DSC204A: Scaleable Data Systems Winter 2025

Lecture 5: April 9 2025

Instructor: Umesh Bellur *Scribe: Guruprasad Parasnis, Sarah Borsotto*

Five Stages of an Execution Pipeline

- **Fetch**: Instructions are fetched from an instruction memory.
- **Decode**: The instructions are decoded.
- **Execute**: Determines which registers to operate on based on the instruction.
- **Memory Operation**: If needed, interacts with memory.
- **Clocking**: All these operations are clocked, giving us the concept of clock speed.

Clock speeds have increased dramatically in accordance with Moore's law, from early 5 MHz processors to modern CPUs running at nearly 4 GHz. However, this results in significant heat generation. Data centers use oil-cooled CPUs to manage heat. With limits on clock speed due to thermal issues, the trend has moved to multi-core CPUs—like NVIDIA's 24,000-core chips. Distributed systems now also handle thermal efficiency more effectively.

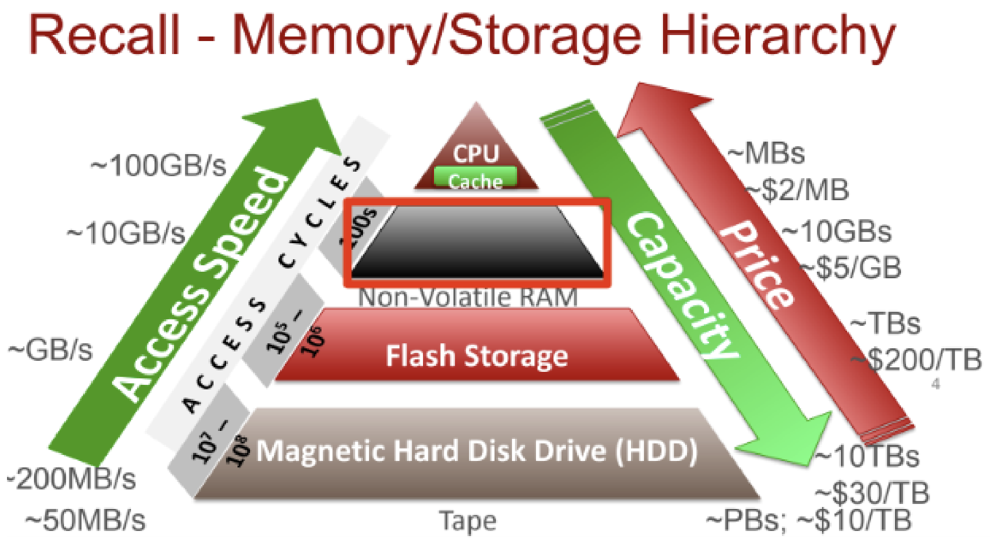


Figure 1: Memory Heirarchy Diagram

Memory and Storage Hierarchy

When designing computer systems, we face a tradeoff between speed, capacity, and cost. The solution is a hierarchy of storage technologies, as seen in Figure 1:

Evaluation Metrics

- **Access Speed**: How fast data can be read/written (GB/s or MB/s)
- **Capacity**: Total amount of storable data (MB/GB/TB/PB)
- **Price**: Cost per unit of storage ($/GBor/$TB)

Hierarchy Levels

1. **CPU Cache (Fastest, Smallest, Most Expensive)**
   - Speed: 100 GB/s
   - Latency: 10 CPU cycles
   - Capacity: Few MBs
   - Cost: $2 per MB
   - Usage: Holds data immediately needed by CPU
2. **Non-Volatile RAM (Fast, Moderate Capacity)**
   - Speed: 10 GB/s
   - Latency: 100s of cycles (10^2 to 10^3)
   - Capacity: Tens of GBs
   - Cost: $5 per GB
   - Usage: Stores critical data that must persist across power cycles
3. **Flash Storage (SSD)**
   - Speed: 1 GB/s
   - Latency: 10^3 to 10^4 cycles
   - Capacity: TBs
   - Cost: $200 per TB
   - Usage: OS storage, application data, commonly used in devices
4. **Magnetic Hard Disk Drive (HDD)**
   - Speed: 50–200 MB/s
   - Latency: 10^5–10^6 cycles
   - Capacity: Tens of TBs
   - Cost: $30 per TB
   - Usage: Archives, backups, media libraries

Why this Hierarchy Matters:

- Combines fast memory for performance with large storage for capacity
- Engineers design systems that balance cost and efficiency
- Modern machines use multiple layers—cache, RAM, SSDs, HDDs

Precaching and Locality of Reference

Precaching or prefetching refers to the strategy of loading data into a fast-access cache before it is explicitly requested by the CPU. The goal is straightforward: to predict what the CPU will likely need and fetch it ahead of time, thereby reducing the latency and significantly increasing performance. It's an approach that complements the memory/storage hierarchy previously discussed.

Why Precaching is Necessary

- Cache is fast but small
- Slower memories add latency
- Waiting for data stalls CPU
- Precaching minimizes idle cycles and maximizes throughput

How Precaching Works

Precaching is based on predictions. Effective prediction boosts performance, while incorrect predictions can waste bandwidth and pollute caches. Therefore, precaching helps to ensure the CPU is always provided with data at maximum efficiency, avoiding idle cycles and enhancing the throughput of instructions. Precaching inherently relies on prediction. The more accurately a CPU can predict its future data requirements, the more effective precaching becomes. Correct predictions lead to significant performance improvements, while incorrect predictions can lead to wasted memory bandwidth and cache pollution.

Prediction Principles

- **Spatial Locality**: If a memory location is accessed, nearby memory locations are likely to be accessed soon. For example in computer programs, when we run a loop, a set of instructions next to one another are called ; viz. When fetching the first instruction, we then fetch the the next x instructions (prefetching)
- **Temporal Locality**: Recently accessed data is likely to be accessed again soon. It is likely that the CPU will ask for contents of the same location again soon. In a program, when we run a loop, the same set of instructions are run repeatedly

This principle is what is known as **Locality of Reference**. It essentially tends to keep the CPU occupied at all times. For example, just like airplanes are diverted to different locations and made to turn around instead of just sitting idle. The principle provides the base for optimizing access to main memory using spatial and temporal locality discussed above.

Where does temporal locality in programs exist?

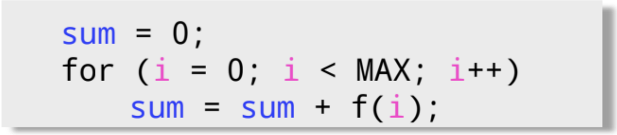
Loops are a prime example of temporal locality in programs. Since instructions inside a loop are reused frequently, computer systems can optimize performance by keeping the instructions in cache.

Why Loops Show Temporal Locality:

- A loop is a block of code that repeats multiple times.
- When a loop runs, the same set of instructions is executed during each iteration.
- The CPU fetches the same instructions again and again, so keeping the same instruction in cache allows for a more efficient program.

What is temporal locality in data?

Temporal locality also applies to data. If a variable is accessed once, it's likely to be accessed again soon, especially inside a loop.



```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + f(i);
```

Figure 2: Temporal Locality in Data

- The variable sum is accessed repeatedly in the loop.
- Every iteration reads and writes to sum.
- Because it's used again and again in a short time, it shows temporal locality.

Registers and Performance

To improve speed, the CPU tries to keep frequently accessed variables like sum in registers. Registers are small, high-speed memory locations inside the CPU. It acts as a temporary storage area for data and instructions that the CPU may need to access frequently during program execution. Registers are significantly faster than accessing memory locations in RAM, making them crucial for the CPU's performance.

Storing sum in a register avoids repeatedly accessing slower main memory (RAM). The value is updated in the register and only written back to memory after the loop ends.

Where does spatial locality in programs exist?

Most programs execute instructions in sequence, like accessing memory location i, then i+1, i+2, etc. So the CPU tries to fetch blocks of nearby instructions in advance, anticipating what will be needed next. These blocks are stored in the instruction cache to avoid delays in future fetches. Loops are a good example of spatial locality, just like with temporal locality:

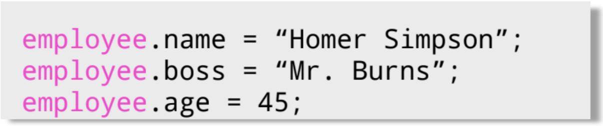
- In a loop, the set of instructions is often stored next to each other in memory.
- CPUs typically fetch multiple instructions at once (e.g., 10 at a time) instead of one-by-one.
- These are stored in the instruction cache, ready for the CPU to execute.

What happens when there's a decision point, like an if-else statement?

The CPU tries to predict which path will be taken (called branch prediction). If it guesses wrong, it ends up fetching the wrong instructions first. Then, it has to discard them and fetch the correct ones, leading to a misprediction.

What is spatial locality in data?

If one data item is accessed, nearby data items will likely be accessed soon too. Since all the fields are stored contiguously, the CPU can access them more efficiently if they're kept together in memory or in cache.



```
employee.name = “Homer Simpson”;
employee.boss = “Mr. Burns”;
employee.age = 45;
```

Figure 3: Spatial Locality in Data

Let's say you're working with an employee record in a database. These fields are typically stored next to each other in memory (in a structure or row). If a program accesses the name, there's a high chance it will also access age, boss, etc. Instead of fetching just the name, the system might fetch the whole record at once.

Another example is a matrix in memory:

In most programming languages (like C, Python NumPy), matrices are stored in row-major order. This means the elements of each row are stored next to each other. So if a program accesses matrix[0][0], it likely will access matrix[0][1], matrix[0][2], and so on, all of which are physically near each other in memory.

Keeping related data close allows the CPU to prefetch or cache it. When the CPU first reads from memory, it also stores a copy of the data in cache. This first read is slower, but the idea is to avoid future slow reads. Therefore, spatial locality improves efficiency by minimizing memory lookups.

Impact of caching and average access time

Caching significantly improves performance by reducing access time:

Cache access: nanoseconds (ns)

Main memory (RAM) access: microseconds (µs)

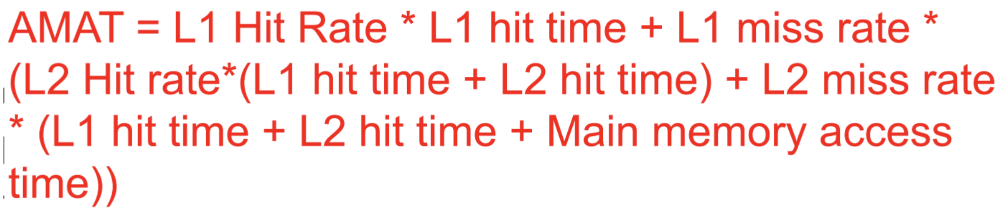
Disk access: milliseconds (ms)

The closer to the CPU, the faster the access.

Some key terms:

- Cache Hit: Data is found in the cache → fast access
- Cache Miss: Data is not in cache → fetch from slower memory
- Cache Hit Rate (h): Probability of a cache hit
- Cache Miss Rate: $1 - h$
- Cache Eviction: If the cache is full, one item must be evicted to make space.

When the CPU needs data, it first checks the cache. If the data is found (a cache hit), it's used immediately, fast and efficient. If the data is not in the cache (a cache miss), the CPU must fetch it from main memory, store a copy in the cache, and then use it. So what's the average access time? Well, it depends on how often we hit or miss in the cache, which is where we calculate the average memory access time (AMAT) using hit rate and access times.



AMAT = L1 Hit Rate * L1 hit time + L1 miss rate * (L2 Hit rate*(L1 hit time + L2 hit time) + L2 miss rate * (L1 hit time + L2 hit time + Main memory access time))

Figure 4: AMAT Calculation

The idea is to calculate the average access time by multiplying the hit rate by the cache access time, then adding the miss rate (1 - hit rate) multiplied by the total time for a miss, which includes both the cache access and the memory access, since the cache is always checked first.

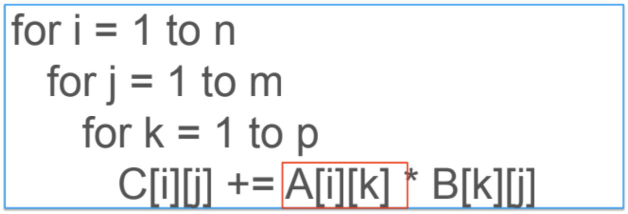
Locality of Reference in Data Science

To make programs run faster, it's important to write them in a way that takes advantage of locality, especially spatial locality.

Matrix multiplication is a classic case where access patterns matter a lot.

In most systems, arrays in memory (DRAM) are stored in row-major order. That means the elements of each row are placed next to each other in memory.

Example memory layout: $A[0][0], A[0][1], A[0][2], ...A[1][0], A[1][1], ...$ and the the Standard 3-Level Loop...



```
for i = 1 to n
   for j = 1 to m
      for k = 1 to p
         C[i][j] += A[i][k] * B[k][j]
```

Figure 5: Three level loop for Matrix Multiplication

Accessing $A[i][k]$ uses spatial locality (elements of a row), so it's cache-friendly. But $B[k][j]$ accesses elements down a column, which are not stored contiguously in memory → causes cache misses. Every time we fetch from $B[k][j]$, it's likely a cache miss, stalling the CPU and slowing everything down.

What is the fix? Switch the j and k loops! The program now reads A[i][k] as before but now accesses B[k][j] in a row-wise fashion too, making it cache-friendly.

Disk

GPT-3 has 175 billion parameters. If stored using bf16 (16 bits = 2 bytes): 175 billion × 2 bytes = 350 GB. These types of models are too large to fully fit in RAM, especially during training or when storing multiple versions. That is why they should be stored on disks, either Solid State Drives (SSDs) or Hard Disk Drives (HDDs).

A disk is made of platters with concentric tracks. Each track is split into sectors (typically 512 bytes). A magnetic head on a movable arm reads/writes data. RPM (Revolutions Per Minute) determines how fast the disk spins. The disk receives a read request using a block address. It moves the arm to the correct track (seek time). It waits for the correct sector to spin under the head (rotational latency). Data is read and sent back. This process causes latency, especially compared to SSDs which have no moving parts.