**DSC204A: Scaleable Data Systems Winter 2025**

# Lecture 6: Disks and Pipelining

*Instructor: Umesh Bellur* *Scribe: Shweta Nalluri,Keertana Kappuram*

## 1 Recap

In the previous lecture, caching and memory were discussed.

As we saw in the Memory Hierarchy pyramid, caching is closest to the CPU. "Memory gap" refers to the significant difference between the CPU's processing speed and the speed at which it can access data from the main memory. Caching is used to fix this, by keeping data that needs to be operated on for processing, close to the CPU. In addition, temporal and spatial locality enable the cache to work.

In the memory hierarchy pyramid, we also observe a trade-off between access speed and storage capacity as we move farther from the CPU. At the top of the hierarchy, closest to the CPU, are the L1, L2, and L3 caches, which are embedded directly on the CPU chip and offer extremely fast access times, typically around 1 nanosecond. As we move down the hierarchy, there are other forms of memory with slightly slower access times in the range of hundreds of nanoseconds to microseconds. For larger storage needs, systems rely on solid-state drives (SSDs), which behave like memory but are significantly slower than RAM. At the bottom of the pyramid are spinning hard disk drives (HDDs), which offer massive storage capacities ranging from 4 to 16 terabytes or more. These can even be connected in arrays to store even larger volumes of data, but with much slower access times compared to memory closer to the CPU.

## 2 Disks

A disk's structure includes a platter. Each platter has a surface. Next, every surface has tracks, which are concentric circles and these tracks have sectors which are magnetically readable or writable entities. We saw an example where we read from two different sectors on the disk.

In figure 1, we can see that the two sectors, marked red and blue, are located on different tracks of the disk, with the blue sector on an inner track and the red sector on an outer track. Initially, the head is positioned over the blue sector and reads data from it. To read from the red sector, the head must first move to the correct track by retracting outward from the inner to the outer track, a movement performed mechanically by a stepper motor. However, this movement alone is not sufficient. Since the disk platter is continuously spinning, the head must also wait until the red sector rotates underneath it. Only then can the head begin reading from the red sector. This process involves three key steps:

1. The head moves to the correct track
2. The system waits for the platter to rotate the red sector beneath the head
3. The data is read as the platter continues to spin.

Importantly, the actual speed of reading data is significantly slower than both the head movement and the rotational latency of the spinning platter, which is itself limited by how fast the platter can physically rotate.

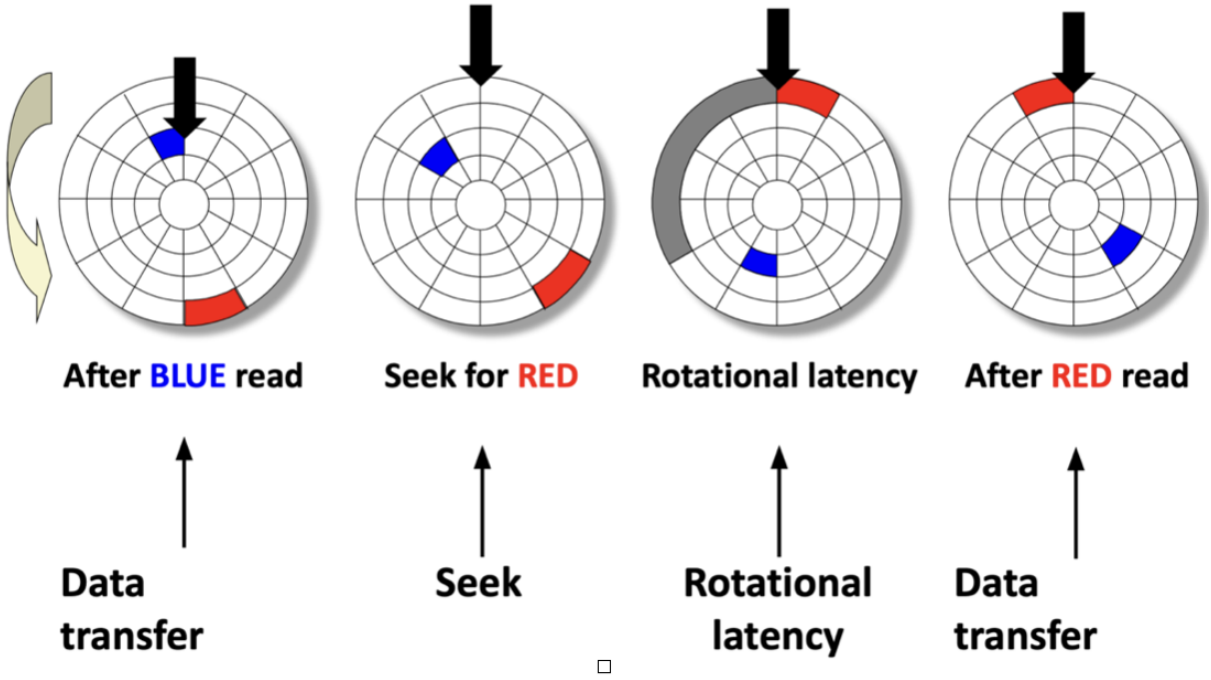


Figure 1: Disk Head Movement and Rotational Latency Visualization

## 3 Components of latency and Disk Access Time

The Disk Access time is defined as the average time taken to access some target sector. The disk access time is calculated as follows:

$$T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$$

The 3 main important components of latency are:

1. **Seek Time**: The seek time, typically ranging from 3-9ms, is defined as the time taken to position the head over the cylinder containing the target sector. It is usually given to you according to the disk you have.
2. **Rotational time/Rotational latency**: Rotational latency, typically measuring 7200 RPMs, is the time taken for half of a rotation. It can also be defined as the time waiting for the first bit of target sector to pass under the r/w head. It is given as:

$$T_{avg\ rotation} = 1/2 * 1/\text{RPMs} * 60\ \text{sec}/1\ \text{min}$$

3. **Reading time/Transfer time**: This usually depends on the speed of the spin and it is defined as the time taken to read the bits in the target sector. The disk reads multiple sectors at a times and

then caches them in the memory to be used when needed. It depends on how fast is the spin and the sector density of the track. It is given as:

$$T_{avg\ transfer} = 1/\text{RPM x } 1/(\text{avg sectors/track}) \text{ x } 60 \text{ secs}/1 \text{ min}$$

The access time is much smaller and also dominated by the seek time and the rotational latency.

Following is an example of calculating the disk access time-

We were given the following values:

Rotational rate = 7200 RPM
Average seek time= 9ms
Average Number of sector/tracks = 400

We then calculated the following:

$T_{avg\ rotation}$ = 1/2 *(60 secs/7200 RPM) * 1000 ms/sec = 4 ms
$T_{avg\ transfer}$ = 60/7200 * 1/400 *1000ms/sec = 0.02 ms
$T_{access}$= 9 ms + 4 ms + 0.02 ms = 13.02 ms

# 4 Pipelining

Pipelining is a method used to improve performance by overlapping the execution of operations. A simple analogy is doing laundry, which involves three stages: washing (30 mins), drying (40 mins), and folding (20 mins). Processing one load sequentially takes 90 minutes. So, for 4 loads, doing them one after another takes 6 hours.

However, this approach is inefficient because when the first load is drying, the washer and the person folding are idle. To make better use of these resources, pipelining can be applied. For example, once the first load is in the dryer, we can immediately start washing the second load. If timed right, starting the next wash 10 minutes after drying begins, we can ensure that by the time washing is done, the dryer is also ready. This coordination reduces the total time to around 3.5 hours instead of 6. These 2 cases are depicted in figures 2 and 3.

This same idea applies to computational pipelines, especially in data science. As data flows in, the first component processes it and passes it to the next stage. Each stage continues processing in parallel, aiming to keep all computational units (like CPUs and GPUs) busy at all times to maximize efficiency. Idle hardware, like idle washing machines or humans in the laundry example, results in wasted time and resources.

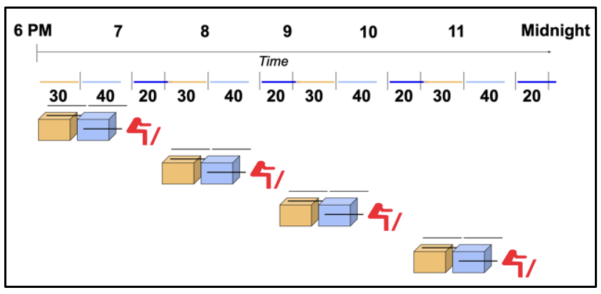


Figure 2: Sequential processing

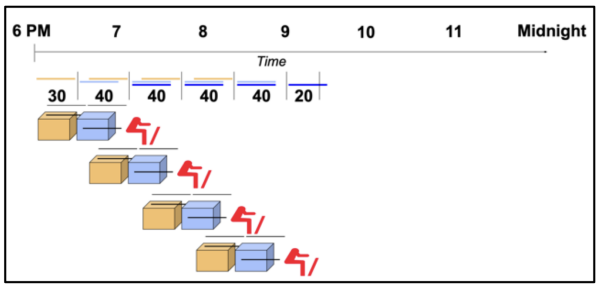


Figure 3: Pipelined processing

# 5 Pipelining Lessons

1. **Latency vs Throughput:**
   Pipelining does not reduce the latency, which is the time taken to complete a single task (in the above example, one load still takes 90 mins even if we use pipelining). What it improves is throughput, which is the number of tasks completed per unit time. For example, completing 4 loads in 3.5 hours instead of 6.

2. **Bottleneck Stage:**
   The throughput of the pipeline is limited by the slowest stage. In the laundry example, drying takes the longest (40 mins), so it sets the pace for the entire pipeline. When designing a computational pipeline, the slowest component determines the overall throughput.

3. **Speedup:**
   Speedup is the ratio of the time taken to execute tasks in a pipelined manner versus sequentially. In a perfectly balanced pipeline (where all stages take equal time), the maximum speedup is equal to the number of stages. However, if the stages are unbalanced, the slowest one limits the speedup.

4. **Pipeline Fill and Drain Time:** There is an initial phase when the pipeline is filling. During this time, some components remain idle. Similarly, at the end, as the last few loads are processed, some stages become free early. These phases reduce the average throughput, especially for a small number of tasks. The pipeline performs best in a steady state with many loads or large datasets.

In real-world data science applications, we often deal with large, streaming datasets. Pipelining ensures efficient processing by keeping expensive resources like CPUs and GPUs constantly engaged, thus maximizing throughput and system performance.

# 6 Instruction Execution and Processing

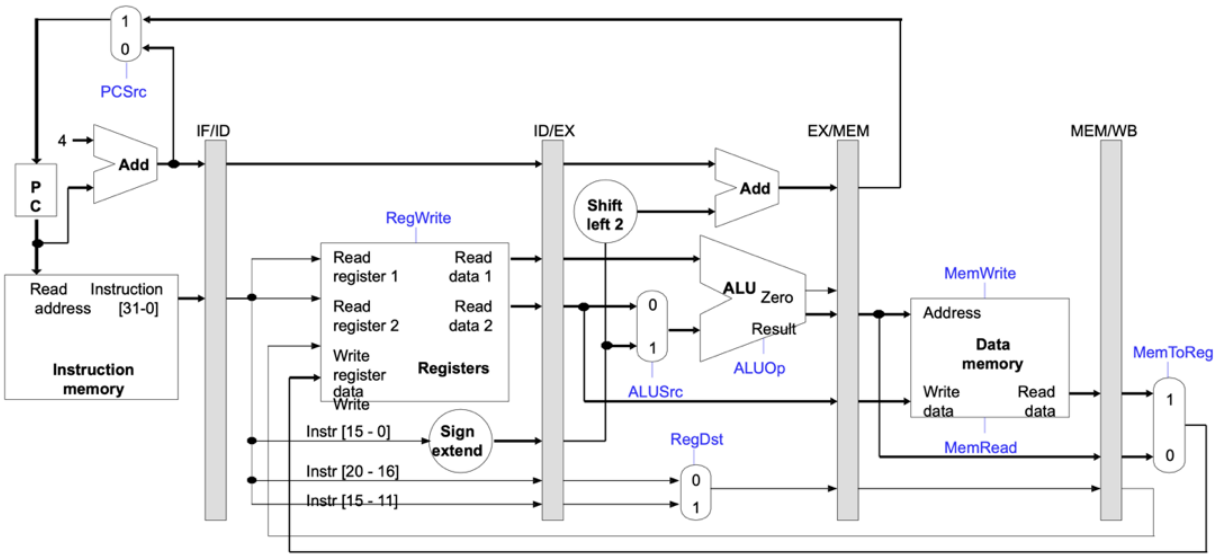


Figure 4: Processor design

Instruction execution in a processor involves a sequence of well-defined steps. These are:

1. **Instruction Fetch (IF)** – The instruction is read from memory.
2. **Instruction Decode (ID)** – The instruction is decoded, source registers are read, and control signals are generated.
3. **Execute (EX)** – The actual computation takes place, such as arithmetic or branch evaluation.
4. **Memory (MEM)** – Memory is accessed if needed to load or store data.
5. **Writeback (WB)** – The result is written back into the register file.

Each of these stages uses different hardware resources: the instruction fetch unit, decoder, ALU (Arithmetic Logic Unit), memory access unit, and register banks. In a traditional, non-pipelined system, when one unit is working (e.g., fetching an instruction), all other units sit idle. This is inefficient, especially when the CPU has numerous computational units, as seen in figure 4. To improve performance, the concept of pipelining is applied. In pipelining, while one instruction is in the decode stage, the next can be fetched. As the first instruction moves to execution, the second starts decoding, and a third can be fetched. This overlap allows multiple instructions to be in different stages simultaneously, keeping all units occupied and boosting overall throughput.

In steady state, when a large number of instructions are being executed, pipelining allows for high throughput, completing one instruction per clock cycle (after the initial pipeline fills). Instead of a long single cycle, the work is spread over multiple smaller cycles, making it possible to run the CPU at a higher clock speed.

# 7 Challenges in pipelining instructions

Pipelining introduces two key challenges:

1. **Pipeline Stalls** Programs often contain data dependencies. For example:

   $$\begin{aligned} A &= B + C \\ D &= A * C \end{aligned}$$

   Here, the second instruction depends on the result of the first. In a pipelined system, if the second instruction reaches the execution stage before the first has completed, it will not have the correct value for A. In such cases, the pipeline must stall, that is, it pauses for a few cycles until the dependent data is ready. These stalls reduce the throughput and are referred to as pipeline hazards.

2. **Pipeline Flush**

   Another issue arises from control flow changes such as branches, conditionals, and loops. Pipelines assume that the next instruction is always the one sequentially after the current one. But with a conditional like if or a loop, the actual next instruction may depend on a condition that isn't known until a later stage. For example, by the time the processor realizes a branch must be taken, it may have already fetched and even started decoding the wrong instruction. These instructions must be flushed from the pipeline, and the correct instruction must be fetched and executed. This process is called a pipeline flush, and it introduces delays, especially in programs with frequent branches or jumps.