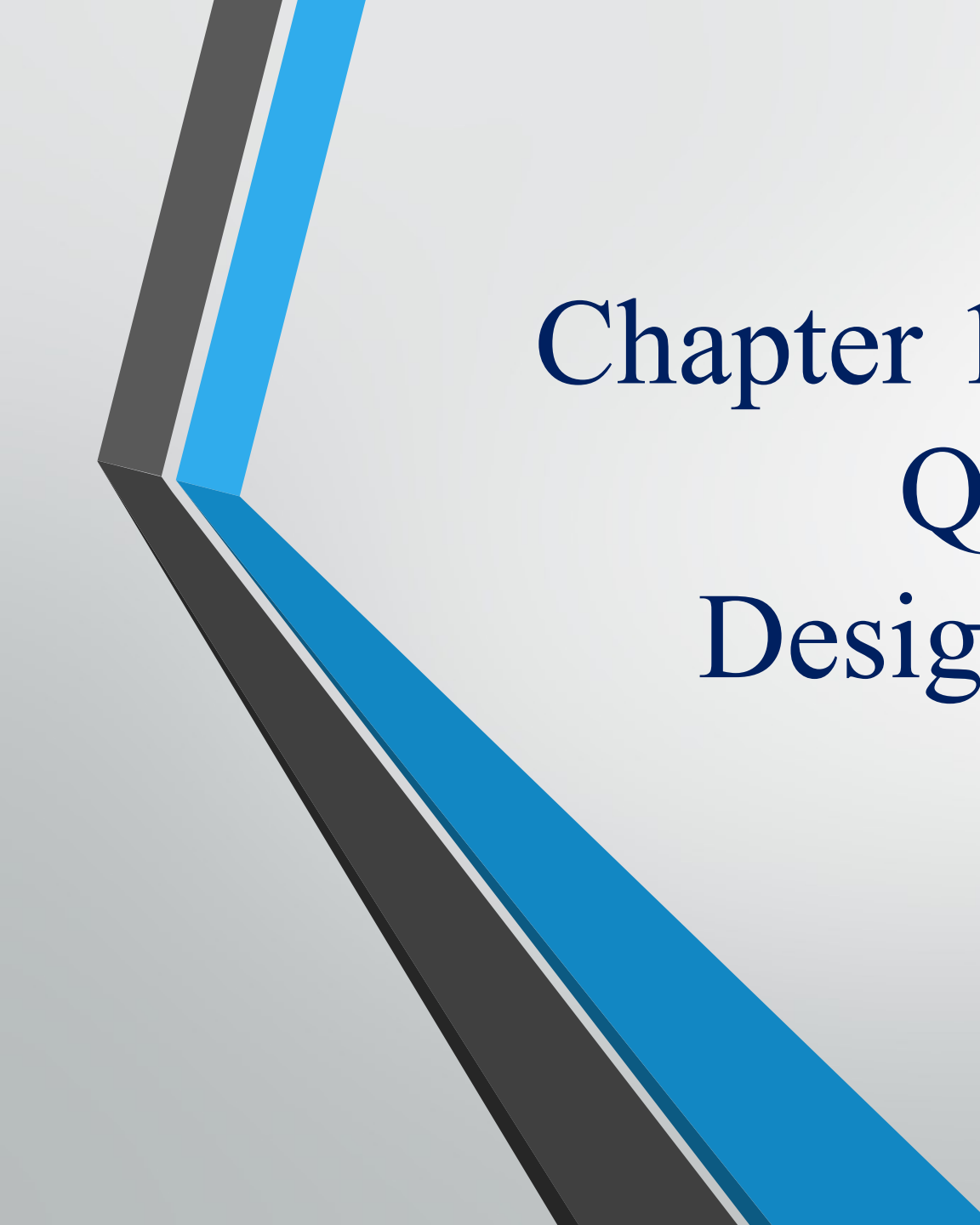




Lecture 2

Dr: Alshaimaa Mostafa Mohammed



Chapter 1: Fundamentals of Quantitative Design and Analysis

Dealing with Complexity: Abstractions

- As an architect, our main job is to deal with tradeoffs
 - Performance, Power, Die Size, Complexity, Applications Support, Functionality, Compatibility, Reliability, etc.
- Technology trends, applications... How do we deal with all of this to make real tradeoffs?
- Abstractions allow this to happen
- Focus is on metrics of these abstractions
 - Performance, Cost, Availability, Power

- In general, energy is always a better metric because it is tied to a specific task and the time required for that task.
- In particular, the energy to execute a workload is equal to the average power times the execution time for the workload.
- Thus, if we want to know which of two processors is more efficient for a given task, we should compare energy consumption (not power) for executing the task.
- For example, processor A may have a 20% higher average power consumption than processor B, but if A executes the task in only 70% of the time needed by B, its energy consumption will be $1.2 \times 0.7 = 0.84$, which is clearly better.

Metrics: Availability

- Availability: Fraction of Time a system is available
- For servers, may be as important as performance
- Mean Time Between Failures (MTBF)
 - Period that a system is usable
 - Typically 500,000 hours for a PC Harddrive
- Mean Time to Repair (MTRR)
 - Recovery time from a failure
 - Should approach 0 for a big server (redundancy)

Performance Metrics

- Execution Time is often what we target
- Throughput (tasks/sec) vs. latency (sec/task)
- How do we decide the tasks?— What the customer cares about, real applications
 - Representative programs (SPEC, SYSMARK, etc)
 - Kernels: Code fragments from real programs (Linpack)
 - Toy Programs: Sieve, Quicksort
 - Synthetic Programs: Just a representative instruction mix (Whetsone, Dhrystone)

Measuring Performance

- In comparing design alternatives, we often want to relate the performance of two different computers, say, X and Y. The phrase “X is faster than Y” is used here to mean that the response time or execution time is lower on X than on Y for the given task. In particular, “X is n times faster than Y” will mean:

$$\frac{\textit{Execution Time}_Y}{\textit{Execution Time}_X} = n$$

Measuring Performance

- Even execution time can be defined in different ways depending on what we count.
- The most straightforward definition of time is called *wall-clock time*, *response time*, or *elapsed time*, which is the latency to complete a task, including disk accesses, memory accesses, input/output activities, operating system overhead—everything.
- With multiprogramming, the processor works on another program while waiting for I/O and may not necessarily minimize the elapsed time of one program. Hence, we need a term to consider this activity.
- *CPU time* recognizes this distinction and means the time the processor is computing, *not* including the time waiting for I/O or running other programs.

Performance Metrics

- Execution Time is often what we target
- Throughput (tasks/sec) vs. latency (sec/task)
 - What the customer cares about, real applications
 - Representative programs (SPEC, SYSMARK, etc)
 - Kernels: Code fragments from real programs (Linpack)
 - Toy Programs: Sieve, Quicksort
 - Synthetic Programs: Just a representative instruction mix (Whetsone, Dhrystone)

Measuring Performance

- Total Execution Time:

$$\frac{1}{n} \sum_{i=1}^n time_i$$

- This is *arithmetic* mean
 - This should be used when measuring performance in execution *times* (CPI=*clock cycles per instruction*)
- Weighted Execution Time:

$$\sum_{i=1}^n weight_i * time_i$$

Measuring Performance

- Normalized Execution Time
- Normalize to *reference* machine Can only use geometric mean (arithmetic mean can vary depending on the reference machine)

$$\sqrt[n]{\prod_{i=1}^n \text{Execution Time } i}$$

Quantitative Principles of Computer Design

- Now that we have seen how to define, measure, and summarize performance, cost, dependability, energy, and power, we can explore guidelines and principles that are useful in the design and analysis of computers:
- **Parallelism**
- **Locality**
- **Focus on the Common Case**

Take Advantage of Parallelism

- Using multiple processors in parallel to solve problems more quickly than with a single processor

Levels Of Parallelism

Parallelism can implement at different levels in a computing system using hardware and software techniques:

1. Data - Level Parallelism

Simultaneously operate on multiple bits of a datum or on multiple data.

Examples :

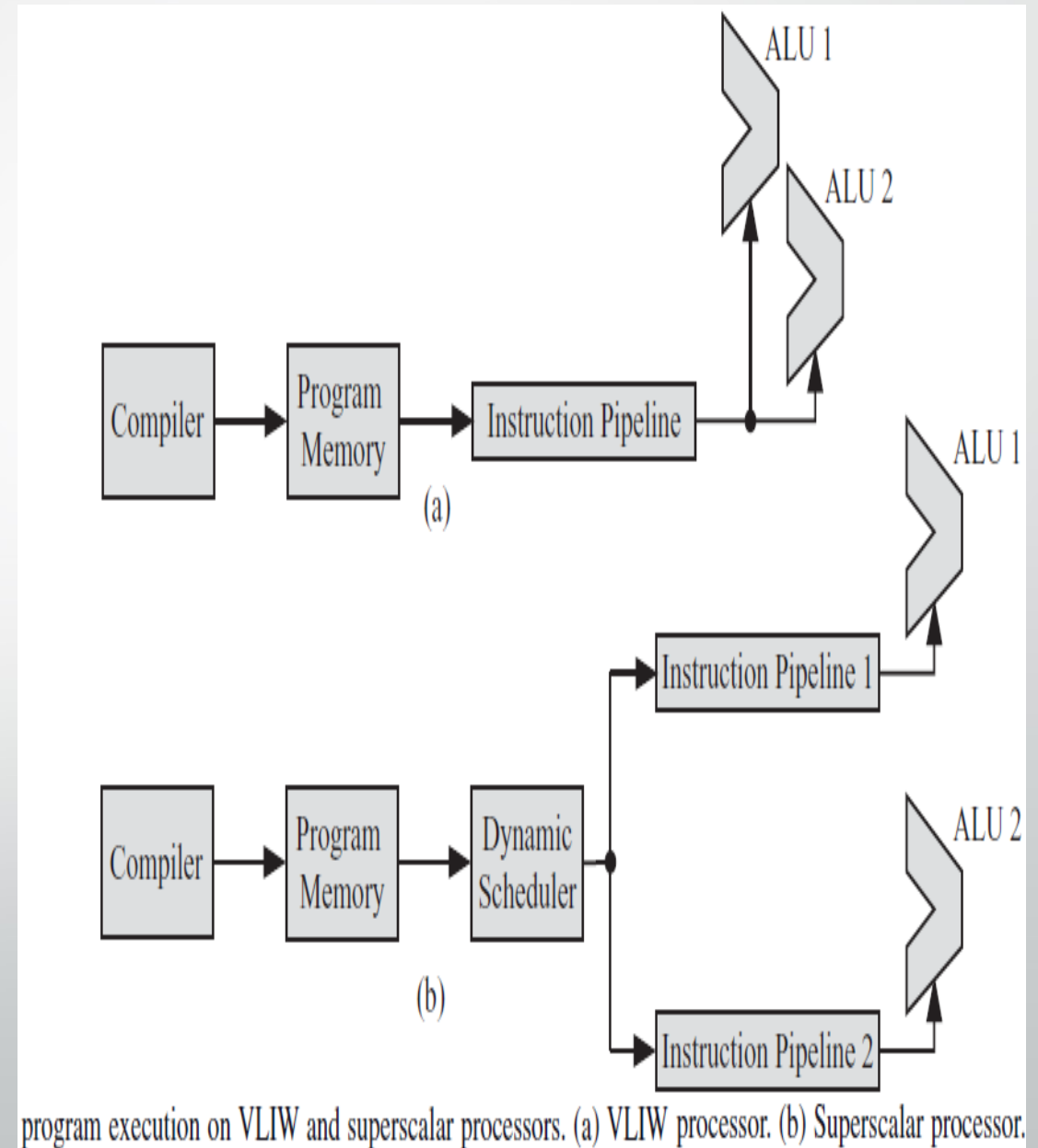
parallel addition, multiplication, and division of binary numbers, vector processors, and systolic arrays for dealing with several data samples.

2. Instruction - Level Parallelism (ILP)

Simultaneously execute more than one instruction by the processor.

Example:

Instruction pipelining.



Levels Of Parallelism- Cont..

3. Thread - Level Parallelism (TLP)

Thread:

A portion of a program that shares processor resources with other threads. (called a lightweight process- Why)

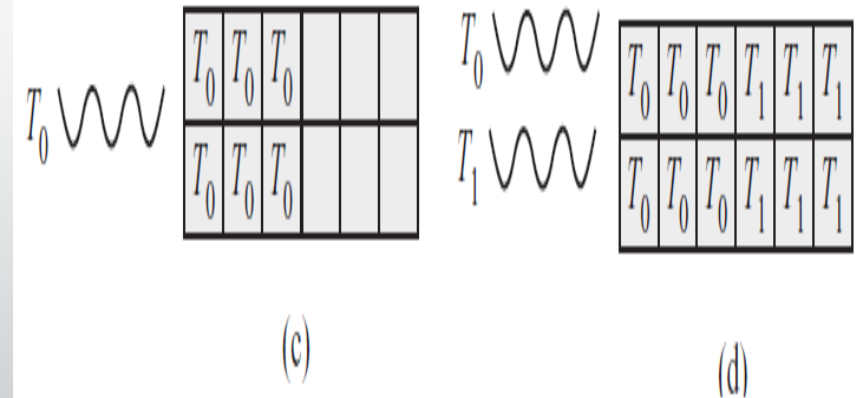
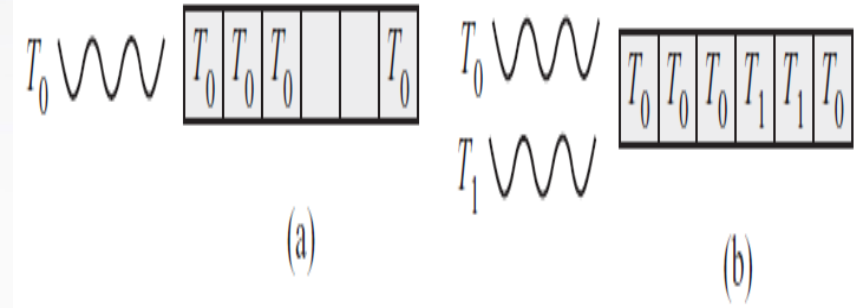
- In TLP, multiple software threads are executed simultaneously on one processor (**Multithreads**) or on several processors (**Multicore**).

4. Process - Level Parallelism.

Process:

a program that is running on the computer.

- A process reserves its own computer resources, such as memory space and registers.
- This is, of course, the classic **multitasking** and **time-sharing** computing where several programs are running simultaneously on one machine or on several machines.



Multithreading in a single processor. (a) Single processor running a single thread.

(b) Single processor running several threads. (c) Superscalar processor running a single thread.

(d) Superscalar processor running multiple threads.

Principle of Locality

- Programs tend to reuse data and instructions they have used recently.
- A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code.
- An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past.
- The principle of locality also applies to data accesses, though not as strongly as to code accesses.

Focus on the Common Case

- Focusing on the common case works for power as well as for resource allocation and performance.
- The instruction fetch and decode unit of a processor may be used much more frequently than a multiplier, so optimize it first.
- It works on dependability as well.
- If a database server has 50 disks for every processor, storage dependability will dominate system dependability.
- In applying this simple principle, we have to decide what the frequent case is and how much performance can be improved by making that case faster.
- A fundamental law, called **Amdahl's law**, can be used to quantify this principle.

Amdahl's Law

- Amdahl's law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Amdahl's law defines the speedup that can be gained by using a particular feature.



SpeedUp

- Suppose that we can make an enhancement to a computer that will improve performance when it is used.
- Speedup is the ratio:

$$\text{Speedup} = \frac{\text{Execution Time for task without enhancement}}{\text{Execution Time for task using enhancement}}$$

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times$$

$$\left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

- Amdahl's law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

1. *The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement*—For example, if 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement, the fraction is $20/60$. This value, which we will call **Fraction_{enhanced}**, is always less than or equal to 1.

2. *The improvement gained by the enhanced execution mode, that is, how much faster the task would run if the enhanced mode were used for the entire program*—This value is the time of the original mode over the time of the enhanced mode. If the enhanced mode takes, say, 2 seconds for a portion of the program, while it is 5 seconds in the original mode, the improvement is $5/2$. We will call this value, which is always greater than 1, **Speedup_{enhanced}**.

$$\text{Speedup}_{\text{Overall}} = \frac{1}{\left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)}$$

As $\text{Speedup}_{\text{Enhanced}} \gg 0$, $\text{Speedup}_{\text{Overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})}$

Example

- Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

- **Sol:**

$$\text{Fraction}_{\text{enhanced}} = 0.4$$

$$\text{SpeedUp}_{\text{enhanced}} = 10$$

$$\text{SpeedUp}_{\text{overall}} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64}$$

Fraction_{enhanced}=95%, Speedup_{Enhanced}=1.1x

$$\text{Speedup}_{\text{Overall}} = 1/((1-.95)+(.95/1.1)) = 1.094$$

Fraction_{enhanced}=5%, Speedup_{Enhanced}=10x

$$\text{Speedup}_{\text{Overall}} = 1/((1-.05)+(.05/10)) = 1.047$$

Make the common
case fast!

Fraction_{enhanced}=5%, Speedup_{Enhanced}=Infinity

$$\text{Speedup}_{\text{Overall}} = 1/(1-.05) = 1.052$$

The Processor Performance Equation

- all computers are constructed using a clock running at a constant rate.
- These discrete time events are called *ticks, clock ticks, clock periods, clocks, cycles, or clock cycles*.
- Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz). CPU time for a program can then be expressed two ways:

CPU time CPU = clock cycles for a program * Clock cycle time

Or

CPU time= CPU clock cycles for a program / Clock rate

The Processor Performance Equation

- In addition to the number of clock cycles needed to execute a program, we can also count the number of instructions executed—the *instruction path length or instruction count (IC)*.
- If we know the number of clock cycles and the instruction count, we can calculate the average number of *clock cycles per instruction (CPI)*.
- Designers sometimes also use *instructions per clock (IPC)*, which is the inverse of CPI.

$$\text{CPI} = \text{CPU clock cycles for a program} / \text{Instruction count}$$

The Processor Performance Equation

- CPU time = Instruction count * Cycles per instruction * Clock cycle time

- Execution Time = seconds/program

