# Hybrid DAG-Based Multi-Core Scheduler: A Task Scheduling Approach for Multi-Core Systems

Dhanya Girdhar[1], Raashi Sharma[2], and Aastha Malhotra[3]

[1]2310110463 (dg218@snu.edu.in)

[2]2310110566 (rs706@snu.edu.in)

[2]2310110007 (am565@snu.edu.in)

## ABSTRACT

Modern multi-core processors have revolutionized computational performance by enabling parallel execution of tasks. However, efficiently scheduling tasks across multiple cores remains a significant challenge, especially when task dependencies must be considered. Traditional scheduling algorithms, such as Round Robin and First-Come-First-Serve, struggle to handle these dependencies effectively, leading to suboptimal CPU utilization and increased execution time. Our project introduces a hybrid scheduler that combines DAG-based scheduling with Rate Monotonic Scheduling (RMS) for priority assignment and load balancing. Implemented in C with Pthreads, our scheduler allows simulation of task execution across multiple cores, respecting dependencies and priority, and outputs performance metrics such as turnaround time.

## 1. INTRODUCTION

### 1.1 Motivation

In today's computing landscape, the proliferation of multi-core processors has shifted the performance bottleneck from hardware capabilities to software efficiency. While hardware manufacturers continue to increase core counts, many applications fail to fully utilize this parallel architecture due to inefficient task scheduling. This gap is particularly pronounced in systems with complex task dependencies, where suboptimal scheduling decisions can lead to processor cores sitting idle while important tasks wait in queue. Our motivation stems from addressing this critical gap between hardware potential and software utilization by developing a scheduling algorithm that intelligently navigates task dependencies while maximizing parallel execution. By creating a hybrid solution that respects both the structural constraints of task relationships and real-time execution priorities, we aim to bridge this gap and enable applications to fully leverage the computational power offered by modern multi-core systems.

### 1.2 Objectives

- Design an efficient task scheduler to accommodate dependent tasks within a DAG-based scheduling technique.
- Implement Rate Monotonic Scheduling (RMS) to always execute high priority tasks first.
- Balance task execution across multiple cores by using load balancing to ensure that all CPU cores participate equally in task execution.
- Reduce execution time and improve system performance throughput over traditional scheduling techniques.
- Allow users to test with both sample and custom DAGs.
- Evaluate scheduler performance with key metrics such as execution time and core utilization.
- Compare the results of our algorithm with standard process scheduling algorithms like FCFS and round robin.

## 2. BACKGROUND AND LITERATURE REVIEW

### 2.1 Scheduling Parallel Real-Time Tasks for Multi-core Systems

Paper Focus: Parallel Real-Time Task Scheduling (PRTTS) to reduce cache/memory contention using dynamic priorities and memory-aware scheduling.

Implementation in our algorithm:

- Priority assignment using Rate Monotonic Scheduling (Real Time Scheduling method), high-frequency tasks get higher priority.

- Fixed priority model for real-time urgency: Once a task is given a priority, that priority doesn't change during the entire scheduling process.

### 2.2 DAG Scheduling and Analysis on Multi-Core Systems by Modelling Parallelism and Dependency

Paper Focus: Introduces Parallelism-aware Workload Distribution Model (P-WDM) for better scheduling by minimizing task interference.

Implementation in our algorithm: Our algorithm optimizes for meeting periodic deadlines (RMS guarantees) whereas P-WDM approach Optimizes for overall throughput by minimizing negative task interactions

### 2.3 Efficiently Scheduling Parallel DAG Tasks on Identical Multiprocessors (SFS Model)

Paper Focus: Combines Federated Scheduling with Semi-Partitioned Scheduling using the SFS (Segmented-Flattened-and-Split) is a modern scheduling strategy designed for DAG-based parallel tasks on identical multiprocessors.

Implementation in our algorithm: We use a dynamic task assignment approach where tasks are assigned to cores as they become available, prioritized by RMS criteria. This contrasts with SFS's combined federated and semi-partitioned scheduling approach.

### 2.4 Parallel Path Progression DAG Scheduling

Paper Focus: Introduces Parallel Path Progression (PPP) in which multiple execution paths, not just critical path; allows for efficient parallel execution with minimal overhead by grouping priorities.

Implementation in our algorithm: We allow for parallel path execution via topological sorting and thread-based simulation. While we do not explicitly model multiple critical paths, our support for concurrent execution simulates the intent.

### 2.5 Efficient Process Scheduling for Multi-Core Systems

Paper Focus: Introduces Longest Path First In (LPFI), which prioritizes tasks with the longest dependency chain.

Implementation in our algorithm: Although we do not compute the longest paths explicitly, our use of DAGs, topological sort, and RMS collectively captures similar benefits by ensuring deep dependencies are scheduled earlier if they have higher urgency.

## 3. METHODOLOGY

### 3.1 Rate Monotonic Scheduling (RMS) Algorithm

```
FUNCTION apply_rate_monotonic_scheduling(dag):
    FOR each task IN dag.tasks:
        IF task.period == 0 THEN
            // Non-periodic task → assign lowest priority
            task.priority ← 1
        ELSE
            // Assign priority inversely proportional to period
            priority ← 10 - (task.period × 9) ÷ 1000
            // Clamp priority within [1, 10]
            IF priority < 1 THEN
                priority ← 1
            ELSE IF priority > 10 THEN
                priority ← 10
            END IF
            task.priority ← priority
        END IF
        IF debug_mode IS TRUE THEN
            PRINT "Task", task.id, "(", task.name, "):",
                "Period =", task.period, ", Priority =", task.
                   priority
        END IF
    END FOR
END FUNCTION
```

*Purpose*

The `apply_rate_monotonic_scheduling` function assigns priorities to tasks in a Directed Acyclic Graph (DAG) based on the Rate Monotonic Scheduling (RMS) algorithm. This is a fixed-priority scheduling method commonly used in real-time systems, where tasks with shorter periods are given higher priorities to ensure timely execution.

*Logic Overview*

- Period Check: If a task's period is 0, it is treated as non-periodic and is assigned the lowest possible priority (1).
- Priority Calculation: For periodic tasks, the priority is computed using the formula:

  ```
  priority = 10 - ((period * 9) / 1000)
  ```

  This formula scales task priorities within a 1–10 range, where:

  - Shorter periods result in higher priorities (closer to 10).
  - Longer periods result in lower priorities (closer to 1).

- Priority Bound Enforcement: To ensure robustness, any calculated priority is clamped between 1 (lowest) and 10 (highest).

*Priority Assignment for Non-Periodic Tasks*

Periodic tasks often have timing constraints and deadlines related to their periods. : Periodic tasks typically represent ongoing system functions that need guaranteed service (like sensor readings, control loops, etc.), while non-periodic tasks are often one-off jobs that can be deferred. Since, RMS is designed for periodic tasks, a non-periodic task can be thought of as having an infinite period as it never repeats itself and therefore it makes sense to give them lower priority.

### 3.2 Task Readiness Evaluation Based on Dependencies

```
FUNCTION is_task_ready(dag, task_id):
    IF dag.tasks[task_id].completed IS TRUE THEN
        RETURN FALSE
    END IF
    FOR each dependency_id IN dag.tasks[task_id].dependencies:
        IF dag.tasks[dependency_id].completed IS FALSE THEN
            RETURN FALSE
        END IF
    END FOR

    RETURN TRUE
END FUNCTION
```

*Purpose*

The `is_task_ready` function determines whether a specific task in a Directed Acyclic Graph (DAG) is ready to be executed. A task is considered ready if:

- It has not already been completed, and
- All its dependencies (i.e., tasks it relies on) have been completed.

*Logic Overview*

- Check Task Completion: If the task has already been marked as completed, the function returns false, since there's no need to re-execute it.
- Check Dependencies:
    - The function loops through the list of task IDs that the current task depends on.
    - For each dependency, it checks whether the dependent task is completed.

    - If any dependency is not yet completed, the task is not ready, and the function returns false.

- Check Ready Status: If all checks pass (i.e., task is not completed and all its dependencies are completed), the function returns true, indicating that the task is ready to be scheduled or executed.

## 3.3 Selecting the Optimal Ready Task via RMS

```
    FUNCTION find_highest_priority_ready_task(dag):
    highest_priority ← -1
    selected_task ← -1
    FOR i FROM 0 TO dag.num_tasks - 1 DO:
        IF is_task_ready(dag, i) AND dag.tasks[i].core_assigned ==
           -1 THEN:
            IF selected_task == -1 OR dag.tasks[i].priority >
               highest_priority THEN:
                highest_priority ← dag.tasks[i].priority
                selected_task ← i
            ELSE IF dag.tasks[i].priority == highest_priority AND
                    dag.tasks[i].period < dag.tasks[selected_task].
                        period THEN:
                selected_task ← i
            END IF
        END IF
    END FOR
    RETURN selected_task
END FUNCTION
```

*Purpose*

This function identifies the most eligible task for execution among all ready and unassigned tasks in the DAG, based on Rate Monotonic Scheduling (RMS). RMS assigns higher priorities to tasks with shorter periods.

*Logic Overview*

- Initialization:
    - `highest_priority` is initialized to -1 as a sentinel value to track the best priority found.
    - `selected_task` is initialized to -1, which indicates no task has been selected yet.
- Iteration Over All Tasks: The function loops through all tasks in the DAG.
- Readiness and Assignment Check: It only considers tasks that are:
    - Ready to execute (checked using `is_task_ready`), and
    - Not currently assigned to any core (i.e., `core_assigned == -1`).
- Priority Comparison:
    - If this is the first valid task or if the task has a higher priority than the current best, it becomes the new `selected_task`.
    - If two tasks have the same priority, the one with the shorter period is preferred (finer-grained RMS tie-breaker).
- Return Value:
    - After evaluating all tasks, the function returns the index (`task_id`) of the task that should be executed next based on RMS priority.
    - If no task is ready, it returns -1.

**3.4 Hybrid DAG + RMS Scheduler**

```
FUNCTION simulate_hybrid_scheduler(dag, num_cores):
PRINT "Running Hybrid DAG-based Scheduler with RMS..."
CALL reset_dag_execution(dag)
simulation_time ← 0
completed_tasks ← 0
ALLOCATE cores[num_cores]
FOR each core i:
    INIT core[i] with id, idle, and zeroed fields
WHILE completed_tasks < dag.num_tasks:
    FOR each core i:
        IF core[i] has task:
            task ← core[i].current_task
            task.remaining_time -= 1
            core[i].time_slice_remaining -= 1
            IF task is finished:
                UPDATE task completion info
                SET core[i] idle
                completed_tasks += 1
                PRINT "Completed"
            ELSE IF time slice over:
                SET core[i] idle
                PRINT "Preempted"
    FOR each core i:
        IF core[i] is idle:
            task_id ← find_highest_priority_ready_task(dag)
            IF task_id ≠ -1:
                ASSIGN task to core[i], set time slice, start
                   time if needed
                PRINT "Started"
```

```
simulation_time += 1

FOR each idle core i:
    core[i].total_idle_time += 1

IF simulation_time MOD 20 == 0:
    PRINT progress bar

CALL delay_ms(10)

IF simulation_time > 10000:
    PRINT "Simulation exceeded time limit"
    BREAK
```

*Function Overview*

The `simulate_hybrid_scheduler` function orchestrates a hybrid task scheduler built on DAG-based task dependency tracking combined with Rate Monotonic Scheduling (RMS) for priority decisions. It simulates task execution over multiple cores, tracks core activity, and measures performance metrics such as turnaround time and core utilization.

- Initialization Phase:
    - Reset DAG execution state and initialize the simulation timer and completed task counter.
    - Dynamically allocate and configure `num_cores` cores, marking each as idle and initializing their tracking attributes.
- Simulation Loop: The core simulation loop runs until all tasks are completed. In each simulation cycle:

- Task Execution Check (each non-idle core is checked):
    - If the task finishes (`remaining_time <= 0`), it is marked completed and unassigned.
    - If the time slice expires, the task is preempted and returned to the ready queue.
- Task Assignment: Each idle core queries for the highest priority ready task.The selected task is assigned to the core and begins execution if not already started.
- Time Advancement:
    - Simulation time is incremented.
    - Idle time is accumulated for idle cores.
    - A visual progress bar is updated every 20 units.
- Safety Control: A maximum time limit (10000 units) prevents infinite loops or deadlocks.

### 3.5 Post-Simulation Analysis

```
PRINT "Simulation completed in", simulation_time

// Results Summary
PRINT table header
total_turnaround ← 0
FOR each task:
    turnaround ← finish_time - start_time
    total_turnaround += turnaround
    PRINT task info
PRINT average turnaround time

// Core Utilization
```

```
total_utilization ← 0
FOR each core i:
    busy_time ← simulation_time - core[i].total_idle_time
    utilization ← busy_time / simulation_time * 100
    total_utilization += utilization
    PRINT core stats
PRINT average utilization


FREE cores
```

*Execution Results*

Once all tasks are completed, the function outputs detailed information about each task:

- Task ID, Name, Duration, Period, Priority
- Start Time, Finish Time, and Turnaround Time

The average turnaround time across all tasks is then computed and displayed. This metric gives insight into the scheduler's responsiveness and overall latency in processing task workloads.

*Core Utilization Statistics*

The function then reports how effectively each core was used during the simulation:

- Busy Time: Duration the core was actively executing tasks.
- Idle Time: Time the core was waiting for task assignments.
- Utilization (%): Ratio of busy time to total simulation time.

The average core utilization across all cores is calculated, providing a high-level view of how balanced and efficient the task distribution was under the scheduler.

## 3.6 DAG Cycle Detection

```
    FUNCTION detect_cycles(dag):
    ALLOCATE visited[dag.num_tasks] as false
    ALLOCATE rec_stack[dag.num_tasks] as false
    has_cycle ← false

    FOR each task i in dag:
        IF not visited[i]:
            CALL dfs_cycle_detection(dag, i, visited, rec_stack,
               has_cycle)

    dag.has_cycles ← has_cycle

    FREE visited and rec_stack

    IF has_cycle:
        PRINT "WARNING: Cycles detected in DAG"
END FUNCTION
```

*Function Overview*

The `detect_cycles` function verifies the validity of a DAG (Directed Acyclic Graph) by checking for cyclic dependencies among tasks. Cycles in the graph indicate invalid task dependencies that can prevent successful scheduling. The function uses Depth-First Search (DFS) with recursion tracking to detect such cycles.

- Initialization Phase

  Allocate two boolean arrays:

  - `visited[]` to track whether each task has been visited.
  - `rec_stack[]` (recursion stack) to track the current DFS path.

  Initialize `has_cycle` to false to track whether any cycle is detected.

- Cycle Detection Loop

Iterate through all tasks in the DAG:

For each unvisited task, call the `dfs_cycle_detection` function:

- Performs recursive DFS from the current task.
- Tracks the call stack using `rec_stack[]`.
- If a task is revisited while still in the recursion stack, a cycle is found.
- Sets `has_cycle = true`.

- Post-Processing

  - Store the result in `dag->has_cycles`.
  - Free the dynamically allocated memory (`visited[]` and `rec_stack[]`).

**3.7 Recursive DFS for Cycle Detection**

```
IF has_cycle is already TRUE:
    RETURN


Mark visited[node] = TRUE
Mark rec_stack[node] = TRUE


FOR each task i in dag:
    IF there is an edge from node to i THEN:
        IF i is not visited THEN:
            CALL dfs_cycle_detection(dag, i, visited, rec_stack,
               has_cycle)
            IF has_cycle is TRUE:
                RETURN
        ELSE IF rec_stack[i] is TRUE THEN:
            SET has_cycle = TRUE
            RETURN
```

```
Mark rec_stack[node] = FALSE
```

*Function Overview*

The `dfs_cycle_detection` function implements a depth-first traversal of a DAG to identify cycles. It uses a recursive DFS pattern combined with a recursion stack to detect back edges, which indicate cycles.

- DFS Setup and Base Case: If a cycle has already been found (`has_cycle == true`), exit early to prevent unnecessary computation. Mark the current node as:
    - visited – to avoid reprocessing it in future DFS paths.
    - in the recursion stack – to track the current path of active recursion calls.
- Recursive Traversal and Cycle Check: For each `task i` in the graph, if an edge exists from node to `i (adjacency_matrix[node][i] == 1)`, check:
    - If `task i` has not been visited, recursively explore it.
    - If a cycle is detected deeper in the recursion, bubble up and exit early.
    - If `task i` is already in the current recursion path (`rec_stack[i] == true`), this forms a back edge, confirming a cycle, and `has_cycle` is set to true.
- Backtracking Phase: Once all neighbors are processed, remove the current node from the recursion stack (`rec_stack[node] = false`) as DFS backtracks.

## 4. RESULTS AND DISCUSSION

### 4.1 Test Case 1: Hybrid DAG-Based Scheduler Simulation with 11 Tasks on 3 Cores

**Table 1: Task Details (Test Case 1)**

| ID | Name | Duration | Period | Priority | Dependencies |
|---|---|---|---|---|---|
| 0 | Task0 | 500 | 200 | 9 | None |
| 1 | Task1 | 400 | 300 | 8 | None |
| 2 | Task2 | 600 | 10 | 10 | None |
| 3 | Task3 | 400 | 130 | 9 | None |
| 4 | Task4 | 500 | 250 | 8 | 1 |
| 5 | Task5 | 600 | 330 | 8 | 1, 2 |
| 6 | Task6 | 200 | 10 | 10 | 2 |
| 7 | Task7 | 400 | 340 | 7 | 2 |
| 8 | Task8 | 500 | 20 | 10 | 3 |
| 9 | Task9 | 100 | 20 | 10 | 0 |
| 10 | Task10 | 400 | 200 | 9 | 0 |

We ran our code with the above task set.

**Simulation completed in** : 1801 time units.

**Table 2: Execution Results for Test Case 1**

| ID | Name | Duration | Period | Priority | Start | Finish | Turnaround |
|---|---|---|---|---|---|---|---|
| 0 | Task0 | 500 | 200 | 9 | 0 | 500 | 500 |
| 1 | Task1 | 400 | 300 | 8 | 800 | 1200 | 400 |
| 2 | Task2 | 600 | 10 | 10 | 0 | 600 | 600 |
| 3 | Task3 | 400 | 130 | 9 | 0 | 400 | 400 |
| 4 | Task4 | 500 | 250 | 8 | 1200 | 1700 | 500 |
| 5 | Task5 | 600 | 330 | 8 | 1200 | 1800 | 600 |
| 6 | Task6 | 200 | 10 | 10 | 600 | 800 | 200 |
| 7 | Task7 | 400 | 340 | 7 | 900 | 1300 | 400 |
| 8 | Task8 | 500 | 20 | 10 | 400 | 900 | 500 |
| 9 | Task9 | 100 | 20 | 10 | 500 | 600 | 100 |
| 10 | Task10 | 400 | 200 | 9 | 600 | 1000 | 400 |

**Average Turnaround Time:** 418.18 time units

**Table 3: Core Utilization Statistics for Test Case 1**

| Core | Busy Time | Idle Time | Utilization % |
|---|---|---|---|
| 0 | 1800 | 1 | 99.94% |
| 1 | 1700 | 101 | 94.39% |
| 2 | 1100 | 701 | 61.08% |

**Average Core Utilization:** 85.14%

*Priority Scheduling (RMS Compliance)*

- Tasks with shorter periods, such as Task2 (Period: 10 ms), Task6 (10 ms), and Task8 (20 ms), were consistently scheduled earlier and more frequently across iterations.
- This reflects correct implementation of RMS, where lower-period tasks are given higher priority (priority 10 in this case).

*Dependency Management*

- Tasks were scheduled in adherence to the DAG constraints.
- For example, Task6, Task9, and Task10 were only executed after Task2 finished, maintaining proper precedence.

*Load Balancing & Core Distribution*

- Core 0 had the highest utilization (99.94%), handling critical tasks like Task2, Task6, and Task4, indicating it carried the bulk of high-priority workload.
- Core 1 was efficiently used for mid- and high-priority tasks like Task3, Task8, and Task5, reaching 94.39% utilization.
- Core 2 had comparatively lower utilization (61.08%), executing longer and less frequent tasks like Task0, Task9, and Task7.

*System Efficiency*

- Despite the complexity of dependencies and priority-based preemptions, the system achieved 85.14% average utilization and kept idle time minimal (only 1 time unit on Core 0).
- The average turnaround time of 418.18 indicates timely task completion and balanced throughput.

## 4.2 Test Case 2: Hybrid DAG-Based Scheduler Simulation with 20 Tasks on 3 Cores

**Table 4: Task Details (Test Case 2)**

| ID | Name | Duration | Period | Priority | Dependencies |
|---|---|---|---|---|---|
| 0 | Task0 | 300 | 900 | 2 | None |
| 1 | Task1 | 600 | 900 | 2 | None |
| 2 | Task2 | 300 | 590 | 5 | None |
| 3 | Task3 | 489 | 308 | 8 | None |
| 4 | Task4 | 328 | 804 | 3 | 9 |
| 5 | Task5 | 396 | 805 | 3 | 0 |
| 6 | Task6 | 297 | 974 | 2 | 9 |
| 7 | Task7 | 563 | 678 | 4 | 1 |
| 8 | Task8 | 375 | 60 | 10 | 2 |
| 9 | Task9 | 850 | 397 | 7 | 2 |
| 10 | Task10 | 798 | 527 | 6 | 3 14 |
| 11 | Task11 | 783 | 802 | 3 | 4 |
| 12 | Task12 | 908 | 830 | 3 | 4 |
| 13 | Task13 | 938 | 90 | 10 | 7 |
| 14 | Task14 | 379 | 892 | 2 | None |
| 15 | Task15 | 567 | 724 | 4 | 8 |
| 16 | Task16 | 739 | 739 | 4 | 1 6 |
| 17 | Task17 | 932 | 202 | 9 | 2 |
| 18 | Task18 | 290 | 685 | 4 | 10 |
| 19 | Task19 | 291 | 921 | 2 | 3 10 |

**Simulation completed in** : 4201 time units.

**Table 5: Execution Results for Test Case 2)**

| ID | Name | Duration | Period | Priority | Start | Finish | Turnaround |
|---|---|---|---|---|---|---|---|
| 0 | Task0 | 300 | 900 | 2 | 1242 | 1542 | 300 |
| 1 | Task1 | 600 | 900 | 2 | 1339 | 2699 | 1360 |
| 2 | Task2 | 300 | 590 | 5 | 0 | 300 | 300 |
| 3 | Task3 | 489 | 308 | 8 | 0 | 489 | 489 |
| 4 | Task4 | 328 | 804 | 3 | 1542 | 1870 | 328 |
| 5 | Task5 | 396 | 805 | 3 | 1639 | 2035 | 396 |
| 6 | Task6 | 297 | 974 | 2 | 2943 | 3240 | 297 |
| 7 | Task7 | 563 | 678 | 4 | 2699 | 3262 | 563 |
| 8 | Task8 | 375 | 60 | 10 | 300 | 675 | 375 |
| 9 | Task9 | 850 | 397 | 7 | 489 | 1339 | 850 |
| 10 | Task10 | 798 | 527 | 6 | 1311 | 2109 | 798 |
| 11 | Task11 | 783 | 802 | 3 | 1870 | 2653 | 783 |
| 12 | Task12 | 908 | 830 | 3 | 2035 | 2943 | 908 |
| 13 | Task13 | 938 | 90 | 10 | 3262 | 4200 | 938 |
| 14 | Task14 | 379 | 892 | 2 | 0 | 1311 | 1311 |
| 15 | Task15 | 567 | 724 | 4 | 675 | 1242 | 567 |
| 16 | Task16 | 739 | 739 | 4 | 3240 | 3979 | 739 |
| 17 | Task17 | 932 | 202 | 9 | 300 | 1232 | 932 |
| 18 | Task18 | 290 | 685 | 4 | 2109 | 2399 | 290 |
| 19 | Task19 | 291 | 921 | 2 | 2653 | 2944 | 291 |

**Average Turnaround Time:** 640.75 time units

**Table 6: Core Utilization Statistics for Test Case 2)**

| Core | Busy Time | Idle Time | Utilization % |
|---|---|---|---|
| 0 | 4145 | 56 | 98.67% |
| 1 | 3935 | 266 | 93.67% |
| 2 | 3043 | 1158 | 72.44% |

**Average Core Utilization:** 88.26%

*Priority Scheduling (RMS Compliance)*

RMS assigns higher priority to tasks with shorter periods. This was strictly followed:

- High Priority: Task 8 (60 ms), Task 13 (90 ms) – Priority 10
- Low Priority: Task 0, 1, 14 ( $\approx$ 900 ) – Priority 2
- Tasks were executed respecting these priorities and their dependencies. For example, Task 13 ran early on Core 1, while Task 10 only executed after Tasks 3 and 14 were completed.

*Dependency Management*

All task dependencies defined in the DAG were correctly respected by the scheduler. Task 10 only started execution after both 3 and 14 tasks were done on Core 1.

*Load Balancing & Core Distribution*

Task distribution across cores was dynamic, based on priority and availability:

- Core 0 handled most high-priority tasks and was nearly maxed out (98.67%).
- Core 1 was also well-utilized (93.67%).
- Core 2 was underutilized (72.44%) as compared to Core 0 and Core 1.

This distribution indicates that the scheduler prioritized high-frequency tasks on Cores 0 and 1 to meet tighter deadlines, while Core 2 absorbed lower-frequency or dependency-bound tasks, effectively balancing real-time constraints with DAG integrity.

*System Efficiency*

The system demonstrated average core utilization of 88.26%, which reflects the scheduler's effective use of available computational resources.

- Core 0 reached near-maximum utilization (99.67%), showcasing its role in handling a large portion of high-priority and high-frequency tasks.
- Core 1, with 93.67% utilization, executing a diverse mix of periodic and dependent tasks while maintaining timing guarantees.
- Core 2, though slightly less loaded at 72.44%, effectively handled lower-frequency and dependency-heavy tasks. This ensured that high-priority cores remained focused on time-sensitive jobs, thereby maintaining system responsiveness and integrity.

### 4.3 Test Case 3: Hybrid DAG-Based Scheduler Simulation with 5 Tasks on 3 Cores

**Table 7: Task Details (Test Case 2)**

| ID | Name | Duration | Period | Priority | Dependencies |
|---|---|---|---|---|---|
| 0 | Task0 | 200 | 0 | 1 | None |
| 1 | Task1 | 300 | 0 | 1 | 0 |
| 2 | Task2 | 590 | 0 | 1 | 1 |
| 3 | Task3 | 279 | 0 | 1 | 2 |
| 4 | Task4 | 280 | 0 | 1 | 3 |

**Simulation completed in** : 1650 time units.

**Table 8: Execution Results for Test Case 3)**

| ID | Name | Duration | Period | Priority | Start | Finish | Turnaround |
|---|---|---|---|---|---|---|---|
| 0 | Task0 | 200 | 0 | 1 | 0 | 200 | 200 |
| 1 | Task1 | 300 | 0 | 1 | 200 | 500 | 300 |
| 2 | Task2 | 590 | 0 | 1 | 500 | 1090 | 590 |
| 3 | Task3 | 279 | 0 | 1 | 1090 | 1369 | 279 |
| 4 | Task4 | 280 | 0 | 1 | 1369 | 1649 | 280 |

**Average Turnaround Time:** 329.80 time units

**Table 9: Core Utilization Statistics for Test Case 3)**

| Core | Busy Time | Idle Time | Utilization % |
|---|---|---|---|
| 0 | 1649 | 1 | 99.94% |
| 1 | 0 | 1650 | 0.00% |
| 2 | 0 | 1650 | 0.00% |

**Average Core Utilization:** 33.31%

*Priority Scheduling (RMS Compliance)*

All tasks were assigned the same: Period: 0 ms and Priority: 1
RMS assigns higher priority to tasks with shorter periods. Since all tasks had equal periods, RMS priority had no effect. RMS compliance was maintained technically, but it was not influential in task scheduling.

*Dependency Management*

The task structure followed a strict linear dependency chain (Task0 → Task1 → Task2 → ...). Each task only began after its direct predecessor completed execution. All task dependencies were respected without violations.

*Load Balancing & Core Distribution*

This test case revealed a key limitation of the system when handling linear task graphs. Due to the strict sequential nature of task dependencies, all five tasks were executed exclusively on Core 0. Core 1 and Core 2 remained entirely idle throughout the simulation.

*System Efficiency*

Despite being a constrained case, the system achieved strong individual core efficiency on Core 0, which maintained nearly full utilization (99.94%). The average turnaround time was reasonable at 329.80 time units, showcasing effective single-core scheduling.
However, this test case illustrates a structural limitation:
Linear task graphs inherently block parallel execution, negating the benefits of a multi-core scheduler. The lack of task branching or independence prevents the exploitation of DAG-based optimization and RMS-driven preemption.
As a result, average core utilization dropped significantly (33.31%), despite one core being fully active.

## 5. COMPARISON WITH EXISTING SCHEDULING ALGORITHMS LIKE FCFS AND ROUND ROBIN

We performed a comparative analysis of our algorithm with existing scheduling algorithms like FCFS and Round Robin.
Although, since our algorithm supports parallel execution, it also handles task dependencies which can affect its efficiency.
For fair comparison, we maintained identical task execution times across all algorithms.

## 5.1 Metrics for Comparison

- Average Turnaround Time: The average time from task submission to completion
- Core Utilization: The percentage of time each core spends actively executing tasks

## 5.2 DAG Set-Up

In our algorithm we simulated 10 tasks with four cores and time quantum=100 (used for preemption). This particular DAG is the sample DAG in our code.

**Table 10: Task Details of Sample DAG**

| ID | Name | Duration | Period | Priority | Dependencies |
|---|---|---|---|---|---|
| 0 | Task0 | 172 | 500 | 6 | None |
| 1 | Task1 | 185 | 200 | 9 | 0 |
| 2 | Task2 | 252 | 800 | 3 | 0 |
| 3 | Task3 | 91 | 300 | 8 | 1 |
| 4 | Task4 | 120 | 250 | 8 | 1 |
| 5 | Task5 | 138 | 350 | 7 | 2 |
| 6 | Task6 | 47 | 150 | 9 | 3 4 |
| 7 | Task7 | 65 | 400 | 7 | 5 |
| 8 | Task8 | 185 | 600 | 5 | 6 |
| 9 | Task9 | 78 | 100 | 10 | 7 8 |

**Simulation completed in** : 708 time units.

### 5.3 Performance Metrics

**Table 11: Execution Results for Sample DAG**

| ID | Name | Duration | Period | Priority | Start | Finish | Turnaround |
|---|---|---|---|---|---|---|---|
| 0 | Task0 | 172 | 500 | 6 | 0 | 172 | 172 |
| 1 | Task1 | 105 | 200 | 9 | 172 | 277 | 105 |
| 2 | Task2 | 252 | 800 | 3 | 172 | 424 | 252 |
| 3 | Task3 | 91 | 300 | 8 | 277 | 368 | 91 |
| 4 | Task4 | 120 | 250 | 8 | 277 | 397 | 120 |
| 5 | Task5 | 138 | 350 | 7 | 424 | 562 | 138 |
| 6 | Task6 | 47 | 150 | 9 | 397 | 444 | 47 |
| 7 | Task7 | 65 | 400 | 7 | 562 | 627 | 65 |
| 8 | Task8 | 185 | 600 | 5 | 444 | 629 | 185 |
| 9 | Task9 | 78 | 100 | 10 | 629 | 707 | 78 |

**Average Turnaround Time:** 125.30 time units

**Table 12: Core Utilization Statistics for Sample DAG)**

| Core | Busy Time | Idle Time | Utilization % |
|---|---|---|---|
| 0 | 707 | 1 | 99.86% |
| 1 | 455 | 253 | 64.27% |
| 2 | 91 | 617 | 12.85% |
| 3 | 0 | 708 | 0.00% |

**Average Core Utilization:** 44.24%

*FCFS Metrics*

Then, we simulated 10 tasks with each having the same execution time with FCFS along with 4 cores.

Total Waiting Time: 2145 time units

Total Turnaround Time: 3398 time units

Average Waiting Time: 214.50 time units

Average Turnaround Time: 339.80 time units

*Round Robin Metrics*

For Round Robin, we used the same time quantum(100) for fair execution and preemption 4 cores.

Average Waiting Time: 111.40 time units

Average Turnaround Time: 236.70 time units

**5.4 Result Analysis**

**Table 13: Comparison Statistics**

| **Scheduling Algorithm** | **Avg. Turnaround Time** | **Core Utilization** |
|---|---|---|
| Our Algorithm (simulated with 4 cores) | 125.30 | Core 0: 99.86%, Core 1: 64.27%, Core 2: 12.85%, Core 3: 0.00% (Avg: 44.24%) |
| FCFS | 214.50 | Core 0: 30%, Core 1:30 %, Core 2: 20%, Core 3: 20% (Avg: 25%) |
| Round Robin | 236.70 | Core 0: 99.72%, Core 1:79.61 %, Core 2: 97.21%, Core 3: 73.46% (Avg: 87.50%) |

## 5.5 Key Observations

- **Superior Turnaround Time**: Despite handling complex dependencies, our RMS implementation achieved an average turnaround time of 125.30 time units, which is faster than both FCFS (214.50) and Round Robin (236.70).
- **Suboptimal Core utilization**: The average core utilization of 44.24% indicates that the system's computational resources weren't fully utilized. This is primarily due to the specific dependency structure of the DAG, which limits parallel execution opportunities. With careful analysis of the dependency graph and task characteristics, the number of cores could be optimized to match the DAG's inherent parallelism. For different DAG structures with higher parallelism potential, we would expect to see improved core utilization metrics (Test Cases 1 and 2 in Section 4).

## 5.6 Tackling Sub-Optimal Utilization with Careful Analysis of the DAG

We ran our algorithm with the same task and DAG structure but with 2 cores and got the following results:

**Simulation completed in** : 774 time units.

**Table 14: Execution Results for Sample DAG (with 2 cores)**

| ID | Name | Duration | Period | Priority | Start | Finish | Turnaround |
|---|---|---|---|---|---|---|---|
| 0 | Task0 | 172 | 500 | 6 | 0 | 172 | 172 |
| 1 | Task1 | 105 | 200 | 9 | 172 | 277 | 105 |
| 2 | Task2 | 252 | 800 | 3 | 172 | 449 | 277 |
| 3 | Task3 | 91 | 300 | 8 | 372 | 463 | 91 |
| 4 | Task4 | 120 | 250 | 8 | 277 | 397 | 120 |
| 5 | Task5 | 138 | 350 | 7 | 449 | 587 | 138 |
| 6 | Task6 | 47 | 150 | 9 | 463 | 510 | 47 |
| 7 | Task7 | 65 | 400 | 7 | 587 | 652 | 65 |
| 8 | Task8 | 185 | 600 | 5 | 510 | 695 | 185 |
| 9 | Task9 | 78 | 100 | 10 | 695 | 773 | 78 |

**Average Turnaround Time:** 127.80 time units

**Table 15: Core Utilization Statistics for Sample DAG (with two cores)**

| Core | Busy Time | Idle Time | Utilization % |
|---|---|---|---|
| 0 | 730 | 44 | 94.32% |
| 1 | 523 | 251 | 67.57% |

**Average Core Utilization:** 80.94%

The improved utilization results with 2 cores confirms that the DAG structure for this particular task set has an inherent parallelism factor closer to 2 rather than 4. By analyzing the dependency graph and identifying the maximum number of tasks that can run concurrently given the dependency constraints, we can determine the optimal core count for any given workload.

## 6.CONCLUSION

The scheduler successfully implemented RMS for priority assignment, ensuring that tasks with shorter periods were given higher priority. However, in scenarios where tasks had uniform periods (such as in the test case with all tasks having a period of 0 ms), RMS had minimal impact on the execution order. This indicated that RMS can provide significant advantages when tasks with varying periods are involved, but the scheduler needs to handle scenarios with homogeneous task periods more efficiently.

The system effectively honored the dependencies within the Directed Acyclic Graph (DAG), ensuring that tasks were executed only after their prerequisites were completed.

While the project successfully demonstrated its capabilities in task scheduling, certain limitations were identified during the test cases. For example, the uniform task setup in one test case minimized the impact of RMS, and task execution led to suboptimal load balancing across cores.

While the scheduler guarantees correctness, priority adherence, and dependency handling, its full potential, such as dynamic core distribution and concurrent execution, is realized only with more complex DAGs featuring branching or parallel paths.

## 7.REFERENCES

- Efficient Process Scheduling for Multi-core Systems

  https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9799455
- Scheduling Parallel Real-Time Tasks for Multicore Systems

  https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9283709
- DAG Scheduling and Analysis on Multi-Core Systems by Modelling Parallelism and Dependency

  https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9779935
- Parallel Path Progression DAG Scheduling

  https://arxiv.org/pdf/2208.11830
- Efficiently Scheduling Parallel DAG Tasks on Identical Multiprocessors

  https://arxiv.org/pdf/2410.17563