** Cheaper per byte

The diagram includes arrows indicating that CPU instructions can directly access registers and cache(s) on chip storage.
The Memory Hierarchy

- Smaller, Faster, Costlier per byte
  - Registers: 1 cycle to access
  - CPU instructions can directly access
- Larger, Slower, Cheaper per byte
  - Cache(s) (SRAM): ~10’s of cycles to access
  - Main memory (DRAM): ~100 cycles to access
  - Flash SSD / Local network: ~100 M cycles to access
  - Local secondary storage (disk): slower than local disk to access
  - Remote secondary storage (tapes, Web servers / Internet): slower than local disk to access
Abstraction Goal

• Reality: There is no one type of memory to rule them all!

• Abstraction: hide the complex/undesirable details of reality.

• Illusion: We have the speed of SRAM, with the capacity of disk, at reasonable cost.
Motivating Story / Analogy

• You work for Netflix’s DVD rental division

• You have a huge warehouse of movies
  – 2-3 days to deliver movie customer
  – Customers don’t like waiting…

• You have many small warehouses of frequent requests
  – delivered in 1 day
Problem: Prediction

• We can’t know the future…

• So… are we out of luck?
  What might we look at to help us decide?

• The past is often a pretty good predictor…
Analogy: two types of Netflix users

1:

2:

What should be next in each user’s queue?
Critical Concept: Locality

• Locality: we tend to repeatedly access recently accessed items, or those that are nearby.

• Temporal locality: An item accessed recently is likely to be accessed again soon. (User 1)

• Spatial locality: We’re likely to access an item that’s nearby others we just accessed. (User 2)
In the following code, how many examples are there of temporal / spatial locality?

Where are they?

```c
int i;
int num = read_int_from_user();
int *array = create_random_array(num);
for (i = 0; i < num; i++) {
    printf("At index %d, value: %d", i, array[i]);
}
```

A. 1 temporal, 1 spatial
B. 1 temporal, 2 spatial
C. 2 temporal, 1 spatial
D. 2 temporal, 2 spatial
E. Some other number
Example

Temporal Locality?

array, num and i used over and over again in each iteration

Spatial Locality?

array bucket access
program instructions

Programs with loops tend to have a lot of locality
and most programs have loops:
it’s hard to write a long-running program w/o a loop
Use Locality to Speed-up Memory Access

**Caching Key idea:** keep copy of “likely to be accessed soon” data in higher levels of Memory Hierarchy to make their future accesses faster:

- recently accessed data (temporal locality)
- data nearby recently accessed data (spatial locality)

If program has high degree of locality, next data access is likely to be in cache

- if little/no locality, then caching won’t help
+ luckily most programs have a high degree of locality
Big Picture

For memory exhibiting locality (stuff we’re using / likely to use):

Work hard to keep them up here!

Bulk storage down here.

Move this up on demand.
Big Picture

Registers
Holds a VERY small amount.

Cache(s) (SRAM)
Faster than cache. (On-chip hardware)
Holds a subset of memory.

Main memory (DRAM)
Faster than disk.
Holds a subset of disk.

Flash SSD / Local network

Local secondary storage (disk)

Remote secondary storage (tapes, Web servers / Internet)
Cache

• In general: a storage location that holds a subset of a larger memory, faster to access

• CPU cache: an SRAM on-chip storage location that holds a subset of DRAM main memory (10-50x faster to access)

• Goal: choose the right subset, based on past locality, to achieve our abstraction

When I say “cache”, assume this for now.
Cache Basics

- CPU real estate dedicated to cache
- Usually two levels on CPU:
  - L1: smallest, fastest
  - L2: larger, slower
- Same rules apply:
  - L1 subset of L2
- Now L3 shared between cores
Pretty big chunk of the space is dedicated to the cache

Each core has its own L1 and L2 cache
Cache Basics

- CPU real estate dedicated to cache
- Usually two levels:
  - L1: smallest, fastest
  - L2: larger, slower
- We’ll assume one cache (same principles)

Cache is a subset of main memory.
(Not to scale, memory much bigger!)
Cache Basics: Read from memory

- In parallel:
  - Issue read to memory
  - Check cache
Cache Basics: Read from memory

- In parallel:
  - Issue read to memory
  - Check cache

- Data in cache (hit):
  - Good, send to register
  - Cancel/ignore memory
Cache Basics: Read from memory

- In parallel:
  - Issue read to memory
  - Check cache

- Data in cache (hit):
  - Good, send to register
  - Cancel/ignore memory

- Data not in cache (miss):
  1. Load cache from memory (might need to evict data)
  2. Send to register
Cache Basics: Write to memory

- Assume data already cached
  - Otherwise, bring it in like read

1. Update cached copy.

2. Update memory?
When should we copy the written data from cache to memory? **Why?**

A. Immediately update the data in memory when we update the cache.

B. Update the data in memory when we evict the data from the cache.

C. Update the data in memory if the data is needed elsewhere (e.g., another core).

D. Update the data in memory at some other time. *(When?)*
When should we copy the written data from cache to memory?  **Why?**

A. Immediately update the data in memory when we update the cache. (“Write-through”)

B. Update the data in memory when we evict the data from the cache. (“Write-back”)

C. Update the data in memory if the data is needed elsewhere (e.g., another core).

D. Update the data in memory at some other time. (When?)
Cache Basics: Write to memory

• Both options (write-through, write-back) viable

• write-through: write to memory immediately
  – simpler, accesses memory more often (slower)

• write-back: only write to memory on eviction
  – complex (cache inconsistent with memory)
  – potentially reduces memory accesses (faster)
Cache Basics: Write to memory

• Both options (write-through, write-back) viable

• write-through: write to memory immediately
  – simpler, accesses memory more often (slower)

• write-back: only write to memory on eviction
  – complex (cache inconsistent with memory)
  – potentially reduces memory accesses (faster)

Sells better.
Servers/Desktops/Laptops
Cache Coherence

- Keeping multiple cores’ memory consistent
Cache Coherence

- Keeping multiple cores’ memory consistent
  - Copy data directly from one cache to the other.
  - Avoid (slower) memory
- Lots of HW complexity here. We might discuss towards end of semester.
Up next:

• Cache details

• How cache is organized
  – finding data
  – storing data

• How cached subset is chosen (eviction)