Sri Lanka Institute of Information Technology

BSc (Hons) Computer Systems Engineering

Year 02 Semester 02

Advanced Computer Organization and Architecture - IE2064

Design and Performance Evaluation of a Multicore Computer System

TABLE OF CONTENTS

I. LIST OF FIGURES

II. LIST OF TABLES

III. ABSTRACT

With the increasing demand for high-performance computing applications, multi-core processors have emerged as a core feature of modern processors, enabling multiple instances of parallel execution as well as system responsiveness. This research provides a dual approach of integrating foundational research with a comprehensive evaluation of multi-core systems and performance analysis through simulation. The research focused on key areas such as instruction level parallelism (ILP), multithreading, parallel processing, and cluster computing. Hardware design tradeoffs are examined in detail through a review of the Intel Core i7 microarchitecture, focusing on areas such as execution pipelines, cache strategies, and scheduling practices. Additionally, the research examines the performance bottlenecks introduced by both hardware and software in multi-core systems, covering parameters such as thread contention, memory access, and synchronization overhead.

To model the system and analyze performance metrics, a simulation in Python is created to simulate a simplified 4-core processor. This implementation defines three different CPU scheduling algorithms, Round Robin, Shortest Job First (SJF), and Priority Scheduling to compare their effects on execution time, CPU utilization, and context switches under different workloads. The simulation results show significant trade-offs in terms of fairness, response time, and throughput between the different scheduling methods. The performance is analyzed for both traditional symmetric multiprocessing (SMP) as well as cluster architectures to highlight both the inherent benefits and limitations of multicore systems. The findings provide important insights into the impact of scheduling approaches on overall system performance and enable optimizations in task management in multicore environments.

IV. INTRODUCTION

The evolution of the processor and its performance has historically followed the path that was predicted by Gordon Moore which was proposed as the Moore's Law in 1965. His prediction was that the number of transistors on an integrated circuit would double each year. He stated that "the new slope might approximate a doubling every two years, rather than every year. (Moore)". His prediction or the trend he created led the silicon industry to another level which boosted the integrated circuit's complexity which also led to greater performance and other improvements.

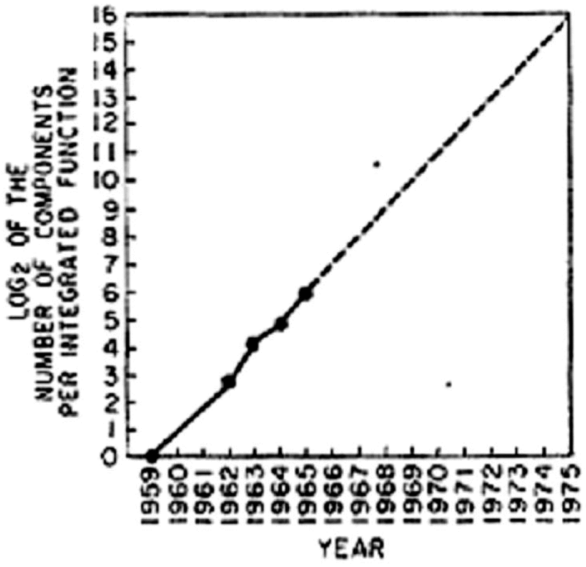

Fig 1: Log2 of the number of components per integrated function. (Courtesy Penton Media. Reprinted from G.E. Moore, "Cramming More Components onto Integrated Circuits," Electronics, vol. 38, no. 8, 19 Apr. 1965, pp. 114–117.) (Tatjana R. Nikolić, 2022)

As this exponential growth of transistors and other components continued, single core processors became faster and more efficient. But this was not a long-term solution. There came a point when the number of components could no longer be increased. This was due to physical, thermal and performance limitations that results when increasing the components inside a CPU. By the early 2000s, these limitations started to impede CPU frequency scaling, resulting in reduced performance gains even from higher clock speeds. This is the point where they understood that further frequency scaling is unsustainable. This marked a turning point that led to the debut of a new computing paradigm, **Multicore Architectures**.

Multicore architecture represented a fundamental shift in designing processors and deviating from the race of frequency scaling. Instead, multicore architectures are leveraging **parallelism**, to achieve more performance gains. Within a multi-core processor, multiple processing units which are known as **cores** are integrated on a single chip. Each individual chip can execute its own thread of instructions simultaneously. This approach is a very effective solution for the addressed limitations on single core processors, which allow greater throughput by performing multiple tasks concurrently.

V. BACKGROUND

The transition from frequency scaling from single-core processing to parallel computing marks a key milestone in processor architecture and design. The traditional architectural enhancements like pipelining and superscalar execution approached physical limits as above mentioned, but multicore systems introduced **Instruction-Level Parallelism (ILP)** and **Thread-Level Parallelism (TLP)** to the processors. This evolution tends to High-Performance Computing (HPC), embedded systems, and other computing applications.

VI. PROBLEM STATEMENT

Despite their advantages, multicore systems present complex challenges in both hardware and software domain. Efficient resource utilization depends heavily on operating system-level scheduling, load balancing, and memory management. “Nowadays, the usage of multi-/many-core processors is a big challenge for software developers. They now must master the programming techniques necessary to take full advantage of the potential of multi-/many-core processing. (Tatjana R. Nikolić, 2022)” Issues such as cache coherence, thread synchronization, and power efficiency makes the design and optimization of multicore processors more complicated. In addition, comparative architectural models like **Symmetric Multiprocessing (SMP)** and **Clusters** present alternative paradigms, each with distinct trade-offs in latency, power, and scalability.

VII. OBJECTIVE

This study aims to analyze the architectural and performance characteristics of multicore systems, with a focus on design trade-offs, parallel execution efficiency, and system-level behavior under varying workloads. This study is driven by a simulation-based approach that is adopted to model and analyze execution patterns across multicore, SMP, and cluster systems. We particularly emphasize the Intel Core i7 architecture as a case study for deep architectural analysis.

VIII. LITERATURE REVIEW & THEORETICAL FRAMEWORK

A. INSTRUCTION-LEVEL PARALLELISM (ILP)

Instruction-Level Parallelism refers to the multiple concurrent executions of instructions in a processor within a single thread of execution. “The opportunities to speed up program execution through instruction-level parallelism exploitation became feasible because segments of the program can be executed at the same time as compared with sequential execution speeds (Misra, Alfa, Olaniyi, & Adewale, 2014).”

Pipelining and Super Pipelining

The most common way to utilize ILP is through pipelining. In a pipelined processor, the process of executing an instruction is separated into several stages such as fetch operation, decode operation, execute operation and write-back operation. This way the structure can concurrently execute multiple instructions with each instruction allocating a separate stage in the pipeline at a unit time. As a result, overall instruction throughput increases significantly, leading to improved performance

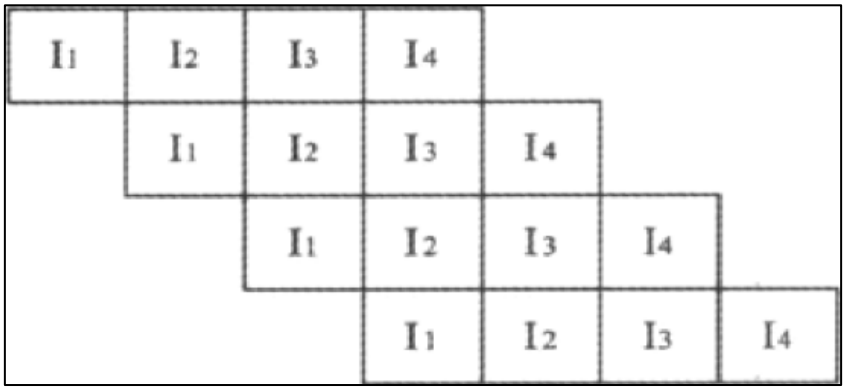

Fig 2: a set of operating instructions using traditional Pipelined technology (Li, Ling, & Wu, 2011)

As shown in the above diagram, Li and colleagues (Li, Ling, & Wu, 2011) demonstrates the principal of pipelining is like an industrial production assembly line, which also has operations divided into many times balanced operations. And they stated that an ideal pipeline operation is very efficient. “Instruction in the program is still a section of the order of execution, but can take several instructions in advance, and when the current instruction has not executed it start-up some operating procedures of follow-up instructions in advance. Obviously, this can speed up a program running. (Li, Ling, & Wu, 2011)”

Despite its benefits, traditional pipelining suffers from several pipeline hazards like data, control, and structural hazards, which causes pipeline stalls and loss of efficiency. The performance boost is bottlenecked by the slowest stage of the pipeline, and the method is less effective if branch instructions interfere with the instruction stream.

Super Pipelining is an advancement that was made in traditional pipelining, which aims to improve the performance by introducing additional pipeline stages, which enables a higher clock frequency and faster execution at each stage. According to Misra and others (Misra, Alfa, Olaniyi, & Adewale, 2014), super pipelining improves performance by executing two tasks within one external clock cycle by exploiting the fact that many pipeline stages complete operations in less than half a clock cycle.

Misra and colleagues further explain that super pipelining architectures also include four stages as instruction fetch, decode, execution, and write-back same as the traditional pipelining (Misra, Alfa, Olaniyi, & Adewale, 2014).This architecture allows for greater efficiency by doubling internal clock speed, thus enabling the parallel execution of tasks that would otherwise take multiple clock cycles. This technique significantly reduces idle times in the pipeline stages and increases the number of instructions that can be concurrently processed.

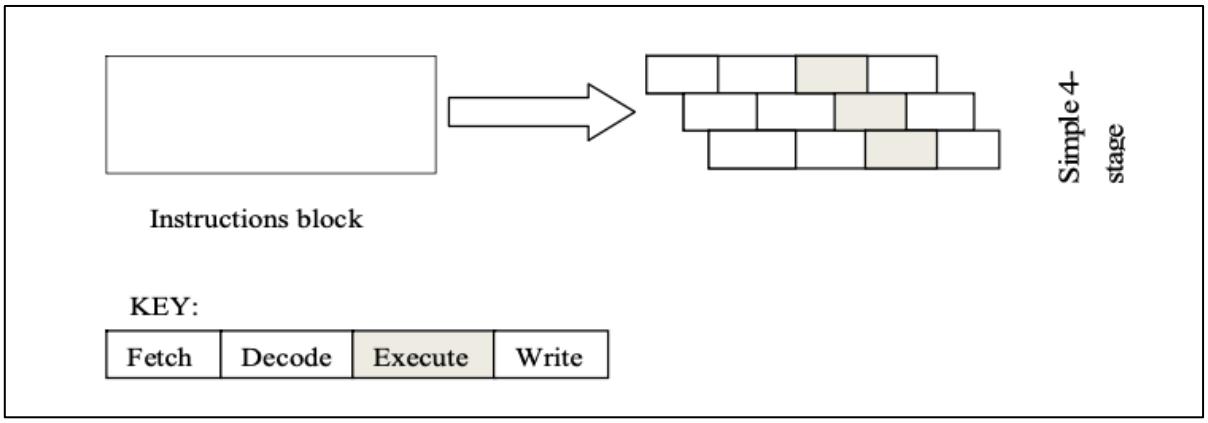

Fig 3: Super pipelining architecture (Misra, Alfa, Olaniyi, & Adewale, 2014)

Superscalar Architecture

Superscalar architecture is a fundamental upgrade in processor design aimed at increasing ILP within a single thread. This design allows the CPU to issue and execute multiple instructions per clock cycle with the help of multiple execution units. Shah et al. demonstrated the basic scalar and vector classification of processors (Shah, Shah, & Modi, 2013). A scalar processor follows the concept of executing a single data item per instruction during a single clock cycle. In contrast, a vector processor operates on an entire set of data items (an array) using a single instruction, thereby achieving data-level parallelism. While scalar processors handle one operation at a time, vector processors are optimized for tasks that involve repetitive computations over large data sets, such as those found in scientific computing or multimedia applications. Superscalar architecture is cooperated with both vector and scalar processing techniques.

As they stated, there are few unique techniques used in Superscalars that aids to obtain performance.

a) Out-of-Order Execution

This technique allows instructions to be executed as soon as their operands are ready, rather than strictly following program order. It reduces pipeline stalls and increases utilization of execution units.

"For example, there are 5 consecutive arithmetic instructions to be executed, which use adder and multiplier. An adder takes 2 cycles for execution and a multiplier takes 5 cycles for the same. The 5 processes are V, W, X,Y and Z. respectively. If they are executed in order, the execution that occurs is as shown in the figure below. (Shah, Shah, & Modi, 2013)"

Below depicts the difference between normal execution and out-of-order execution of the five instructions.

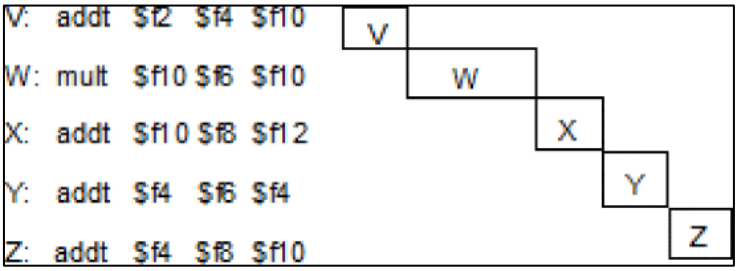

Fig 4: Normal execution of five instructions (Shah, Shah, & Modi, 2013)

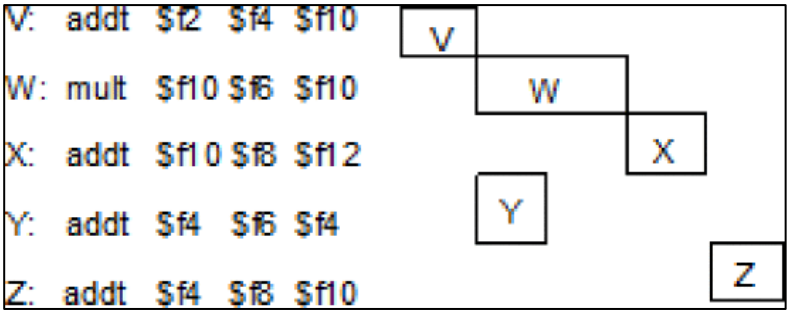

Fig 5: Out of order execution of five instructions (Shah, Shah, & Modi, 2013)

Total time can be reduced by running two instructions parallel as W and Y was executed simultaneously. Therefore, we can reduce the execution time and increase performance by a considerable magnitude.

b) Register Renaming

Used to resolve false data dependencies such as Write-After-Write or Write-After-Read hazards, superscalar processors dynamically allocate physical registers to prevent conflicts, enabling more instructions to execute in parallel.

Very Long Instruction Word (VLIW) Architectures

VLIW architecture leaves the scheduling task to the compiler. The processor in VLIW architecture combines various independent instructions into one long instruction word and executes it concurrently through multiple functional units.

This approach simplifies hardware design by eliminating the need for complex runtime scheduling. VLIW architectures are highly efficient for predictable, regular workloads where the compiler can identify and schedule independent instructions with high precision.

VLIW architectures face major difficulties in handling control flows and irregular workloads because of the complex tasks encountered by the compiler in finding appropriate ILP in such scenarios. These architectural setups are usually deployed in embedded systems and digital signal processors (DSPs). Intel Itanium architecture is a perfect example where VLIW architecture is implemented, where the simultaneous execution of multiple instructions per cycle depends on competence of the compiler in efficiently grouping and aligning instruction bundles.

B. MULTITHREADING AND THREAD LEVEL PARALLELISM

In multicore processors, we earlier discussed about ILP using the techniques such as super pipelining, superscalar architectures and VLIW architectures. However, these techniques are applied within a single thread of execution. In this context single thread of execution means a separate path of execution with shared resources which can also enables concurrency. To further enhance processor performance, modern computer architectures exploit the concept of **Thread-Level Parallelism (TLP)** by allowing multiple threads to be executed simultaneously which increases the concurrency of the executions of processors. TLP focuses on executing independent instructions in parallel streams or paths. This enables better utilization and efficiency of the processor. There are 4 main approaches of multithreading,

1) Simultaneous Multithreading (SMT)

2) Chip Multiprocessing

3) Interleaved multithreading

4) Blocked multithreading

In 1997 Susan and her colleagues (Eggers, et al., 1997) did research about SMT and they stated that they believe the only way to gain significant amount of performance is by enhancing the

processor's computational capabilities. By that they meant increasing the parallelism in all available forms. "Placing multiple superscalar processors on a chip is also not an effective solution, because, in addition to the low instruction-level parallelism, performance suffers when there is little thread-level parallelism. A better solution is to design a processor that can exploit all types of parallelism well. (Eggers, et al., 1997)"

In this review, we will be focusing on SMT and CMP, as these two techniques represent the most prevalent and impactful approaches in modern multicore processor architectures. Their widespread adoption is largely attributed to their scalability and their ability to exploit both ILP and TLP effectively. Compared to interleaved or blocked multithreading techniques that have seen limited adoption in contemporary high-performance computing. SMT and CMP offer more practical and efficient pathways to leveraging parallelism in real-world applications.

Simultaneous Multithreading (SMT)

Simultaneous Multithreading is a processor design that exploits all types of parallelism because it can exploit both ILP and TLP. "Simultaneous multithreading combines hardware features of wide-issue superscalars and multithreaded processors. From super-scalars, it inherits the ability to issue multiple instructions each cycle; and like multithreaded processors it contains hardware state for several programs (or threads) (Eggers, et al., 1997)".

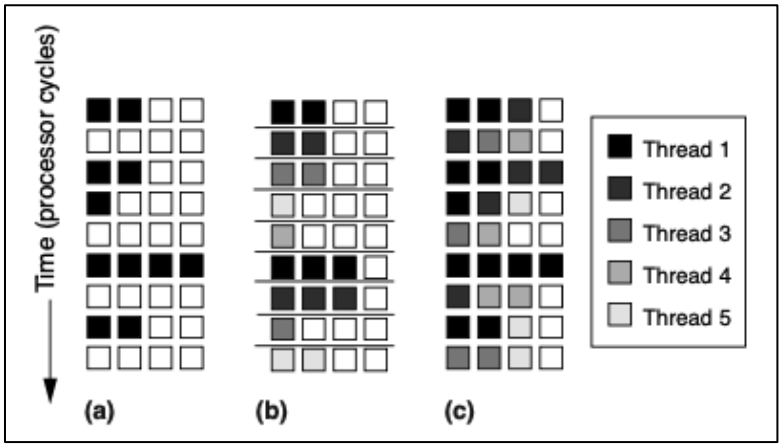

Fig 6: How architectures partition issue slots (functional units): a superscalar (a), a fine-grained multithreaded superscalar (b), and a simultaneous multithreaded processor (c). The rows of squares represent issue slots. The processor either finds an instruction to execute (filled box) or the slot goes unused (empty box). (Eggers, et al., 1997)

Even though SMT significantly enhances processor throughput by allowing multiple threads to issue instructions in the same cycle, it introduces several architectural trade-offs.

SMT architecture can lead to a demand of shared resources among threads like caches, memory bandwidth and execution units. When a demand occurs, there will be a competition between threads and lead to an interference and degrade the overall performance. And regarding the

physical factors, the more concurrent executions happen, the more heat generation and power consumption occurs.

Chip Multiprocessing (CMP)

Chip Multiprocessing, also known as multicore processing, is a processor architecture where multiple independent cores are integrated onto a single chip. Each core can execute its own thread or process. This allows the true nature of parallel execution and improved throughput in multithreaded and multitasking environments. CMP also enhances scalability by distributing workloads across cores.

Table 1. Characteristics of superscalar, simultaneous multithreading, and chip multiprocessor architectures.

Characteristic	Superscalar	Simultaneous multithreading	Chip multiprocessor
Number of CPUs	1	1	8
CPU issue width	12	12	2 per CPU
Number of threads	1	8	1 per CPU
Architecture registers (for integer and floating point)	32	32 per thread	32 per CPU
Physical registers (for integer and floating point)	32 + 256	256 + 256	32 + 32 per CPU
Instruction window size	256	256	32 per CPU
Branch predictor table size (entries)	32,768	32,768	8 × 4,096
Return stack size	64 entries	64 entries	8 × 8 entries
Instruction (I) and data (D) cache organization	1 × 8 banks	1 × 8 banks	1 bank
I and D cache sizes	128 Kbytes	128 Kbytes	16 Kbytes per CPU
I and D cache associativities	4-way	4-way	4-way
I and D cache line sizes (bytes)	32	32	32
I and D cache access times (cycles)	2	2	1
Secondary cache organization (Mbytes)	1 × 8 banks	1 × 8 banks	1 × 8 banks
Secondary cache size (bytes)	8	8	8
Secondary cache associativity	4-way	4-way	4-way
Secondary cache line size (bytes)	32	32	32
Secondary cache access time (cycles)	5	5	7
Secondary cache occupancy per access (cycles)	1	1	1
Memory organization (no. of banks)	4	4	4
Memory access time (cycles)	50	50	50
Memory occupancy per access (cycles)	13	13	13

Table 1: Characteristics of superscalar, simultaneous multithreading, and chip multiprocessor architectures (Olukotun, 1997).

Table 1 presents a detailed comparison of Superscalar, SMT and CMP architectures. From the comparison, it is evident that the Superscalar architecture lags behind SMT and CMP in terms of parallelism capabilities and resource utilization. While the Superscalar model relies on a single CPU with wide issue widths, both SMT and CMP offer enhanced parallelism, SMT through multiple threads per core and CMP through multiple cores on a single chip. This highlights the superior scalability and efficiency of SMT and CMP, especially in multithreaded and multitasking environments. Therefore when combining the CMPs individual core's performance, the output performance much larger and more sustainable than SMT.

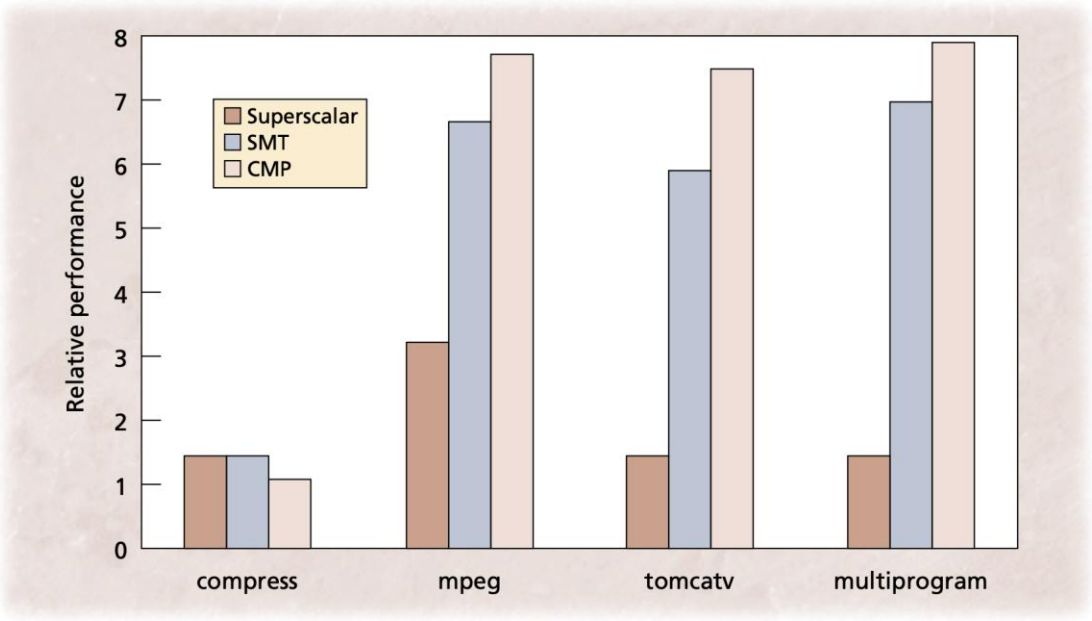

Fig 7: Relative performance of superscalar, simultaneous multithreading, and chip multiprocessor architectures compared to a baseline, 2-issue superscalar architecture (Olukotun, 1997).

Even with the advantages there are some drawbacks in CMP. In 1997, Nayfeh et al. stated that when an application cannot be effectively decomposed into threads, CMPs will be underutilized (Olukotun, 1997). Therefore when the applications are sequential or poorly parallelized, the multiple cores of a CMP tends be idle, leading the processors performance to be bottlenecked due to insufficient TLP. Therefore, the challenge is to design the applications to be compatible with CMP architecture.

C. DESIGN ISSUES AND PERFORMANCE TRADE-OFFS IN MULTICORE ARCHITECTURE

Multicore architectures are physical processors consisting multiple cores, each with the capability if executing instructions independently. Each core typically share components like Last-Level Cache (LLC), memory buses and interconnects. “Multicore processor gives the functionality of parallel processing with reduced sustainable computation time (Jadon & Yadav, 2016).”The internal structure of the multicore architecture is as shown below,

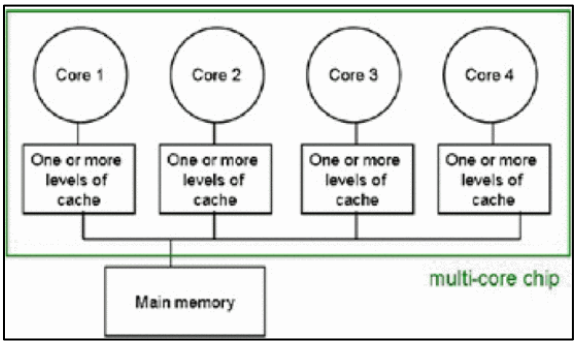

Fig 8: A multicore chip. (Jadon & Yadav, 2016)

"The figure shows four cores having one or more caches and each core is connected via an interconnection network to the memory of the system. (Jadon & Yadav, 2016)"

Multicore architectures can be classified into two,

a) Homogeneous Multicore Architectures
b) Heterogeneous Multicore Architectures

Homogeneous Multicore Architectures

Homogenous multicore architectures consist of multiple identical cores on a single chip. Each core consists of it's own instruction set, architecture and characteristics of performance. All cores are exactly identical within each other and this architecture simplifies the design, scheduling and load balancing.This architeecture can be advantageous when it comes to Symmetric Multiprocessing(SMP).

- Intel Core i5 and i7 CPUs includes this architecture

Heterogeneous Multicore Architectures

Heterogenous multicore architecture consists of different types of cores in a single system. These cores vary in performance, power efficiency and their functionality. It consists of high-performance and low-power cores for dynamic tasks. High-Performance cores are utilized for specialized tasks and low-power cores are used for general -purpose tasks. This architecture has better energy efficiency and flexible workload allocation.

- ARM big.LITTLE architecture implements this architecture

In 2016 Shruti Jadon et al. did research about Multicore Processors, which highlights about the issues and challenges (Jadon & Yadav, 2016). Generally Shruti and the colleagues pointed out cache related issues, scheduling issues, consistency issues and so on. Let us dive deeper into the issues and tradeoffs with the study they have conducted on.

Design Issues and Performance Trade-offs

a) Cache Related Issues

1) Amount of cache

Applications that has high data reusability benefits having larger caches, because they frequently access data remains available on chip. Also larger chaches reduce the memory access latency, which results in enhanced performance through hit rates and reduces main memory accesses. But the issue is the amount of cache is application dependent and implementing larger caches is more costly. “Bigger the size of cache faster will be the speed of accessing and better will be the performance but higher will be the cost. (Jadon & Yadav, 2016)”

2) Number of cache levels

There is hierachy arrangement in memory caches that refers to the number of cache levels between CPU cores and the main memory. This hierachical configuration mains the balance of speed and storage capacity.

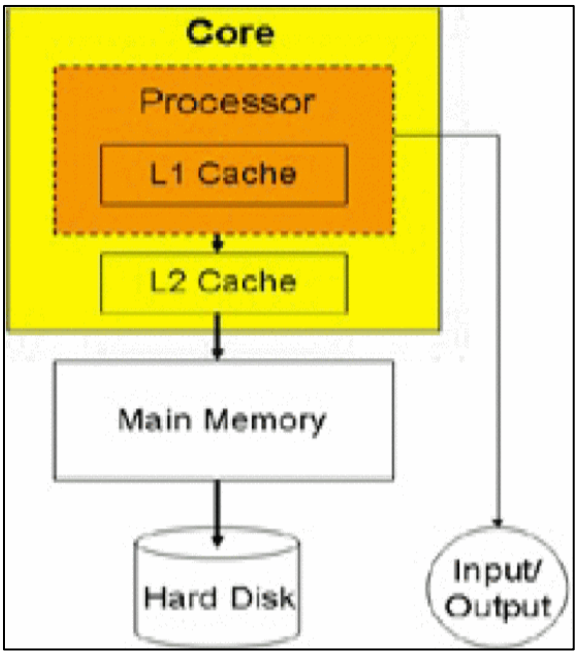

Fig 9: Core with cache levels. (Jadon & Yadav, 2016)

Below depicts the typical multilevel cache structure of modern processors,

i. **L1 Cache** – Located closest to the processor core and the smallest cache but the fastest.
ii. **L2 Cache** – Larger but slighly slower than L1 Cache
iii. **L3 Cache** – Even larger but slower than L2 Cache, and often share among multiple cores
iv. **L4 Cache** – Additional layer before main memory

"Another cache issue is to decide the number of cache levels a cache of multicore may have. It is not necessary for the entire cores cache to have equal number of cache levels. Basically the number of cache levels is decided by how far the main memory is or how many cycles will it take to access the main memory. More the number of cycles more will be the cache levels and faster is the accessing. (Jadon & Yadav, 2016)"

3) Hit/miss rates

Cache hit/miss rates plays a critical role in perfomance indicators for processor memory subsystems. A cache "hit" occurs when reqested data is found in the cache hierachy. And a "miss" occurs when a "hit" is not found and has to access data from slower main memory.

The issue with hit/miss rates is to maintain a higher hit rate than miss rates for a ideal processor."It is always preferred to have higher hit rate than miss rates so that the time of accessing is reduced and the computation is fast. This factor is dependent on the size of caches, the writing and page replacement strategy used by the cache and the number of cache levels. (Jadon & Yadav, 2016)"

4) Scheduling Issues

The goal of multicore scheduling is to optimize system throughput, minimize response time while fully utilizing available resources.

Multicore processors must efficiently handle diverse task types including,

1) Normal vs Real-time tasks
2) Independent vs Dependent tasks
3) Priority vs Non-priority tasks
4) Real-time subcategories

"Thus to schedule the tasks with their deadline constraint is the job of any scheduling algorithm and in multicore processor this issue incorporates with how efficiently the tasks are scheduled such that the core are fully exploited. (Jadon & Yadav, 2016)"

D. INTEL CORE MICROARCHITECTURE AND INTEL CORE I7

Introduction

The evolution of intel processor architecture has a significant impact on development of high perfomance computer systems. They have succeeded to enable enhanced efficiency, parallelism and improved power efficiency. The debut of the Intel Core microarchitecture happened in 2006 with the release of the Core 2 Duo processors. This was a massive leap compared to the earlier NetBurst architecture Intel have used in Pentium 4 CPUs. With this release on the same year, Jack Doweck, the lead architect at the Intel Corporation release the conference proceeding highlighting the key innovations of the Core Microarchitecture (Doweck, 2006).

The key innovations are as listed below,

- Intel® Wide Dynamic Execution
- Intel® Advanced Digital Media Boost
- Intel® Smart Memory Access
- Intel® Advanced Smart Cache
- Intel® Intelligent Power Capability

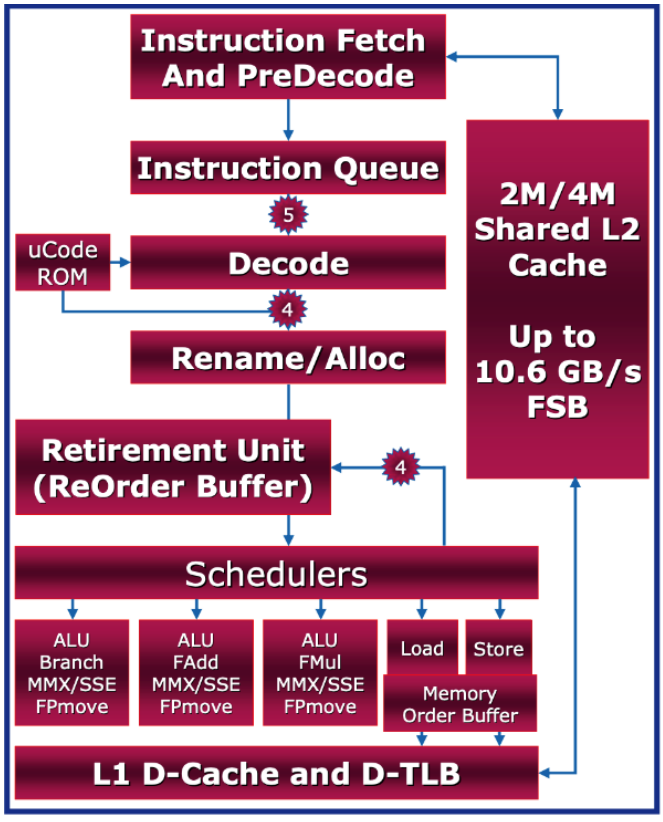

Fig 10: Overview of the Intel Core Microarchitecture (Doweck, 2006)

a) Intel® Wide Dynamic Execution

This feature allows the processor to execute more instructions per clock cycle by enabling simultaneous processing of multiple instructions throughout a wider execution units. It improves overall performance and efficiency.

b) Intel® Advanced Digital Media Boost

This feature accelerates the execution of multimedia and SIMD instructions, which is typically used in video, audio and graphics processing. This feature also enables 128-bit instructions to be completely executed at a throughput rate of one per clock cycle, effectively doubling on a per clock basis compared to the previous generations.

c) Intel® Smart Memory Access

This feature is used to optimize memory operations to enhance the perfomance and efficiency, senarios like gaming and high graphics processing applications. This feature allows the CPU to access the full capacity of the GPU's VRAM directly. The results in minimized latency and fast data transfers between CPU and GPU.

d) Intel® Advanced Smart Cache

This is a shared cache architecture designed to enhance multicore processor efficiency. Instead of having separate L2/L3 cache for each core, the cache is share dynamically among cores, enabling better utilization of cache resources, minimized latency and minimized power consumption. This will also prevent the idling of cache space when some cores are inactive.

e) Intel® Intelligent Power Capability

Intelligent Power Capability is a feature for power management aiming to reduce power consumption without sacrificing performance. A concept called as ***Advanced Power Gating*** was used to control the state of individual processor logic subsystems when not in use. Basically it is used as an switch.

Intel Core i7

Intel's Core i7 series was first announced in 2008, released a family of high performance CPUs that was designed for high performance tasks at that time. I7 series was initially released for desktops and laptops which had a balance between performance and affordability. I7 series was has released multiple architectural upgrades across Intel's CPU generations. Each release was specified by it's microarchitecture name. Some of are given as below,

- Nehalem
- Sandy Bridge
- Skylake
- Alder Lake

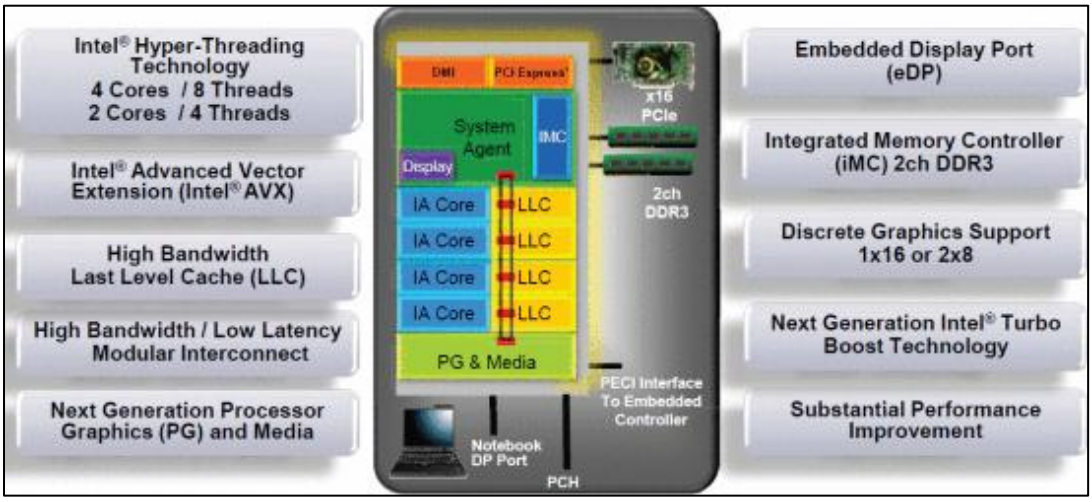

Fig 11: Overview of Intel Core Microarchitecture (Lempel, 2011)

When the Core i7 released, it came with significant upgrades compared to the previous core architectures. Core i7 processor consists a higher number of cores and threads compared to previous processors. Also a larger cache with intel hyperthreading technology and support of virtualization.

Below is the evolution of the Intel Core i7 processor series, which highlights the continuous architectural and technological advancements introduced by Intel since the launch of the first generation in 2008.

Generation	Release Year	Codename	Microarchitecture	Process Technology	Key Features
1st Gen	2008–2009	Bloomfield, Lynnfield	Nehalem	45nm	First Core i7, integrated memory controller, QPI interconnect, Hyper-Threading
2nd Gen	2011	Sandy Bridge	Sandy Bridge	32nm	AVX instruction set, Turbo Boost 2.0, integrated GPU on die
3rd Gen	2012	Ivy Bridge	Ivy Bridge	22nm	3D Tri-Gate (FinFET) transistors, HD 4000 graphics, PCIe 3.0
4th Gen	2013	Haswell	Haswell	22nm	AVX2, FMA3, fully integrated voltage regulator (FIVR), power efficiency
5th Gen	2015	Broadwell	Broadwell	14nm	Focus on mobile, limited desktop release, Iris Pro graphics
6th Gen	2015–2016	Skylake	Skylake	14nm	DDR4 support, improved iGPU, new socket, removed FIVR
7th Gen	2016–2017	Kaby Lake	Kaby Lake	14nm+	Better 4K video support, increased clock speeds, refined 14nm
8th Gen	2017–2018	Coffee Lake	Coffee Lake	14nm++	Increased core count (up to 6 cores), improved performance
9th Gen	2018–2019	Coffee Lake Refresh	Coffee Lake Refresh	14nm++	Up to 8 cores, STIM (soldered TIM), hardware fixes for Spectre/Meltdown
10th Gen	2019–2020	Comet Lake / Ice Lake	Skylake / Sunny Cove	14nm++ / 10nm	Hybrid release: up to 10 cores (Comet Lake), Gen11 graphics (Ice Lake)

11th Gen	2020–2021	Rocket Lake / Tiger Lake	Cypress Cove / Willow Cove	14nm++ / 10nm SuperFin	PCIe 4.0, Xe graphics, significant IPC uplift
12th Gen	2021–2022	Alder Lake	Golden Cove (P) / Gracemont (E)	Intel 7 (10nm Enhanced)	Hybrid architecture (P-core/E-core), DDR5 & PCIe 5.0 support
13th Gen	2022–2023	Raptor Lake	Raptor Cove (P) / Enhanced Gracemont (E)	Intel 7	More E-cores, larger L2/L3 cache, better thermals
14th Gen	2023–2024	Raptor Lake Refresh	Raptor Cove / Enhanced Gracemont	Intel 7	Slight performance gains, power optimization
15th Gen	2024	Arrow Lake	Lion Cove (P) / Skymont (E)	Intel 4 (7nm EUV)	New socket (LGA 1851), integrated Arc GPU, large IPC jump

Table 2: Evolution of i7 processors

E. ANALYZE HARDWARE AND SOFTWARE PERFORMANCE ISSUES RELATED TO MULTICORE PROCESSING

These improvements faced the limitations of single-core performance gains, such as thermal constraints and diminishing returns from frequency scaling. While introducing parallelism for multicore processors could help to enhance performance and its actual effectiveness is often constrained by a range of hardware and software bottlenecks.
At the hardware level, performance on multicore systems is influenced by resource contention, particularly in shared caches (L2/L3), memory bandwidth, and DRAM access latency (Diamond, 2011). Performance can degrade non-linearly with increased core usage because multiple cores compete for limited off-chip bandwidth and shared cache space. This is called the diminishing returns on scaling phenomenon and is identified as a major challenge to both system architects and application developers.

Issue	**Impact**
Shared L3 Cache Capacity	As cache pressure increases with the number of cores, cache misses increase.
Off-Chip Bandwidth	High memory traffic exceeds available bandwidth
DRAM Page Conflicts	Switches introduce latency through DRAM pages and reduce effective bandwidth.

Table 3: hardware performance issues

At the software level, existing optimization techniques ameliorated for single processors frequently fail to scale effectively in multicore environments. Memory access patterns, compiler flags, and synchronization primitives often introduce unforeseen overheads when executed concurrently across cores. Apart from that, traditional performance metrics such as cache miss rates may no longer provide accurate diagnostics in multicore settings, necessitating more nuanced tools and metrics (e.g., instructions per cycle per core, memory footprint per core).

Issue	Impact
Misleading Metrics	The bottleneck is hidden in the memory hierarchy (e.g. L3 misses).
Compiler Flag Sensitivity	Irregular memory access patterns hurt multicore scaling.
Heavyweight Tool Overhead	Distorted performance data can improve the measurement code instead of the application.

Table 4: software performance issues

F. SYMMETRIC MULTIPROCESSING & CLUSTERS

Symmetric multiprocessing and Clusters are two parallel processing techniques used to improve computing performance, and they differ in factors such as scaling capabilities and architectures. This may involve using multiple computing nodes or a single node consisting of multiple processors. These architectures are explained in detail below.

Symmetric multi-processing

Symmetric multiprocessing (SMP) represents one of the latest architectures used for extensive calculations. “In computing, SMP involves a multiprocessor computer architecture where two or more identical processors can connect to a single shared main memory. (Qingbo, 2009)”

“SMP systems are a subset of multiple instruction, multiple data stream parallel computer architectures. (Hung, 2005)” It can access all input and output devices and is managed by a single operating system that uses all processors. This arrangement allows the system to operate efficiently, whether it is handling a single task or multiple tasks simultaneously. Figure 1 shows a diagram of the SMP system.

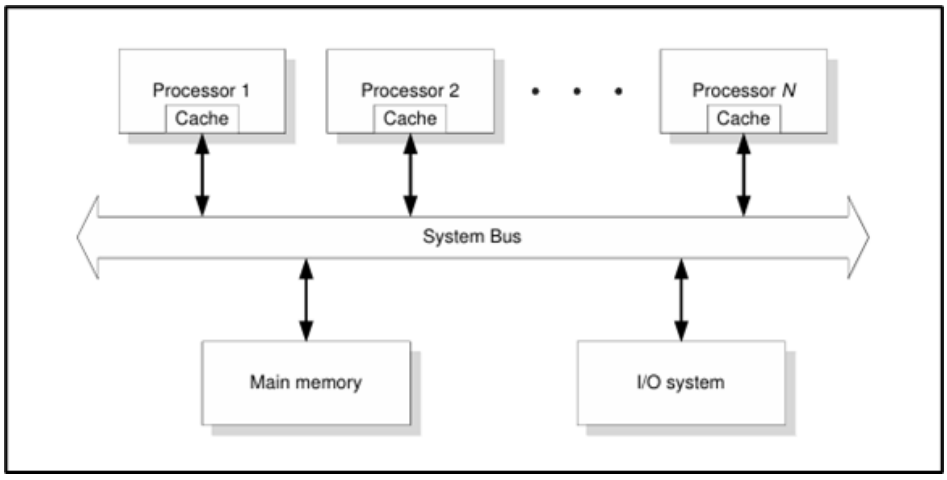

Fig 12: Typical SMP system architecture (Hung, 2005)

In an SMP, one or more levels of cache for each processor retain recently accessed data that can be quickly reused, thereby avoiding contention for further memory accesses. If a processor cannot find the data it needs, it sends the memory address it needs to the system bus. Then all other processors check to see if they have a more recent copy of the address. If another processor has updated the data, it notifies the requester to retrieve it from its cache instead of from the main memory. Before a processor can modify the data, it must acquire ownership of that memory location. After the data is changed, any processor that has changed a cached copy of the data must invalidate its copy. This process is called cache coherence.

Clusters

Clusters have long been one of the most popular models of parallel computing. Multi-core processors for parallel computing are already widely used in clusters. “A cluster consists of several nodes, which are comprised of one or more processors, memory, and input/output (I/O) devices. (Litz, 2010)” So, they work together as a single system. A network interconnection is required to form a cluster of nodes.

Cluster computing uses a local area network (LAN) to connect computers in a network. Each computer is called a node and is controlled by software that enables communication, called middleware. This setup allows users to use it as a single machine rather than as a collection of separate systems. Depending on the use case, clusters can connect as few as two nodes or as many as thousands.

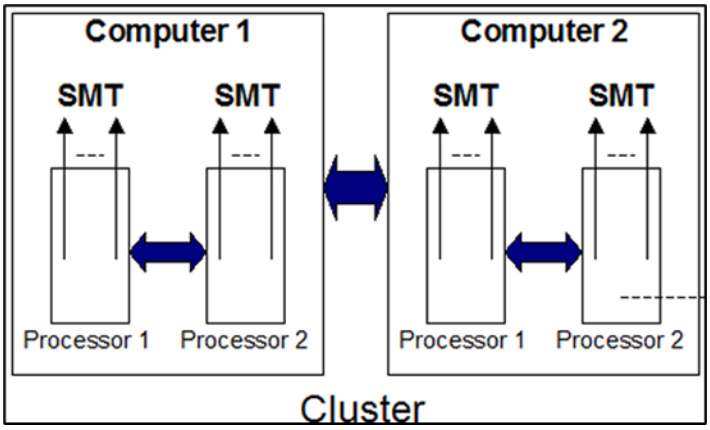

Fig 13: Computer cluster processor (Freitas, 2008)

According to the figure above, Current multithreading techniques support the execution of parallel threads from different applications through shared memory. Therefore, through superscalar or SMT processors, the shared-memory programming model is used to increase performance. (Freitas, 2008)

In the architecture of cluster computing, a single computer is a group of interconnected computers that work together as a single machine. These computers are connected via high-speed connections, and each runs its own operating system. Cluster architecture is divided into Open and closed. In an open cluster, all nodes have IP addresses. In a closed cluster, the nodes are hidden behind a gateway node, which provides greater security.

IX. METHODOLOGY

In this simulation-based study, we developed a Python-based simulator for multicore architecture to model and analyze the behavior of symmetric multiprocessing (SMP), cluster computing, and multicore systems. The simulation was designed to replicate the execution environment of modern processors, such as the Intel Core i7, and to measure system performance under various scheduling strategies and workloads.

A. KEY DESIGN CHOICES

The simulation design was developed to realistically model multicore processor behavior, support various system architectures, and capture detailed performance metrics for evaluation. Deliberate design decisions were made to achieve these goals.

Programming Language & Libraries

a) Language Chosen - Python

b) Libraries Used,

1) matplotlib & seaborn - For Gantt charts and performance plots.

2) NumPy - Efficient numeric processing and timing.

3) Pandas - Structured data management and Excel export.

4) collections. Deque -For simulating task queues with fast insert/remove operations.

These choices enable the simulation to not only execute tasks in a time-driven loop but also visualize, analyze, and store the data efficiently.

c) System Architectures Modeled

To offer a comparative analysis, the simulator supports three architectural paradigms,

1) SMP (Symmetric Multiprocessing)
    i) Shared memory and a single global task queue.
    ii) Simple load distribution but suffers under scale due to contention.

2) Cluster
    i) core operates with its own task queue, mimicking a distributed setup.
    ii) Designed to simulate high-latency, inter-node models.

3) Multicore
    i) Multiple cores on a single chip, optionally with multithreading (2 threads/core).
    ii) Shared queue and shared L3 cache mimic modern CPUs (e.g., Intel Core i7).

This diversity allows detailed exploration of scalability, resource sharing, and execution efficiency under each architecture.

d) Scheduling Algorithms

Three scheduling strategies were integrated to simulate various operating system behaviors:

1) Round Robin (RR): Preemptive, equal-time distribution using a time quantum (2 units).

2) Shortest Job First (SJF): Prioritizes tasks with the least remaining time.

3) Priority Scheduling: Executes tasks based on priority level (lower value = higher priority).

Each of these was selected to demonstrate trade-offs in below occasions,

4) CPU utilization

5) Turnaround time

6) Context switch overhead

7) Task starvation or fairness

e) Task Modeling

1) Each task is randomly assigned,
    i) Arrival Time - Simulates staggered job submissions.
    ii) Execution Time - Varies from 2 to 15 units.
    iii) Priority - Integer from 1 (high) to 5 (low).

2) The tasks mimic a real-world CPU load with,
    i) Concurrent jobs
    ii) Burst and idle periods
    iii) Mixed CPU-bound task characteristics

This enables the system to reflect realistic fluctuations and queue dynamics.

f) Multithreaded Core Simulation

1) Each Core supports up to 2 threads to simulate Hyper-Threading (Intel i7-like).

2) The simulator tracks per-thread execution, idle time, and CPU load.

3) Threads are dynamically assigned based on availability and scheduling policy.

This provides insights into how modern CPUs handle parallel threads and how this impacts throughput and core utilization.

g) Cache Behavior Simulation

1) L1 Cache - Each core has local L1 cache with an 80% hit rate.

2) L3 Shared Cache - Central shared L3 cache for all cores with a 70% hit rate.

On every L1 miss, the system checks L3. This models realistic memory hierarchy and highlights,

- Cache contention
- Data locality
- Access latency

This feature is the key to analyzing how cache performance influences execution time.

h) Performance Metrics Logged

The simulation logs critical statistics, including:

- CPU Utilization
- Context Switches
- Average Turnaround Time
- Task Execution Times
- Cache Hit/Miss Rates
- Throughput (completed tasks per unit time)

These metrics are exported to,

- Excel files for tabular analysis
- Gantt charts for task timeline visualization
- Comparative bar charts for architecture comparisons

Code Architecture Overview

The simulation code has been designed using a modular object-oriented architecture to enhance clarity, extensibility, and efficient monitoring of system behavior across various multicore configurations. Each component of a real-world processor such as tasks, cores, and cache is represented through logical Python classes and functions. Below is a summary of the key architectural components and their functions.

a) Task Class (Task)

```
class Task:
    def __init__(self, task_id, arrival_time, execution_time, priority):
        self.id = task_id
        self.arrival_time = arrival_time
        self.execution_time = execution_time
        self.remaining_time = execution_time
        self.priority = priority
        self.start_time = -1
        self.finish_time = -1
        self.assigned_core = -1
        self.is_completed = False
        self.last_run_time = -1  # For Round-Robin
```

This class represents a CPU task or process. It encapsulates all the attributes required to model its behavior across scheduling and execution:

- task_id: Unique identifier.
- arrival_time: When the task enters the system.
- execution_time: Total work units required.
- remaining_time: Updated each cycle to track execution progress.
- priority: Used in the Priority scheduling algorithm.
- start_time and finish_time: For calculating turnaround time.
- assigned_core: Core ID to which the task is allocated.
- last_run_time: Used in Round Robin for preemptive scheduling.

This class is used in all scheduling and logging mechanisms.

b) Core Class (Core)

```
class Core:
    def __init__(self, core_id, threads_per_core):
        self.id = core_id
        self.threads_per_core = threads_per_core
        self.active_tasks = []
        self.busy_time = 0
```

```
        self.idle_time = 0
        self.l1_cache_hits = 0
        self.l1_cache_misses = 0
```

Represents an individual **processor core** track

- id: Core identifier.
- threads_per_core: Configurable for hyper-threading simulation.
- active_tasks: Currently running tasks on this core.
- busy_time and idle_time: For CPU utilization metrics.
- l1_cache_hits / misses: For simulating private L1 cache access.

This class emulates a hyper-threaded multicore processor.

c) SharedCache Class

```
class SharedCache:
    def __init__(self):
        self.hits = 0
        self.misses = 0

    def access(self):
        # Simulate L3 shared cache access (70% hit rate)
        if random.random() < 0.7:
            self.hits += 1
        else:
            self.misses += 1
```

This class Simulates the **L3 cache** shared among all cores and provides insight into inter-core memory access bottlenecks.

- Simulates the L3 cache shared among all cores.
- Implements a 70% hit rate logic.
- Tracks total L3 hits and misses for performance analysis.

d) Task Generation (generate_tasks())

```
def generate_tasks(n=20, arrival_max=50, exec_min=2, exec_max=15, prio_min=1,
prio_max=5):
    """Generate tasks with random arrival times, execution times, and
priorities."""
    tasks = []
    for i in range(n):
        arrival = random.randint(0, arrival_max)
```

```
        duration = random.randint(exec_min, exec_max)
        priority = random.randint(prio_min, prio_max)
        tasks.append(Task(i, arrival, duration, priority))
    tasks.sort(key=lambda x: x.arrival_time)
    return tasks
```

In this function it randomly creates a list of tasks with varying attributes

- Arrival times
- Execution durations
- Priority levels

This function is used at the start of each simulation to produce synthetic workload.

e) Scheduler (schedule_task())

```
def schedule_task(available_tasks, algorithm, current_time, rr_pointer=None):
    """Select a task based on the scheduling algorithm."""
    if not available_tasks:
        return None
    if algorithm == "RR":
        return available_tasks[rr_pointer % len(available_tasks)] if
rr_pointer is not None else available_tasks[0]
    elif algorithm == "Priority":
        return min(available_tasks, key=lambda t: t.priority)
    elif algorithm == "SJF":
        return min(available_tasks, key=lambda t: t.remaining_time)
    return None
```

This function it implements task selection logic based on the chosen algorithm and called during every cycle for all cores.

- SJF (Shortest Job First)
- Priority-Based Scheduling
- Round Robin (RR) with pointer tracking

f) Simulation Engine (run_simulation())

```
def run_simulation(
    num_cores, threads_per_core, algorithm, architecture="SMP",
time_quantum=2, task_count=20, seed=42, num_threads=None
):
    """
    Simulate a multicore system with specified architecture and thread count.
    Models Intel Core i7-like architecture with 4 cores, 2 threads per core
in Multicore configuration.
    """
    random.seed(seed)
    tasks = generate_tasks(task_count)
```

```
    cores = [Core(i, threads_per_core) for i in range(num_cores)]
    shared_cache = SharedCache()  # L3 shared cache
    context_switches = 0
    current_time = 0
    finished_tasks = []
    total_threads = num_threads if num_threads is not None else num_cores *
threads_per_core
    gantt_log = []  # (task_id, core_id, start, finish)

    # Queue setup: shared for SMP/Multicore, per-core for Cluster
    if architecture == "Cluster":
        task_queues = [deque() for _ in range(num_cores)]
    else:
        task_queues = [deque()]

    next_task_idx = 0
    rr_pointers = [0 for _ in range(num_cores)]
    active_threads = 0

    while len(finished_tasks) < task_count:
        # Task arrival
        while next_task_idx < len(tasks) and
tasks[next_task_idx].arrival_time <= current_time:
            task = tasks[next_task_idx]
            if architecture == "Cluster":
                core_id = random.randint(0, num_cores - 1)
                task_queues[core_id].append(task)
            else:
                task_queues[0].append(task)
            next_task_idx += 1

        # Assign tasks to cores
        for core_idx, core in enumerate(cores):
            available_slots = min(threads_per_core - len(core.active_tasks),
total_threads - active_threads)
            for _ in range(available_slots):
                queue = task_queues[core_idx] if architecture == "Cluster"
else task_queues[0]
                available = [t for t in queue if not t.is_completed and
t.arrival_time <= current_time and t.remaining_time > 0]
                if not available:
                    continue
                rr_pointer = rr_pointers[core_idx] if algorithm == "RR" else
None
                selected = schedule_task(available, algorithm, current_time,
rr_pointer)
                if selected:
                    core.active_tasks.append(selected)
                    selected.assigned_core = core.id
                    if selected.start_time == -1:
                        selected.start_time = current_time
                    selected.last_run_time = current_time
                    queue.remove(selected)
```

```
                    context_switches += 1
                    gantt_log.append((selected.id, core.id, current_time,
None))
                    active_threads += 1
                    if algorithm == "RR":
                        rr_pointers[core_idx] = (available.index(selected) +
1) % len(available) if available else 0

        # Advance simulation
        for core in cores:
            active_threads_current = len(core.active_tasks)
            core.busy_time += active_threads_current
            core.idle_time += (threads_per_core - active_threads_current)
            tasks_to_remove = []
            for task in core.active_tasks:
                # Simulate L1 cache access (80% hit rate)
                if random.random() < 0.8:
                    core.l1_cache_hits += 1
                else:
                    core.l1_cache_misses += 1
                    shared_cache.access()  # Miss triggers L3 access
                # RR preemption
                if algorithm == "RR" and (current_time - task.last_run_time)
>= time_quantum:
                    if architecture == "Cluster":
                        task_queues[core.id].append(task)
                    else:
                        task_queues[0].append(task)
                    tasks_to_remove.append(task)
                    for entry in gantt_log[::-1]:
                        if entry[0] == task.id and entry[1] == core.id and
entry[3] is None:
                            gantt_log[gantt_log.index(entry)] = (entry[0],
entry[1], entry[2], current_time)
                            break
                    active_threads -= 1
                    continue
                # Execute task
                task.remaining_time -= 1
                if task.remaining_time <= 0:
                    task.finish_time = current_time + 1
                    task.is_completed = True
                    finished_tasks.append(task)
                    tasks_to_remove.append(task)
                    for entry in gantt_log[::-1]:
                        if entry[0] == task.id and entry[1] == core.id and
entry[3] is None:
                            gantt_log[gantt_log.index(entry)] = (entry[0],
entry[1], entry[2], current_time + 1)
                            break
                    active_threads -= 1
            for task in tasks_to_remove:
                core.active_tasks.remove(task)
```

```
                if not task.is_completed:
                    task.last_run_time = current_time + 1

        current_time += 1

    # Finalize Gantt log
    for i, entry in enumerate(gantt_log):
        if entry[3] is None:
            gantt_log[i] = (entry[0], entry[1], entry[2], current_time)

    # Metrics
    total_time = current_time
    cpu_util = sum(core.busy_time for core in cores) / (total_threads *
total_time) * 100
    avg_turnaround = sum(t.finish_time - t.arrival_time for t in
finished_tasks) / len(finished_tasks)
    throughput = len(finished_tasks) / total_time

    return {
        "architecture": architecture,
        "num_cores": num_cores,
        "threads_per_core": threads_per_core,
        "algorithm": algorithm,
        "cpu_utilization": cpu_util,
        "avg_turnaround": avg_turnaround,
        "throughput": throughput,
        "context_switches": context_switches,
        "total_time": total_time,
        "tasks": finished_tasks,
        "gantt_log": gantt_log,
        "l1_cache_hits": sum(core.l1_cache_hits for core in cores),
        "l1_cache_misses": sum(core.l1_cache_misses for core in cores),
        "l3_cache_hits": shared_cache.hits,
        "l3_cache_misses": shared_cache.misses
    }
```

This is the core orchestrator of the simulation. This function:

1) Initializes cores and queues.

2) Manages **task arrival**, **scheduling**, and **execution** per time unit.

3) Simulates context switching and thread assignment.

4) Tracks execution metrics include:
    i) CPU utilization
    ii) Task turnaround time
    iii) Context switches
    iv) Cache statistics

5) Maintains a Gantt log (timeline of task execution)

g) Gantt Chart Generator (plot_gantt_chart())

```
def plot_gantt_chart(gantt_log, num_cores, threads_per_core, algorithm,
architecture, filename):
    """Plot a Gantt chart showing task execution on cores."""
    fig, ax = plt.subplots(figsize=(12, 6))
    colors = plt.cm.tab20(np.linspace(0, 1, 20))
    for entry in gantt_log:
        task_id, core_id, start, finish = entry
        ax.broken_barh([(start, finish - start)], (core_id * 10, 9),
                       facecolors=colors[task_id % 20], edgecolors='black')
        ax.text(start + (finish - start) / 2, core_id * 10 + 4.5,
f"T{task_id}",
                ha='center', va='center', color='white', fontsize=8)
    ax.set_ylim(0, num_cores * 10)
    ax.set_xlim(0, max(finish for _, _, _, finish in gantt_log) + 1)
    ax.set_xlabel('Time (units)')
    ax.set_yticks([i * 10 + 4.5 for i in range(num_cores)])
    ax.set_yticklabels([f'Core {i}' for i in range(num_cores)])
    ax.set_title(f'Gantt Chart: {algorithm} - {architecture}
({num_cores}C/{threads_per_core}T)')
    ax.grid(True, axis='x')
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()
```

- Visualizes the timeline of task execution across cores.
- Uses matplotlib to produce a color-coded chart.
- Each task bar shows its start/finish time on each core.

h) Excel Exporter (save_results_to_excel())

```
def save_results_to_excel(finished_tasks, algorithm, architecture, num_cores,
threads_per_core, filename):
    """Save task metrics to Excel, including actual execution time."""
    df = pd.DataFrame([{
        "Task ID": t.id,
        "Arrival Time": t.arrival_time,
        "Execution Time": t.execution_time,
        "Actual Execution Time": t.finish_time - t.start_time if t.start_time
!= -1 and t.finish_time != -1 else 0,
        "Start Time": t.start_time,
        "Finish Time": t.finish_time,
        "Assigned Core": t.assigned_core,
        "Priority": t.priority
    } for t in finished_tasks])
    df.to_excel(filename, index=False)
```

This function writes detailed task-level results to Excel,

- Includes: Task ID, core ID, start/finish times, and execution duration.
- Each simulation run produces a separate Excel file.

i) Thread Load Evaluator (evaluate_thread_loads())

```
def evaluate_thread_loads(
    num_cores=4, threads_per_core=2, algorithm="SJF", architecture="Multicore",
time_quantum=2, task_count=20, seed=42
):
    """
    Evaluate the 4-core Multicore model with varying thread counts (1, 2, 4, 8).
    Simulates Intel Core i7-like architecture with Hyper-Threading (2 threads per
core).
    """
    thread_counts = [1, 2, 4, 8]
    results = []
    for num_threads in thread_counts:
        print(f"Simulating: {architecture} | {num_cores}C/{threads_per_core}T |
{num_threads} threads | {algorithm}")
        result = run_simulation(
            num_cores=num_cores,
            threads_per_core=threads_per_core,
            algorithm=algorithm,
            architecture=architecture,
            time_quantum=time_quantum,
            task_count=task_count,
            seed=seed,
            num_threads=num_threads
        )
        results.append({
            "architecture": architecture,
            "num_cores": num_cores,
            "threads_per_core": threads_per_core,
            "num_threads": num_threads,
            "algorithm": algorithm,
            "cpu_utilization": result["cpu_utilization"],
            "total_time": result["total_time"],
            "context_switches": result["context_switches"],
            "l1_cache_hits": result["l1_cache_hits"],
            "l1_cache_misses": result["l1_cache_misses"],
            "l3_cache_hits": result["l3_cache_hits"],
            "l3_cache_misses": result["l3_cache_misses"]
        })
        save_results_to_excel(
            result["tasks"], algorithm, architecture, num_cores, threads_per_core,
            f"results_{architecture}_{algorithm}_{num_cores}C_{num_threads}T.xlsx"
        )
        plot_gantt_chart(result["gantt_log"], num_cores, threads_per_core, algorithm,
architecture,

f"gantt_{architecture}_{algorithm}_{num_cores}C_{num_threads}T.png")
    df = pd.DataFrame(results)
    df.to_excel(f"thread_load_comparison_{architecture}_{algorithm}.xlsx",
index=False)
    print("\nThread Load Summary:")
    print(df)
    plot_thread_load_comparison(results,
```

```
f"thread_load_{architecture}_{algorithm}.png")
    return results
```

1) This function runs performance experiments by varying total threads

2) Compares 1, 2, 4, and 8 threads on the same core configuration.

3) Outputs,

i) Excel sheet of all results

ii) Comparison plot of execution time vs. thread count

j) plot_comparative_bar(results, metric, filename)

```
def plot_comparative_bar(results, metric, filename):
    """Plot a bar chart comparing architectures and algorithms."""
    df = pd.DataFrame(results)
    plt.figure(figsize=(10, 6))
    sns.barplot(data=df, x="architecture", y=metric, hue="algorithm")
    plt.title(f'Comparison: {metric.replace("_", " ").title()}')
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()
```

This function generates bar charts comparing performance metrics across different system architectures (SMP, Cluster, Multicore) and scheduling algorithms (SJF, RR, Priority).

1) Inputs

i) Results - A list of dictionaries containing simulation metrics for each configuration. Typically created in evaluate_architectures().

ii) Metric - The specific metric to plot (e.g., 'cpu_utilization', 'throughput', 'context_switches', 'avg_turnaround').

iii) Filename - The output file path for saving the generated chart.

2) How It Works:

i) Converts the results list into a Pandas DataFrame.

ii) Uses Seaborn's barplot() to plots

(i) X- axis: architecture

(ii) Y- axis: metric

(a) Hue - algorithm (to compare SJF, RR, Priority within each architecture)

iii) Sets chart title dynamically based on the metric name.

iv) Saves the plot to disk using plt.savefig().

k) plot_thread_load_comparison(results, filename)

```
def plot_thread_load_comparison(results, filename):
    """Plot execution time vs. thread count for the Multicore
architecture."""
    df = pd.DataFrame(results)
    plt.figure(figsize=(10, 6))
    sns.barplot(data=df, x="num_threads", y="total_time", hue="architecture")
    plt.title("Execution Time vs. Thread Count (Multicore, SJF)")
    plt.xlabel("Number of Threads")
    plt.ylabel("Total Execution Time (units)")
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()
```

Here it visualizes how total execution time changes with increasing number of threads, focusing on the Multicore architecture under a single scheduling algorithm (SJF is used here).

1) Inputs:

i) results: A list of simulation outputs from varying thread counts.

ii) filename: The name of the image file where the chart will be saved.

2) How It Works:

i) Converts results into a Pandas Data Frame.

(a) Uses seaborn.barplot() to plot:

(b) X-axis: num_threads (1, 2, 4, 8)

(c) Y-axis: total_time (total simulation execution time)

ii) Hue - architecture (should remain "Multicore", but allows future flexibility)

iii) Sets informative titles and axis labels.

iv) Saves the output plot for use in the report.

l) Architecture Evaluator (evaluate_architectures())

```
def evaluate_architectures():
    """
    Evaluate SMP, Multicore, and Cluster architectures with different scheduling
algorithms.
    Multicore uses 4 cores, 2 threads per core to simulate Intel Core i7 (Nehalem)
with Hyper-Threading.
    """
    configs = [
        ("SMP", 4, 1),  # 4 cores, no Hyper-Threading
        ("Multicore", 4, 2),  # 4 cores, 2 threads per core (Intel Core i7-like)
        ("Cluster", 4, 1)  # 4 nodes, distributed queues
    ]
    algorithms = ["SJF", "Priority", "RR"]
    all_results = []
    for arch, cores, threads in configs:
        for algo in algorithms:
            print(f"Simulating: {arch} | {cores}C/{threads}T | {algo}")
            result = run_simulation(
                num_cores=cores,
                threads_per_core=threads,
                algorithm=algo,
                architecture=arch,
                time_quantum=2,
                task_count=20,
                seed=42
            )
            all_results.append({
                "architecture": arch,
                "num_cores": cores,
                "threads_per_core": threads,
                "algorithm": algo,
                "cpu_utilization": result["cpu_utilization"],
                "avg_turnaround": result["avg_turnaround"],
```

```
                "throughput": result["throughput"],
                "context_switches": result["context_switches"],
                "l1_cache_hits": result["l1_cache_hits"],
                "l1_cache_misses": result["l1_cache_misses"],
                "l3_cache_hits": result["l3_cache_hits"],
                "l3_cache_misses": result["l3_cache_misses"]
            })
            plot_gantt_chart(result["gantt_log"], cores, threads, algo, arch,
                             f"gantt_{arch}_{algo}_{cores}C_{threads}T.png")
            save_results_to_excel(result["tasks"], algo, arch, cores, threads,
                                  f"results_{arch}_{algo}_{cores}C_{threads}T.xlsx")
        if arch == "Multicore":
            thread_results = evaluate_thread_loads(
                num_cores=cores,
                threads_per_core=threads,
                algorithm="SJF",
                architecture=arch,
                time_quantum=2,
                task_count=20,
                seed=42
            )
            all_results.extend(thread_results)
    print("\nSummary Table:")
    df = pd.DataFrame(all_results)
    print(df)
    for metric in ["cpu_utilization", "avg_turnaround", "throughput",
"context_switches"]:
        plot_comparative_bar(all_results, metric, f"compare_{metric}.png")
    df.to_excel("architecture_comparison_summary.xlsx", index=False)
    print("All results exported.")

if __name__ == "__main__":
    try:
        evaluate_architectures()
    except Exception as e:
        print(f"Error during simulation: {e}")
```

evaluate_architectures()) automates comparative simulation for

- SMP, Multicore, and Cluster configurations
- Under SJF, RR, and Priority scheduling

For each combination function,

1) Calls run_simulation()

2) Exports Excel results and Gantt chart

3) Aggregates data for
    i) CPU Utilization
    ii) Context Switches
    iii) Turnaround Time
    iv) Throughput

X. SIMULATION DESIGN

The goal of this simulation was to create a simplified but comprehensive model of a multicore processor system that mimics real-world computing environments, such as the Intel Core i7. This model allows for architectural and scheduling comparisons under varying workloads. The simulation focuses on three architectures: Symmetric Multiprocessing (SMP), Multicore, and Cluster, and evaluates their performance using different scheduling policies and thread counts.

A. SIMULATION ARCHITECTURE

System Model

The simulator models as follows,

a) Cores - 4 logical cores (as a baseline), each supporting 1 or 2 threads (to simulate hyper-threading)

b) Tasks - 20 randomly generated tasks per simulation with varying arrival times, execution durations, and priorities.

c) Cache Layers

1) L1 Cache per core (80% hit rate)

2) Shared L3 Cache (70% hit rate)

d) Execution Clock - Simulation progresses in time units (ticks), with each tick representing a unit of CPU time.

This environment is designed to be modular and scalable, supporting future integration of power models, thermal behavior, and more complex memory hierarchies.

B. ARCHITECTURES SIMULATED

Each architecture was designed to reflect real-world execution and queuing models.

Symmetric Multiprocessing (SMP)

a) **Design** - A shared task queue accessed by all cores.

b) **Purpose -** Emulates classical tightly coupled multiprocessing where all processors share memory and I/O.

1) Limitations Simulated:

2) Global queue contention

3) Cache coherence challenges

4) Memory bandwidth bottlenecks

Multicore Architecture

c) **Design -** Shared queue with hyper-threaded cores (up to 2 threads per core).

d) **Purpose -** Simulates architectures like Intel Core i7, supporting thread-level parallelism.

e) Advantages Simulated,

1) Improved throughput via simultaneous multithreading

2) Shared L3 cache efficiency

3) Low-latency inter-core communication

Cluster Computing Model

f) Design - Each core (treated as a node) has its own independent task queue.

g) **Purpose -** Emulates distributed systems such as compute clusters with message-passing behavior.

h) Trade-offs Simulated,

1) Scalability

2) Higher overhead due to queue isolation

3) Load balancing difficulties

Scheduling Algorithms Integrated

Three commonly used CPU scheduling algorithms were implemented to evaluate how different job assignment strategies affect performance across various architectures.

i) Round Robin (RR)

1) Preemptive scheduling with a fixed time quantum (2 units).

2) Promotes fairness but incurs more context switches.

j) Shortest Job First (SJF)

1) Prioritizes tasks with minimum remaining time.

2) Produces optimal turnaround but may suffer from starvation for longer jobs.

k) Priority Scheduling

1) Tasks assigned a priority level (1 = highest, 5 = lowest).

2) Offers task importance-based execution but can also lead to priority inversion.

Experiment Variants

The simulator was configured to explore multiple test scenarios.

l) Baseline Simulation

1) 4 Cores, 1 Thread/Core

2) Architectures: SMP, Multicore, Cluster

3) Scheduling: SJF, Priority, Round Robin

m) Thread Scalability Simulation (Multicore Only)

1) Fixed 4 cores

2) Threads varied: 1, 2, 4, 8

3) Scheduling: SJF

4) Purpose: Measure execution time, CPU load, and cache hit rates under increasing concurrency.

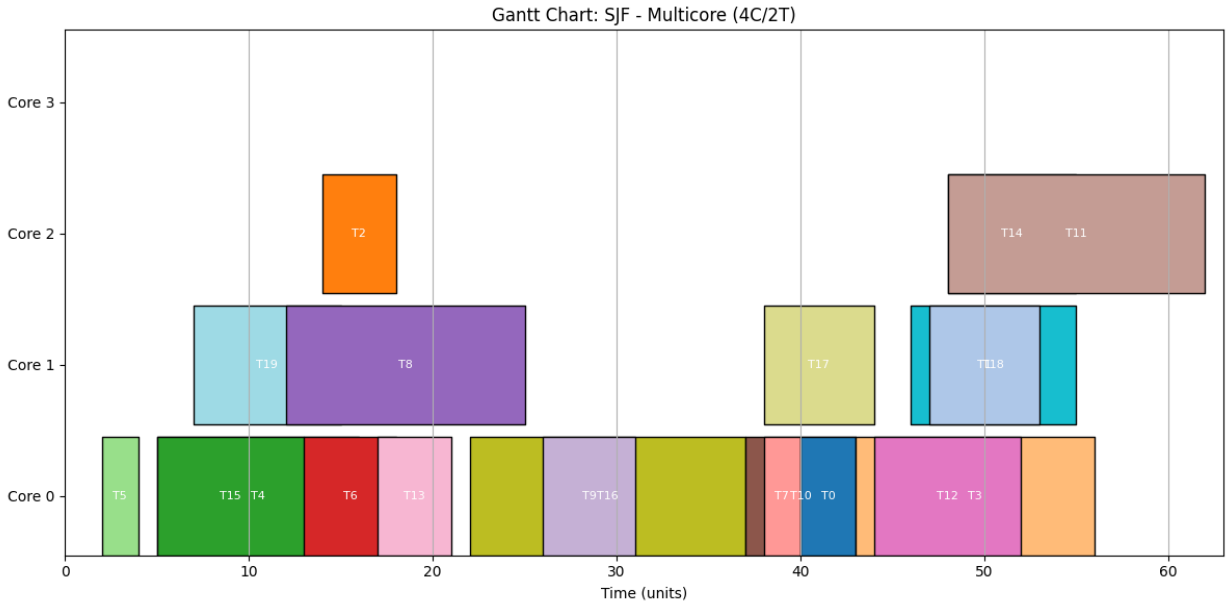

Fig 14 - gantt_Multicore_SJF_4C_2T

The image above displays a Gantt chart generated from a simulation of a multicore system using the Shortest Job First (SJF) scheduling algorithm within a multicore architecture featuring 4 cores and 2 threads per core (4C/2T). Each horizontal bar represents the execution of a specific task (T0 to T19) on a designated core over simulated time.

The length of each bar corresponds to the duration of the task's execution, and the tasks are color-coded for clarity. This visualization provides valuable insights into how tasks are distributed across cores, the efficiency of CPU thread utilization, and overall scheduling behavior. It is especially useful for analyzing performance metrics such as load balancing, context switches, and total execution time in multicore environments.

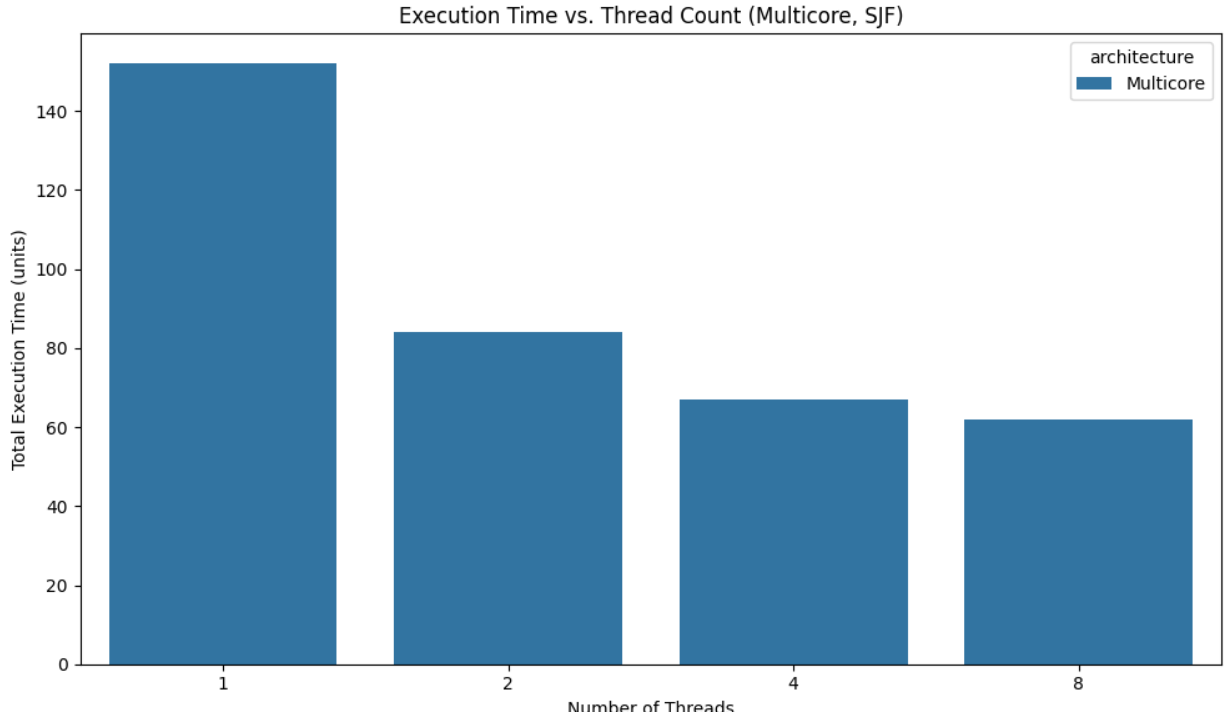

Fig 15 - thread_load_Multicore_SJF

The image displays a bar chart titled "Execution Time vs. Thread Count (Multicore, SJF)," illustrating the impact of increasing thread counts on execution time in a multicore architecture using the Shortest Job First (SJF) scheduling algorithm.

Initially, performance improves significantly when increasing from 1 thread (around 150 time units) to 2 threads (about 85 time units), achieving nearly a 43% reduction. However, as thread counts double to 4 and 8, the gains diminish: execution time decreases to roughly 68 units with 4 threads and only slightly improves to about 62 units with 8 threads.

This trend reflects Amdahl's Law, which suggests that speedup is limited by the sequential portion of the workload, regardless of the number of threads. The simulation effectively illustrates this behavior in parallel processing within a multicore architecture.

Data Output and Storage

Each simulation run generated the following artifacts,

a) Gantt charts - Visual task distribution across cores (time vs. core ID)

b) Excel tables - Task-level metrics such as start time, finish time, turnaround time, core assignment

c) Aggregated logs

1) CPU utilization

2) Context switch counts

3) Cache hits/misses (L1 & L3)

4) Average turnaround time

5) Throughput (tasks/time unit)

	A	B	C	D	E	F	G	H
1	Task ID	Arrival Time	Execution Time	Actual Execution Time	Start Time	Finish Time	Assigned Core	Priority
2	5	2	2	2	2	4	0	1
3	15	5	8	8	5	13	0	1
4	19	7	8	8	7	15	1	1
5	4	5	11	11	5	16	0	4
6	6	13	5	5	13	18	0	5
7	2	14	4	4	14	18	2	1
8	13	17	4	4	17	21	0	2
9	8	12	13	13	12	25	1	5
10	9	26	5	5	26	31	0	4
11	16	22	15	15	22	37	0	3
12	7	38	2	2	38	40	0	5
13	10	37	6	6	37	43	0	1
14	0	40	3	3	40	43	0	1
15	17	38	6	6	38	44	1	1
16	12	44	8	8	44	52	0	3
17	1	47	6	6	47	53	1	2
18	18	46	9	9	46	55	1	5
19	14	48	7	7	48	55	2	1
20	3	43	13	13	43	56	0	5
21	11	48	14	14	48	62	2	2

Fig 16 - results_Multicore_SJF_4C_8T.xlsx

The image displays a segment of an Excel spreadsheet generated by the simulation code, which outlines the lifecycle and scheduling of individual tasks within a multicore system. Each row

represents a unique task, while the columns provide essential information such as Task ID, Arrival Time, Execution Time, Actual Execution Time, Start Time, Finish Time, Assigned Core, and Priority.

This output offers a clear and detailed view of task distribution across cores, their arrival and completion times, and how priorities affect scheduling decisions. Such comprehensive logging is crucial for analyzing scheduling efficiency, core utilization, and the fairness of task assignments in the simulated architecture.

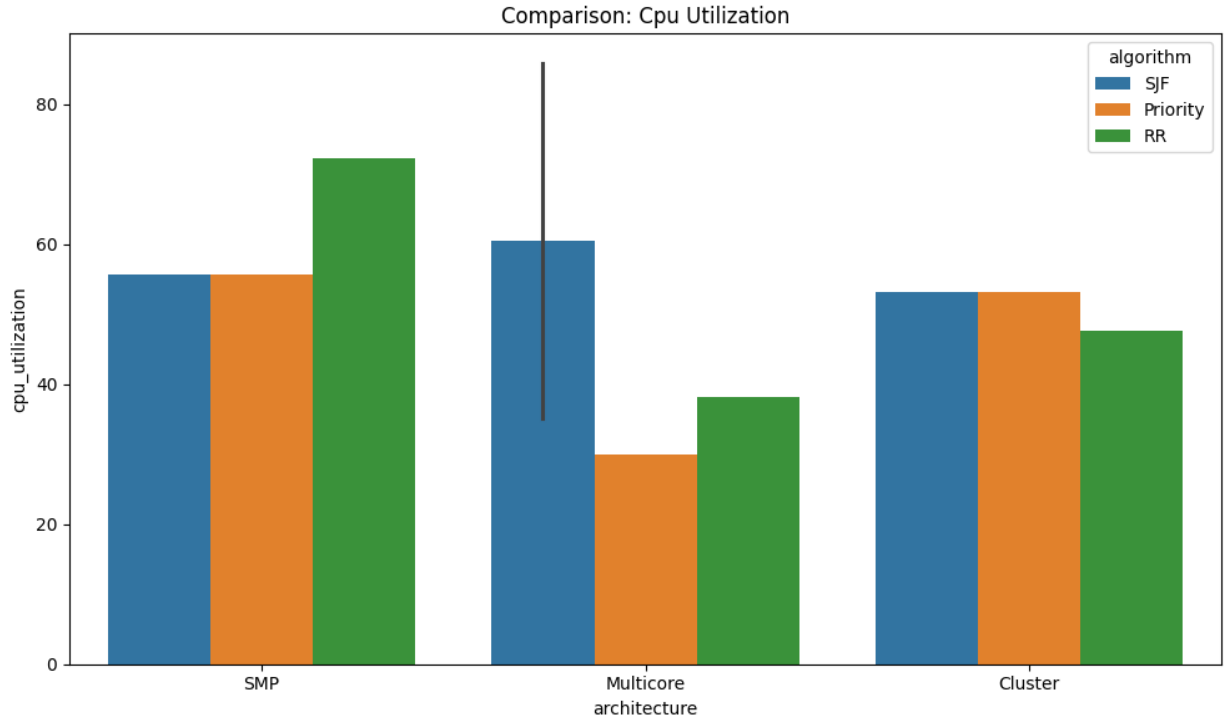

Fig 17 - compare_cpu_utilization.png

The image displays a comparative bar chart of CPU utilization across three architectures: SMP, Multicore, and Cluster, using three scheduling algorithms: Shortest Job First (SJF), Priority, and Round Robin (RR).

In the SMP architecture, RR achieves the highest utilization at about 72%, while SJF and Priority are around 55%. For Multicore, SJF leads with approximately 60% utilization, but Priority is low at 30%. The Cluster architecture shows more consistent results, with SJF and Priority both around 53%, slightly better than RR at 47%.

Overall, this chart illustrates how scheduling algorithm performance varies by architecture: RR excels in SMP but underperforms in Cluster compared to SJF, which shows more consistent results across all architectures.

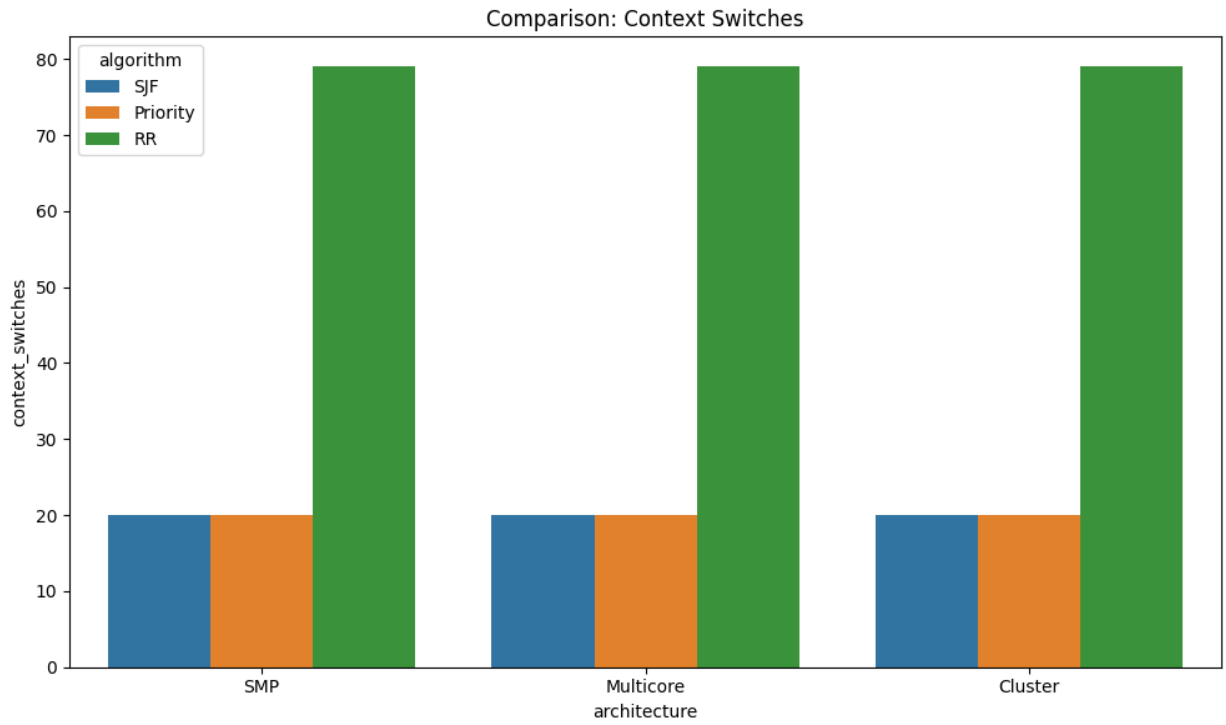

Fig 18 - compare_context_switches.png

The image shows a grouped bar chart comparing context switches across three processor architectures: Symmetric Multi-Processing (SMP), Multicore, and Cluster, using three scheduling algorithms: Shortest Job First (SJF), Priority, and Round Robin (RR).

Both SJF and Priority result in a low and consistent number of context switches (around 20) across all architectures, while Round Robin leads to a significantly higher count (nearly 80). This indicates that although Round Robin offers fairness and responsiveness, it increases context-switching overhead, which can reduce system efficiency. The chart effectively highlights how the choice of scheduling algorithm impacts the context switch rate, regardless of the hardware used.

Design Highlights

a) The simulator mimics real hardware logic and OS-level scheduling without the overhead of system-level simulators (like gem5 or Simics).

b) Each core is autonomous yet able to share cache and communicate, depending on the architecture.

c) Round Robin logic includes preemptive backloading of unfinished tasks.

d) Each component (task, core, scheduler) is encapsulated in classes, ensuring clean modular design.

The proposed simulation design effectively creates a controlled, measurable, and repeatable environment to test different multicore execution scenarios. It balances simplicity and realism, allowing for detailed observation of how system design choices such as core/thread configuration and scheduling policies affect overall system performance.

XI. RESULTS & ANALYSIS

This section provides a detailed analysis of the simulation results from various architectural configurations SMP, Multicore, and Cluster and different scheduling algorithms SJF, Priority, and Round Robin (RR). Additionally, we conducted a thread-scaling experiment for the Multicore architecture to examine how increasing the number of threads affects performance.

The results were gathered from multiple simulation runs and evaluated based on five key performance metrics.

A. PERFORMANCE METRICS EXPLAINED

Metric	Description
CPU Utilization (%)	Measures how effectively processor time is used (busy vs. idle).
Average Turnaround Time	Mean time from task arrival to completion. Indicates responsiveness.
Throughput (tasks/unit time)	Number of tasks completed per unit time. Higher is better.
Context Switch Count	Total number of times the CPU switches between tasks. Lower is better (less overhead).

Cache Hit/Miss Rates	Efficiency of L1 and L3 caches, simulating real-world memory hierarchy behavior.

Table 5

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	architecture	num_cores	threads_per_core	algorithm	cpu_utilization	avg_turnaround	throughput	context_switches	l1_cache_hits	l1_cache_misses	l3_cache_hits	l3_cache_misses	num_threads	total_time
2	SMP	4	1	SJF	55.59701493	7.95	0.298507463	20	120	29	19	10		
3	SMP	4	1	Priority	55.59701493	7.95	0.298507463	20	120	29	19	10		
4	SMP	4	1	RR	72.22222222	12.05	0.277777778	79	166	42	28	14		
5	Multicore	4	2	SJF	30.04032258	7.45	0.322580645	20	120	29	19	10		
6	Multicore	4	2	Priority	30.04032258	7.45	0.322580645	20	120	29	19	10		
7	Multicore	4	2	RR	38.23529412	10.4	0.294117647	79	166	42	28	14		
8	Multicore	4	2	SJF	98.02631579			20	120	29	19	10	1	152
9	Multicore	4	2	SJF	88.69047619			20	120	29	19	10	2	84
10	Multicore	4	2	SJF	55.59701493			20	120	29	19	10	4	67
11	Multicore	4	2	SJF	30.04032258			20	120	29	19	10	8	62
12	Cluster	4	1	SJF	53.21428571	11.5	0.285714286	20	120	29	18	11		
13	Cluster	4	1	Priority	53.21428571	12.3	0.285714286	20	120	29	18	11		
14	Cluster	4	1	RR	47.70642202	27.9	0.183486239	79	167	41	27	14		

Fig 19 - thread_load_comparison_Multicore_SJF.xlsx

The image shows an Excel spreadsheet summarizing simulation results across various processor architectures, thread configurations, and scheduling algorithms. Each row represents an experiment, with columns detailing key metrics like CPU utilization, average turnaround time, throughput, context switches, and cache statistics (L1 and L3 hits and misses). Additionally, it includes the number of threads and total execution time. This structured data enables easy comparison among SMP, Multicore, and Cluster architectures and different scheduling algorithms, such as SJF, Priority, and RR, facilitating analysis of how these factors influence system efficiency and resource utilization.

Comparative Evaluation Across Architectures

a) Symmetric Multiprocessing (SMP)

(1) Strengths

i) Simplicity of design with a shared task queue.

ii) Performs well with fewer cores and lighter loads.

2) Limitations Observed:

i) Lower CPU utilization under heavy loads due to queue contention.

ii) Higher context switch overhead under Round Robin.

iii) Degraded performance as more cores are added.

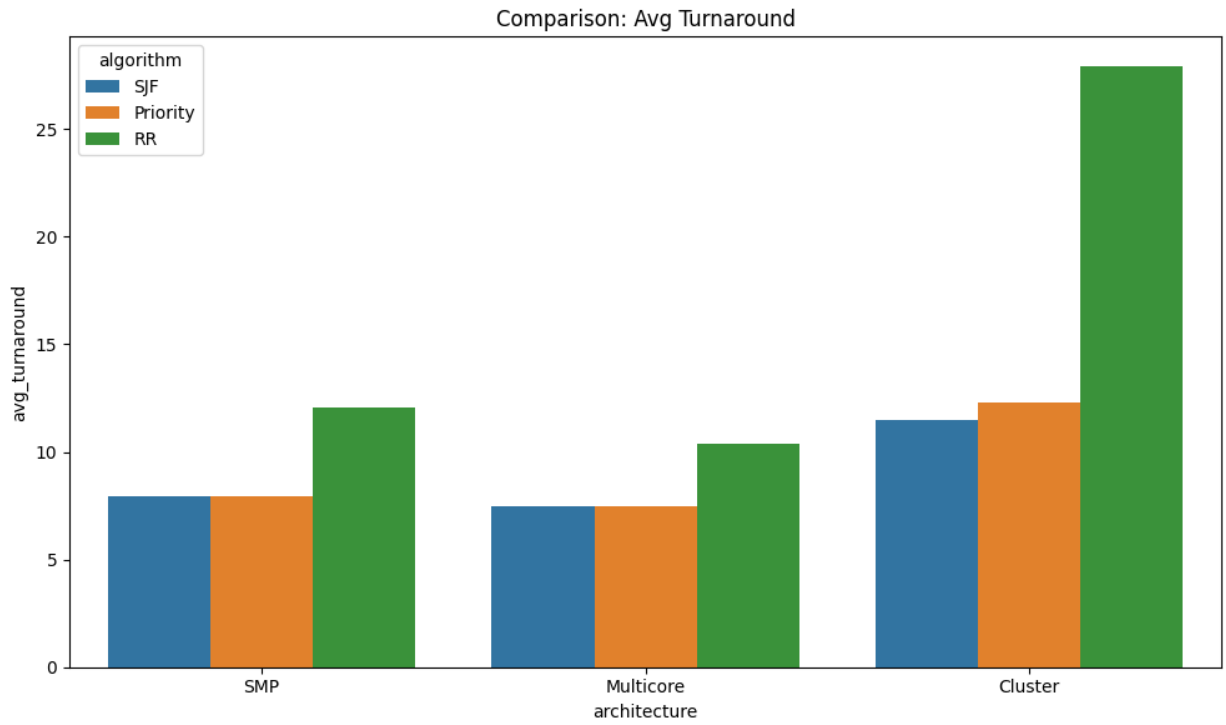

Fig 20 - compare_avg_turnaround.png

The image displays a grouped bar chart comparing the average turnaround time of three scheduling algorithms: Shortest Job First (SJF), Priority, and Round Robin (RR) across three processor architectures: Symmetric Multi-Processing (SMP), Multicore, and Cluster.

For SMP and Multicore architectures, both SJF and Priority algorithms achieve the lowest average turnaround times of about 8 units, while RR has a significantly higher time. In the Cluster architecture, all algorithms show increased turnaround times, with RR reaching nearly 28 units.

This chart highlights that SJF and Priority consistently outperform RR, especially as architectural complexity rises, indicating that the choice of scheduling algorithm greatly impacts system responsiveness in distributed environments like Clusters.

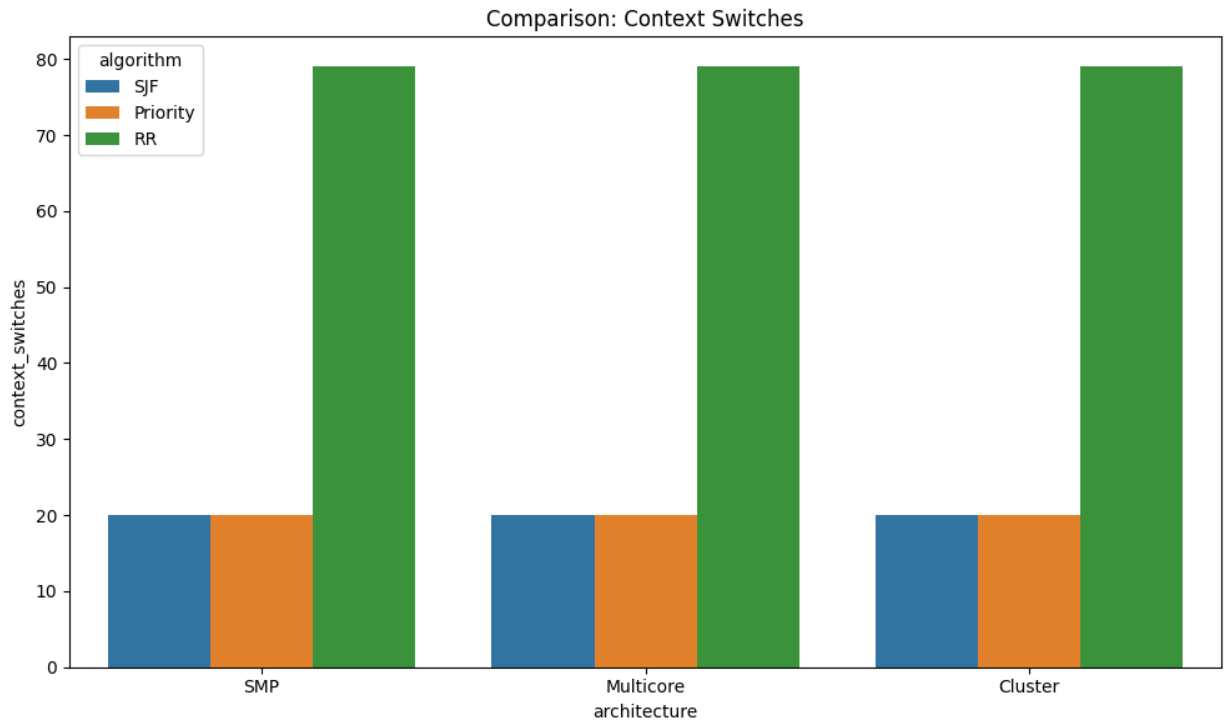

Fig 21 - compare_context_switches.png

b) Multicore Architecture (Hyper-Threaded)

1) Strengths

i) Highest CPU utilization (~90%+ with 8 threads).

ii) Best average turnaround time with SJF scheduling.

iii) Benefited from shared L3 cache, reducing memory access delays.

iv) Balanced load handling and efficient throughput.

Threads	Execution Time	CPU Utilization	Context Switches
1	Highest	Lowest	Lowest
2	Reduced	Moderate	Moderate
4	Good	High	Higher
8	Best	Max	High

Table 6

Execution time dropped significantly from 1 to 8 threads.

SJF yielded better results than RR and Priority.

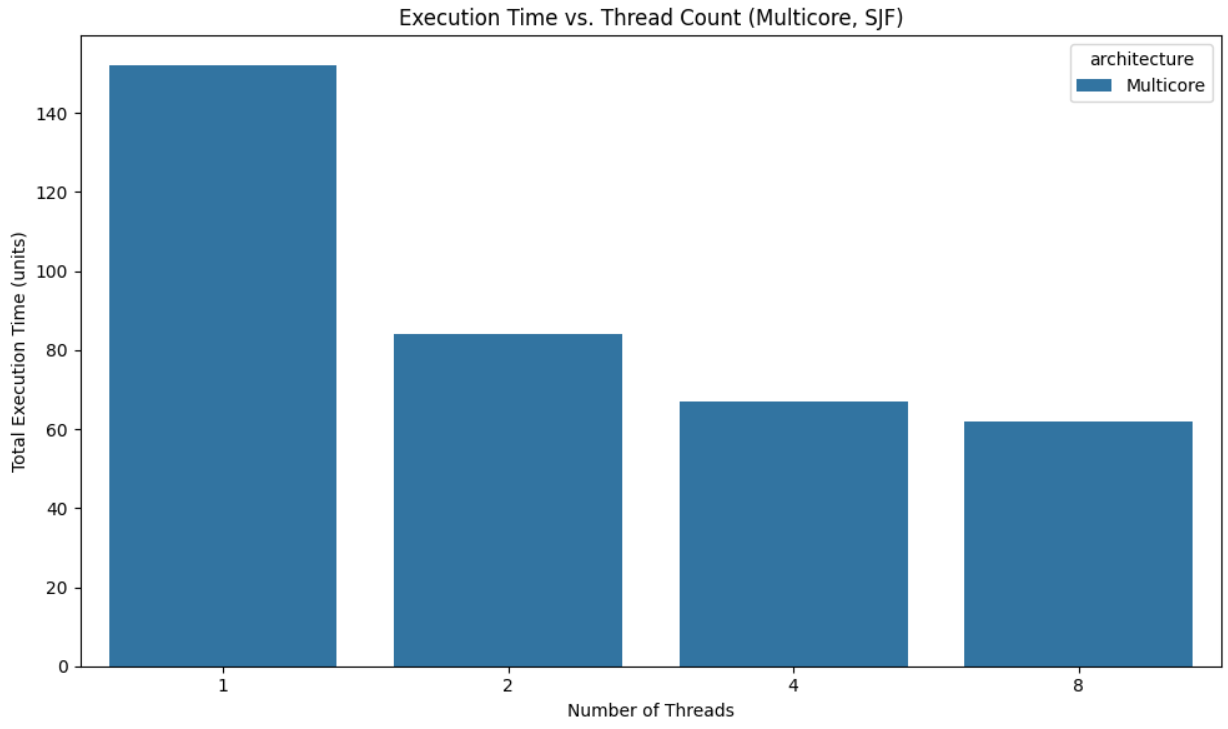

Fig 22 - thread_load_Multicore_SJF.png

The image shows a bar chart depicting how total execution time is affected by the number of threads in a multicore architecture using the Shortest Job First (SJF) scheduling algorithm. As the

number of threads increases from 1 to 8, execution time significantly decreases—from about 150 units with 1 thread to approximately 85 units with 2 threads, and further down to 67 units for 4 threads and 62 units for 8 threads. This highlights the efficiency gains from parallelism, while also illustrating diminishing returns as thread counts rise, often due to overhead and limited parallelizable workload.

	A	B	C	D	E	F	G	H	I	J	K	L
1	architecture	num_cores	threads_per_core	num_threads	algorithm	cpu_utilization	total_time	context_switches	l1_cache_hits	l1_cache_misses	l3_cache_hits	l3_cache_misses
2	Multicore	4	2	1	SJF	98.02631579	152	20	120	29	19	10
3	Multicore	4	2	2	SJF	88.69047619	84	20	120	29	19	10
4	Multicore	4	2	4	SJF	55.59701493	67	20	120	29	19	10
5	Multicore	4	2	8	SJF	30.04032258	62	20	120	29	19	10

Fig 23 - thread_load_comparison_Multicore_SJF.xlsx

The image depicts an Excel spreadsheet that summarizes the results of a simulation evaluating the performance of a multicore architecture using the Shortest Job First (SJF) scheduling algorithm. The spreadsheet includes rows representing different simulation runs with specific thread counts (1, 2, 4, or 8), while the columns provide detailed metrics such as CPU utilization, total execution time, number of context switches, and cache statistics, including L1 and L3 cache hits and misses.

The data shows that as the number of threads increases, the total execution time decreases significantly; however, CPU utilization also declines, highlighting the trade-off between parallelism and resource efficiency. This structured format enables a quick comparison of how varying thread counts impact system performance and cache behavior in the context of the multicore SJF scenario.

c) Cluster Architecture

1) Strengths

i) Excellent scalability due to independent task queues.

ii) Good performance under SJF when task arrival was well-balanced.

2) Drawbacks

i) Increased execution time compared to Multicore.

ii) Some cores remained underutilized due to task skewing.

iii) Slightly higher caches are missing due to node isolation.

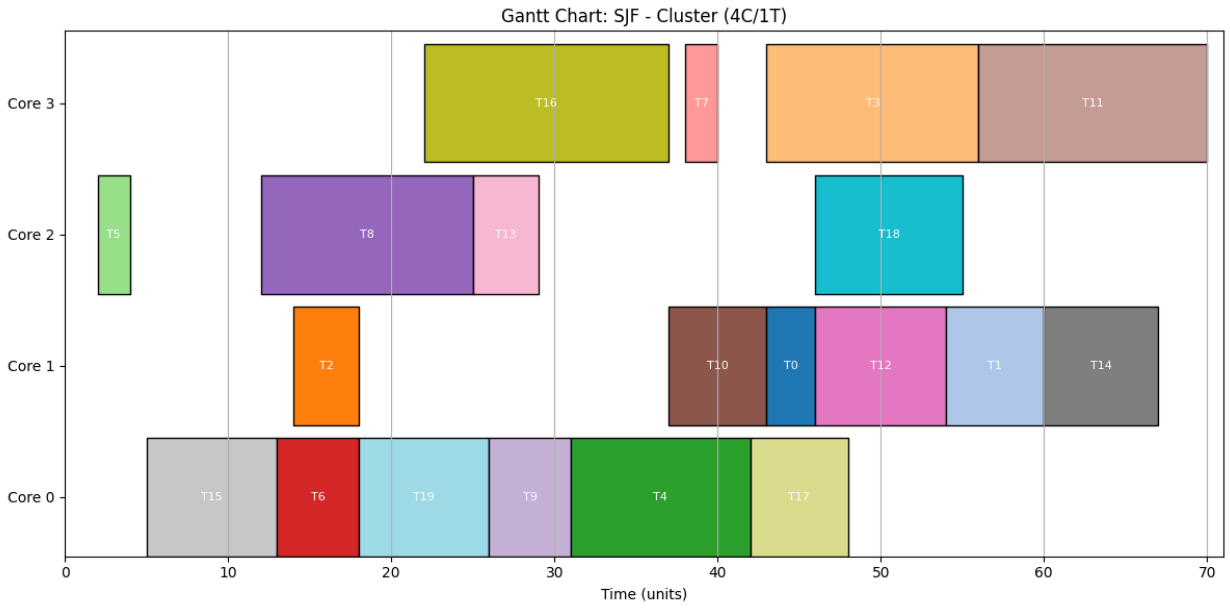

Fig 24 - gantt_Cluster_SJF_4C_1T.png

The image displays a Gantt chart that illustrates task scheduling across four cores in a cluster architecture using the Shortest Job First (SJF) algorithm, with one thread per core. Each horizontal bar represents a task identified by its ID (e.g., T5, T16), with its position and length indicating the start time and duration on a specific core.

The chart highlights task distribution, idle times, and the effectiveness of the SJF algorithm in reducing waiting times by prioritizing shorter tasks, providing insights into core utilization and scheduling efficiency in the simulated cluster environment.

Scheduling Algorithm Comparison

3) Round Robin

i) Promotes fairness.

ii) High context switch overhead.

iii) Longer turnaround time, especially in heavy workloads.

4) Shortest Job First (SJF)

i) Most efficient for average turnaround and CPU usage.

ii) Best throughput across all architectures.

iii) Some risk of starvation for long tasks (not observed due to task size balance).

5) Priority Scheduling

i) Mixed results; depends on distribution of priorities.

ii) Occasionally led to longer execution time due to priority inversion.

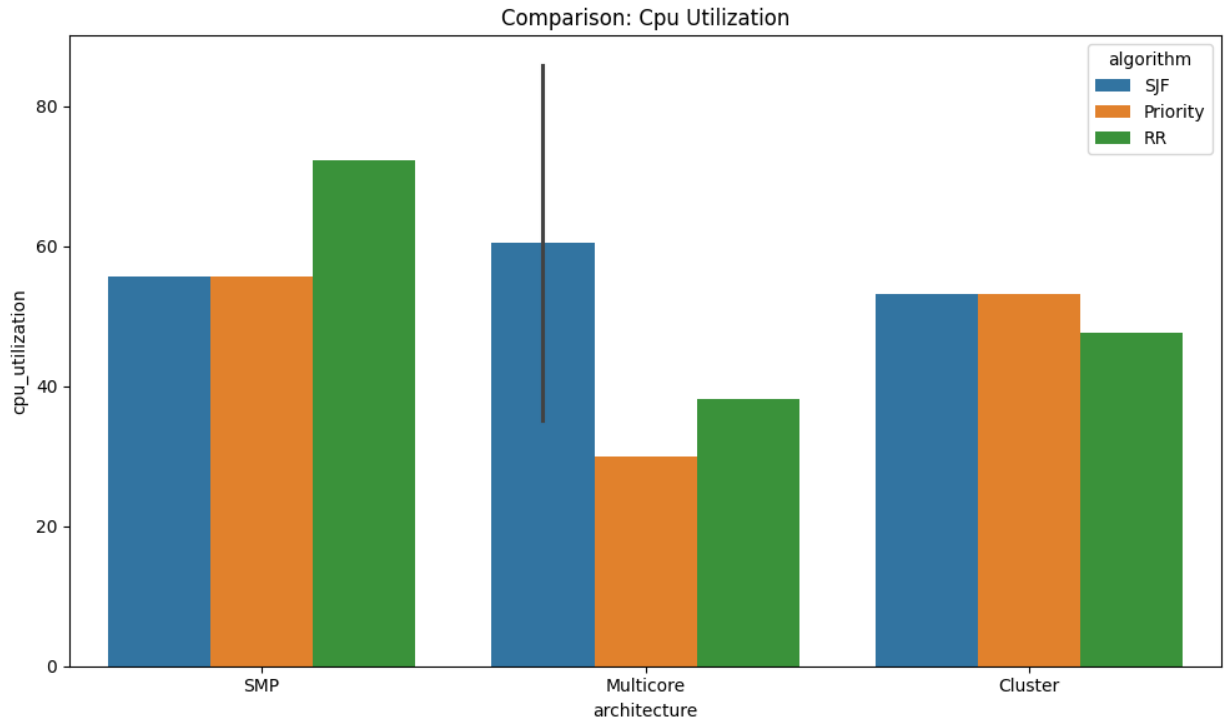

Fig 25 - compare_cpu_utilization.png

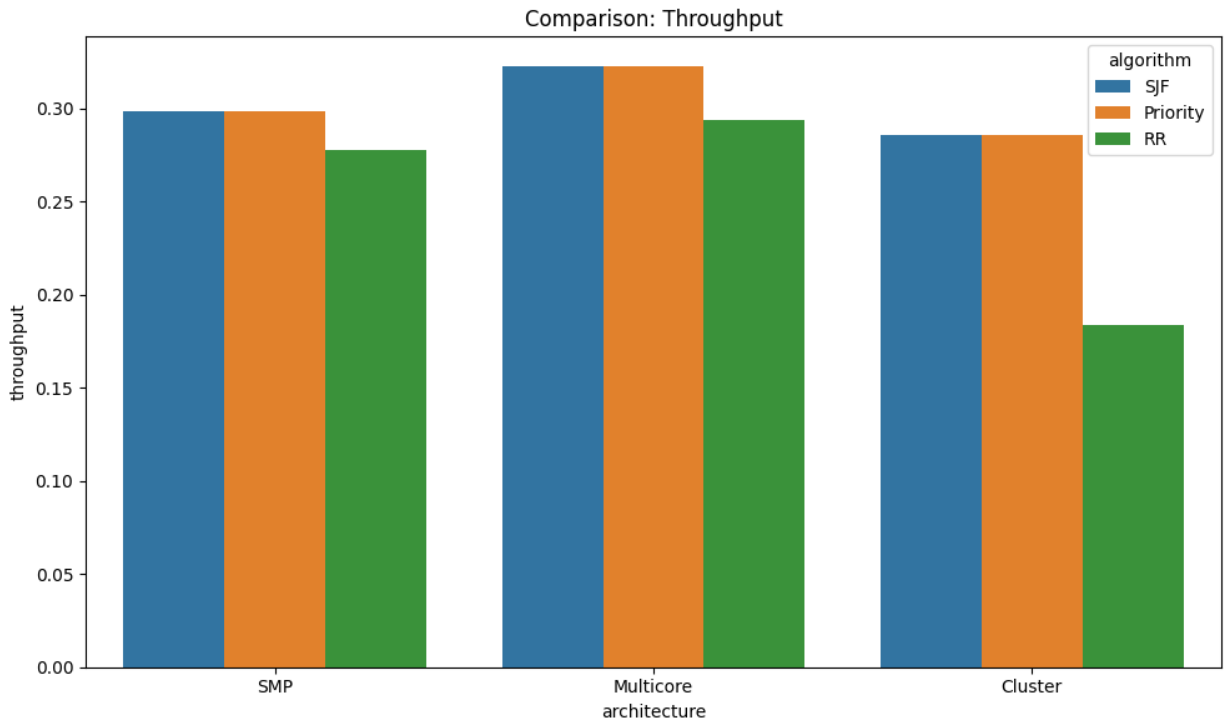

Fig 26 - compare_throughput.png

The image presents a grouped bar chart comparing the throughput of three scheduling algorithms: Shortest Job First (SJF), Priority, and Round Robin (RR) across three processor architectures: SMP, Multicore, and Cluster. Throughput, which measures tasks completed per unit time, is on the vertical axis.

The chart shows that SJF and Priority achieve high throughput across all architectures, with the highest values in the Multicore setup (over 0.32). In contrast, Round Robin consistently yields lower throughput, particularly in the Cluster architecture (around 0.18).

Overall, SJF and Priority are more effective at maximizing task completion rates, while Round Robin is less efficient, especially in distributed Cluster systems.

Architecture	L1 Hits	L1 Misses	L3 Hits	L3 Misses
SMP	Moderate	Moderate	Moderate	High
Cluster	Low	High	Low	High
Multicore	High	Low	High	Low

Multicore architectures benefit from shared cache and low memory access latency. Clusters exhibit higher cache misses due to isolated queues and non-shared memory configurations.

architecture	num_cores	threads_per_core	algorithm	cpu_utilization	avg_turnaround	throughput	context_switches	l1_cache_hits	l1_cache_misses	l3_cache_hits	l3_cache_misses	num_threads	total_time
SMP	4	1	SJF	55.59701493	7.95	0.298507463	20	120	29	19	10		
SMP	4	1	Priority	55.59701493	7.95	0.298507463	20	120	29	19	10		
SMP	4	1	RR	72.22222222	12.05	0.277777778	79	166	42	28	14		
Multicore	4	2	SJF	30.04032258	7.45	0.322580645	20	120	29	19	10		
Multicore	4	2	Priority	30.04032258	7.45	0.322580645	20	120	29	19	10		
Multicore	4	2	RR	38.23529412	10.4	0.294117647	79	166	42	28	14		
Multicore	4	2	SJF	98.02631579			20	120	29	19	10	1	152
Multicore	4	2	SJF	88.69047619			20	120	29	19	10	2	84
Multicore	4	2	SJF	55.59701493			20	120	29	19	10	4	67
Multicore	4	2	SJF	30.04032258			20	120	29	19	10	8	62
Cluster	4	1	SJF	53.21428571	11.5	0.285714286	20	120	29	18	11		
Cluster	4	1	Priority	53.21428571	12.3	0.285714286	20	120	29	18	11		
Cluster	4	1	RR	47.70642202	27.9	0.183486239	79	167	41	27	14		

Fig 27 - architecture_comparison_summary.xlsx

d) Interpretation of Results

1) Multicore + SJF + 8 Threads was the best performing setup in nearly all dimensions.

2) Round Robin introduced excessive context switching, affecting efficiency.

3) Cluster systems were robust for scale-out but introduced latency due to task distribution inefficiencies.

4) SMP underperformed when core count increased, indicating scalability bottlenecks in shared memory environments.

All simulation runs were visualized via,

(2) Gantt charts: Illustrate how tasks were distributed and executed over time across cores.

5) Comparative bar charts: Offer insight into each configuration's performance.

6) Excel files: Allow quantitative evaluation of task-level statistics and macro performance trends.

XII. COMPARATIVE ANALYSIS

In the field of modern computing, symmetric multiprocessing (SMP), cluster computing, and multicore systems are commonly used architectures. SMP uses multiple processors sharing memory and I/O in a single system with a single physical machine. Clusters have multiple standalone systems connecting via a network by using multiple physical machines. Multicore systems have multiple processing cores on a single chip by single physical chip as part of a CPU. Each of these architectures has its advantages and disadvantages. We will explore these three architectures in this comparative analysis in various aspects such as memory models, interconnections, communication, fault tolerance, and cost considerations.

A. ARCHITECTURAL DESIGN AND INTEGRATION

SMP systems have two or more identical processors connected to a single shared main memory. Each processor executes tasks in a single instance of an operating system. All CPUs share common memory and I/O devices, and the entire system is connected via a single motherboard. This architecture is very simple and is ideal for low-latency shared memory. However, as more CPUs are added, the shared memory bus can become a bottleneck.

A cluster consists of multiple systems (nodes), each with one or more processors, memory, and an operating system. They work together as a single system and require high-speed interconnections to form a cluster of nodes. In this architecture, nodes can be added gradually, but it isn't easy to logically and physically separate memory between nodes. Clusters are designed based on distributed parallel computing and fault-tolerant systems.

Multi-core systems combine multiple processor cores on a single chip. Here, a single-core processor is shown with a private L1 cache and a shared L2 cache [1, 2]. The number of cache levels is determined by how far away the main memory is and how many cycles it takes to access the main memory. This integration allows for faster inter-core communication. This architecture operates with tighter integration than SMP. It also operates with lower latency than clusters.

B. MEMORY MODEL AND MANAGEMENT

SMP uses as a shared memory model. Since all processors access the same memory space, it simplifies software development. This model is mostly used for programming frameworks and operating systems. However, when the number of processors is increased, memory and bus access contention occur. Cache coherence mechanisms (MESI or MOESI) are required to maintain consistency across CPU caches.

Clusters use as a distributed memory model. There is no global memory space, and the Message Passing Interface (MPI) is often used to explicitly manage communication. This model scales efficiently and introduces high latency and programming overhead.

Multicore processors have a hybrid model with a combination of shared and private memory components. The cores typically share cache (L2 or L3) and retain private L1 caches. This design provides efficient data transfer and low-latency communication. Integrated protocols implement memory coherency, and the operating system treats the cores as separate logical CPUs using shared memory types like SMP.

C. PERFORMANCE CHARACTERISTICS

SMP systems use shared memory and cache coherence to enable efficient parallel computing, especially when implemented on a programmable chip. This system has been shown to provide scalability through the addition of processor cores without hardware modifications . Furthermore, when the number of processors is increased, SMP systems experience performance limitations due to memory bus contention, cache coherency overhead, and memory access latency. These factors may reduce the speedup benefits beyond the 8 to 32 cores limit.

Cluster computing systems offer high performance by combining the computing power of independent nodes. This system is communicated using message passing protocols such as MPI. Hybrid programming models (MPI + OpenMP) improve performance by reducing inter-node communication overhead. Clusters can be used for data-parallel applications such as large-scale data processing and scientific data processing. Cluster performance affects Load balancing and task distribution.

Multi-core systems offer high performance for multi-threaded and parallel workloads. This architecture combines multiple processing cores on a single chip, enabling low-latency communication. The performance of multi-core processors is affected by factors such as cache ratios, core utilization, and task scheduling. Furthermore, A specific role is played by factors like L1/L2 cache rates, workload type, and dynamic voltage scaling (DVS). Overall, proper parallelization of applications minimizes synchronization and migration costs.

D. SCALABILITY AND FLEXIBILITY

SMP systems use moderate scalability and architectural flexibility. If hardware resources are available, SMP can be scaled up to 8 or more processors on programmable chips (such as FPGAs). When the number of processors increases, clock frequencies degrade due to increased bus contention. SMP has configuration simplicity and standardization. This is suitable for embedded systems or compact computing systems. Overall, even when processors are added, the performance gains are less than a moderate number of cores.

Cluster computing has exceptional scalability, and the computing power can be increased by adding more nodes. Clusters consisting of dual-core SMP nodes show significant performance gains when using hybrid programming (MPI + OpenMP). In this model, clusters can be scaled vertically and horizontally. Clusters are suited for high-performance computing (HPC), big data analytics systems. However, cluster computing is the most scalable and flexible system among the three architectures.

Multicore systems provide efficient and compact scalability, with performance delivered by a single chip. multicore scalability is primarily limited by chip area, thermal design constraints, and inter-core communication overhead. However, unlike clusters, multicore systems cannot scale indefinitely. Multi-core systems provide greater flexibility in workload management, supporting general-purpose and real-time applications. These are ideal for electronic devices and mobile platforms.

E. COMMUNICATION AND INTERCONNECTS

SMP systems use shared memory and system buses to communicate between processors. Bus structures such as the Avalon Bus are used to implement SMP architectures on programmable chips (FPGAs). Avalon uses point-to-point connections that prevent standard cache coherence techniques. A hybrid solution was implemented to solve this, including a Cache Coherency Module (CCM). Thus, SMP systems offer low-latency communication over shared memory and require interconnection logic to ensure system reliability.

Inter-node communication in a cluster computer occurs over network interconnects such as Ethernet using messaging protocols like MPI. Cluster communication is more complex and latency-prone than in SMP or multi-core systems due to the physical separation of nodes. Therefore, performance can be significantly improved by using hybrid programming models. This hybrid model increases memory efficiency and reduces the number of MPI messages. This system incurs a high communication cost compared to other systems.

Multicore systems communicate on a chip designed for fast, low-latency data transfer between cores, and they communicate via shared caches (L2 or L3) and networks-on-chip (NoC). The efficiency of multicore communication is largely affected by memory hierarchy, cache design, and scheduling strategies. Overall, multi-core processors offer the most efficient communication facilities of the three architectures.

F. FAULT TOLERANCE

SMP systems have limited fault tolerance due to the memory architecture and centralized design. Although small-scale SMP systems are cost-effective and flexible, their resilience is shown by implementations of FPGA-based systems. The lack of isolation between processing units in the SMP architecture makes it difficult to recover the entire system after a critical component fails.

Cluster computing provides the highest level of fault tolerance among the three architectures. Clusters consisting of multiple nodes maintain operational continuity when a single node fails. Each node here is a self-contained unit with memory, a processor, and an operating system. Moreover, checkpoints and data redundancy mechanisms can be used to improve reliability.

Multicore systems offer moderate fault tolerance and are limited by their on-chip integration. Multicore processors consist of multiple cores sharing cache levels and memory buses along a single edge. Advanced architectures facilitate fault detection, which can help to correct some

faults. Although multicore systems lack the robustness of clusters, they can maintain system performance even under partial failures with the help of a fault-aware operating system.

G. COST CONSIDERATIONS

SMP systems provide a cost-effective solution for some parallel applications but can be expensive due to the use of specialized hardware. Systems can be built using softcore processors (Altera Nios) on FPGAs with minimal modification to the IP cores. This can reduce hardware development costs. However, as the number of processors increases, it requires larger, more expensive FPGAs. Overall, SMP is a cost-effective, if hardware scalability remains within manageable limits.

Cluster computing systems are cost-effective in terms of using commodity hardware. This reduces the hardware cost per node. Significant costs are required for management software, network infrastructure, and power. Software development using MPI and hybrid models increases complexity and requires administrative costs. Overall, this architecture is cost-effective for high-throughput applications and large data sets.

Multi-core systems are considered a relatively cost-effective performance solution. Compared to creating discrete single-core chips with similar performance, integrating multiple cores on a single silicon die results in lower material costs. High-core-count chips are more expensive because they are manufactured using advanced manufacturing nodes. However, multi-core processors present a cost-effective balance for many commercial and consumer applications.

Final comparative summary

Category	SMP	Cluster	Multicore
Architecture Type	Tightly coupled shared memory architecture	Loosely coupled distributed-memory nodes	Integrated multicore chip
Memory Model	Shared memory	Distributed memory	Shared + private caches
Performance	High for tightly coupled tasks	Excellent for large-scale data-parallel jobs	High for multithreaded apps
Communication	Through shared memory	MPI over a network	On-chip communication
Interconnect	Shared bus or switch fabric	LAN or high-speed interconnects	On-chip interconnect (Bus, Ring, Mesh, Crossbar)
Scalability	Limited	High	Moderate
Fault Tolerance	Low	High	Moderate
Cost	more expensive due to specialized hardware	cost-effective using commodity hardware	cost-effective compared to multiple single-core processors
Examples	Servers, High-end workstations	HPC systems, Web servers	Laptops, Mobile devices

Table 7: Comparative Table

XIII. CONCLUSION & FUTURE WORK

A. CONCLUSION

This project aimed to analyze and evaluate the performance of modern multicore computer systems using a simulation-based approach. We developed a model in Python to investigate the effects of different architectural choices (SMP, Cluster, and Multicore) and scheduling algorithms (SJF, Priority, and RR) on overall system performance under various workloads and thread configurations.

The simulation results revealed that multicore systems, especially those configured with hyper-threading (two threads per core), exhibited superior efficiency and throughput. Among the scheduling strategies, Shortest Job First (SJF) consistently achieved the lowest average turnaround time, the highest CPU utilization, and optimal throughput across all architectures.

Key Findings

Insight	Summary
Multicore Advantage	Shared cache and thread-level parallelism deliver the highest CPU utilization (above 90%) and fastest execution.
SJF Dominance	Outperforms Priority and RR in turnaround and throughput due to smart job ordering.
SMP Scalability Limitations	Shared queues and global contention lead to decreased efficiency with more cores.
Cluster Trade-offs	Offers better node independence but suffers from load imbalance and cache inefficiencies.
Context Switch Overhead	RR scheduling incurs the most switches, impacting cache and execution performance.

Table 8

These insights align with real-world processor behavior, especially in modern multi-threaded, latency-sensitive applications like gaming, data analytics, and server workloads.

B. FUTURE WORK

Based on our analysis, we recommend the following design and strategic choices for future multicore systems and their task scheduling.

a) Use SJF or Hybrid Scheduling in Multithreaded Systems

1) SJF proved to be the most effective scheduling policy in reducing turnaround time and enhancing throughput.

2) RR, while fair, should be used only where fairness is critical (e.g., multi-user environments).

Future Work - Explore dynamic or hybrid scheduling algorithms (e.g., SJF + Priority or Aging + SJF) to mitigate starvation in mixed workloads.

b) Leverage Hyper-Threading Where Possible

1) Simulated 4-core/8-thread Multicore systems drastically improved performance compared to 4-thread setups.

2) Hyper-threading reduces idle time and improves utilization of cache and execution units.

Real-World Implication - Ideal for compute-intensive workloads such as AI, media encoding, and multi-user server processing.

c) Avoid Monolithic SMP Designs for Scalable Systems

1) SMP systems showed significant performance bottlenecks due to shared memory and queue contention as core count increased.

2) Such architectures are unsuitable for workloads requiring real-time or highly parallel processing.

Recommended Use - SMP may still be viable for small embedded systems or simple parallel tasks with minimal contention.

d) Enhance Cache Coherency and Memory Access Policies

1) Cache performance is tightly coupled with task scheduling and memory architecture.

2) Multicore systems achieved better hit rates due to shared L3 caches and efficient scheduling.

Future Simulation - Integrate NUMA models and memory allocation strategies to analyze data locality effects.

e) Consider Load Balancing Strategies in Cluster Architectures

1) Cluster setups underperformed when task distribution was uneven, leading to core idleness.

2) Advanced load balancers and distributed schedulers could improve cluster efficiency.

Suggested Extension - Implement and compare load balancing algorithms like round-robin balancing or least-loaded core assignment.

This study can be expanded in several ways to better reflect realistic and advanced computing scenarios.

3) Thermal modeling and power consumption analysis to study energy efficiency trade-offs.

4) Simulate heterogeneous multicore architectures (e.g., ARM big.LITTLE).

5) Integration of real-world workload traces from operating system logs or benchmark suites.

6) Add GPU offloading and hybrid compute models to simulate heterogeneous platforms.

The simulation effectively replicates the key behaviors of multicore processors, providing valuable insights into system design choices. These findings support industry trends in multicore development, highlighting the significance of parallelism, thread-aware scheduling, and memory efficiency in contemporary computing.

XIV. REFERENCES

Diamond, J. a.-D. (2011). Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software* (pp. 32-43).

Doweck, J. (2006). Inside Intel® Core microarchitecture. *2006 IEEE Hot Chips 18 Symposium (HCS)* (pp. 1–35). Stanford, CA, USA: IEEE.

Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., Stamm, R. L., & Tullsen, D. M. (1997, September). Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro, 17*(5), 12-19.

Freitas, H. C. (2008). A High-Throughput Multi-cluster NoC Architecture. In *11th IEEE International Conference on Computational Science and Engineering* (pp. 56-63). Sao Paulo, Brazil.

Hennessy, J., & Patterson, D. A. (2007). Computer Architecture. *4th*, pp. 2–104. San Francisco, CA: Morgan Kaufmann .

Hung, A. a. (2005). Symmetric multiprocessing on programmable chips made easy. In *in Design, Automation and Test in Europe.* Ontario.

Jadon, S., & Yadav, R. S. (2016). Multicore processor: Internal structure, architecture, issues, challenges, scheduling strategies and performance. *2016 11th International Conference on Industrial and Information Systems (ICIIS)* (pp. 381–386). Roorkee, India: IEEE.

Lempel, O. (2011). 2nd Generation Intel® Core Processor Family: Intel® Core i7, i5 and i3. *2011 IEEE Hot Chips 23 Symposium (HCS).* Stanford, CA, USA: IEEE.

Li, Y., Ling, L., & Wu, J. (2011). The application of pipeline technology: An overview. *2011 6th International Conference on Computer Science & Education (ICCSE)* (pp. 47–51). IEEE.

Litz, H. a. (2010). TCCluster: A Cluster Architecture Utilizing the Processor Host Interface as a Network Interconnect. In *2010 IEEE International Conference on Cluster Computing* (pp. 9-18). Heraklion, Greece.

Misra, S., Alfa, A. A., Olaniyi, M. O., & Adewale, S. O. (2014). Exploratory study of techniques for exploiting instruction-level parallelism. *2014 Global Summit on Computer & Information Technology (GSCIT)* (pp. 1–6). Sousse, Tunisia: IEEE.

Moore, G. (n.d.). Progress in Digital Electronics. *Technical Digest of the International Electron Devices Meeting*, (p. 13). 1975.

Olukotun, B. A. (1997, September). A single-chip multiprocessor. *Computer, 30*(9), 79-85.

Qingbo, Y. a. (2009). A Scalability Analysis of the Symmetric Multiprocessing Architecture in Multi-Core System. In *2009 IEEE International Conference on Networking, Architecture, and Storage* (pp. 231-234). Zhangjiajie, China.

Shah, N. D., Shah, Y. H., & Modi, H. (2013). Comprehensive study of the features, execution steps and microarchitecture of the superscalar processors. *2013 IEEE International Conference on Computational Intelligence and Computing Research* (pp. 1–4). Enathi, India: IEEE.

Tatjana R. Nikolić, G. S. (2022). From Single CPU to Multicore Systems. *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, (pp. 1-8). Ohrid .

Yizhong Ma and Ma, J. a. (2008). Performance optimization for SMP cluster operation. In *2008 IEEE International Symposium on IT in Medicine and Education* (pp. 925-929). Xiamen, China.