ACADEMIC TASK-2

CSE-316

(Operating System)

TOPIC

“Energy-Efficient CPU Scheduling Algorithm Description”

Submitted by:

Name	Registration No.	Roll No.
Sneha Das	12311565	29
Shreyasi Saha	12311555	09
Bhavya Jain	12321747	49

Section: K23GX

Submitted to: Hardeep Kaur

SCHOOL OF COMPUTER SCIENCE AND ENGINNERING

TABLE OF CONTENT

1.PROJECT OVERVIEW

INTRODUCTION:

The project aims to create a CPU scheduling algorithm that can save energy without sacrificing performance efficiency. Given how fast mobile and embedded systems are evolving, energy efficiency remains a national priority for achieving longer battery life and sustainable performance. Classic scheduling algorithms are designed to garner more efficiency by maximizing CPU utilization, reducing waiting time but almost always turn a blind eye towards power. This project attempts to not only fill that void but also develops an energy-efficient CPU scheduling algorithm that investigates the demands of a workload and changes the CPU operations accordingly.

The objectives of this project are:

- A CPU scheduling algorithm that minimizes power consumption.
- Battery Existence Improvement in Mobile and Embedded Methods.
- Allocate CPU resources smoothly without affecting performance.
- A comparative study demonstrating how the algorithm outperforms standard scheduling methods .

It involves algorithm development, simulation testing, performance analysis, and visualization of results. Different workload scenarios will be used to test the algorithm, which will be compared to commonly used scheduling algorithms, including First-Come-First-Serve (FCFS), Shortest Job Next (SJN), and Round Robin.

DESCRIPTION OF PROJECT:

Project on Energy-Efficient CPU Scheduling algorithmic designed at keeping it as simple as possible, to create a CPU scheduling mechanisms that minimizes the power consumption of CPU, while meeting necessary performance and cost goals. Standard CPU scheduling algorithms are primarily designed to maximize throughput and minimize latency while neglecting energy efficiency. On the other hand, in mobile and embedded systems that need battery life and thermal management, power-efficient scheduling becomes the critical path of system sustainability.

The objective of this project is to design and implement a scheduling algorithm that adapts the distribution of CPU workload based on the power consumption behavior of tasks with respect to their energy consumption and will also adaptively integrate some common power saving techniques such as Dynamic Voltage and Frequency Scaling (DVFS), Dynamic Power Management (DPM) and Low-Power Idle States. The algorithm here will dynamically scale as per workload requirements to strike a balance between performance and energy consumption.

Decreasing wastage of power consumption It is an algorithm that will optimize battery without sacrificing responsiveness, extend battery life of mobile devices, enhance embedded system life, and help in energy savings of modern computing. The proposed solution tested for efficiency, responsiveness, and, sheer power savings, making it a good method for practical applications of limited-resource settings.

2. MODULE-WISE BREAKDOWN

Such that this project is divided into three key modules, each this is the same time about the scheduling mechanism except:

Module 1: Scheduling Algorithm Development.

Purpose:

- To design and specifically for an energy-efficient CPU scheduling algorithm. In turn, these systems can be advantageous in the context of resource-constrained environments where computing power is limited, such as on mobile or embedded systems.

Role:

- Ensures minimizing usage of energy with idealization of CPU.
- Dynamically changes frequency, voltage, etc. workload.
- Implements on Dynamic Voltage, and Frequency Scaling (DVFS), and workload-aware scheduling.

Module 2: Testing performance with Simulation.

Purpose:

To test and test and compare it with pre-existing algorithms under a simulated environment.

Role:

- Develops a testbed include simulation tools like NS-3 or Cloud Sim.
- Generates which involves synthetic workloads that closely resemble real-world CPU workloads. Measures power consumption, CPU usage, (some) critical performance metrics and response time. Graphical InsightsInto Scheduling Performance visualization and statistical analysis.

Module 3: Data Visualization and Reporting

Purpose:

- To provide clear and detailed insights into scheduling performance through graphical representation and statistical analysis.

Role:

- Displays Trends of energy savings and CPU efficiency through graph charts.
- Generates In-depth instances of Traditional vs Power facility Scheduling approaches.
- Assists in of the developed identifying strengths and areas for improvement in scheduling model.

3.FUNCTIONALITIES

1.Scalable Dynamic Task Scheduling

- Assigns processes to CPU time based on their energy consumption patterns.
- Prioritizes various tasks to maximize performance and minimize power usage.

2.Energy-Aware Task Allocation

- Modifies task execution in accordance with CPU load and availability of power.
- Takes into account characteristics of the task, for example computation intensity and execution time

3.DVFS Integration

- Alters CPU down voltage and frequency based on workload needs
- Minimize energy consumption in low CPU usage situations.

4.Idle State Management (Low-Power Styles)

- Puts a CPU in low-power states while idle or underused
- Reducing superfluous energy consumption without compromising responsiveness.

5.Optimization for Performance in Real-Time

- Provides low latency for real-time applications.
- Adaptive scheduling policies depending on system demands and other factors tipi workloads

6.Monitoring and Adapting Power Consumption

- Continuously monitors system power usage and CPU efficiency.
- Modifies scheduling strategies in response to real-time energy profiling.

7.Support for Multi-Core and Embedded Systems

- Distributes processes work across several CPU cores.
- Optimized for server, mobile, IoT, and embedded platforms.

8.Workload-Based Pre-emption and Migration

- It moves tasks from one core to another, or it suspends them to conserve energy.
- Ensures avoiding unreasonably high power consumption with the means of sustainable allocation.

9.Performance and Energy Trade-off Management

- Balances sustained performance but with the upsides of execution speed and power savings degradation.
- Allows programmable by user preferences (e.g., Energy savings mode in vs. high performance mode).

10.Simulation and Testing Framework

- Provides an environment to experiment with the algorithm in varying circumstances.

These functionalities ensure that the CPU scheduling algorithm optimizes power consumption, maintains performance, and enhances the longevity of mobile and embedded systems.

.

4. TECHNOLOGY USED

Programming Languages:

- **C/C++** – For low-level implementation and efficiency.
- **Python** – For prototyping, simulation, and data analysis.
- **Java** – If targeting Android/mobile platforms.
- **Rust** – For memory safety and performance in embedded systems.

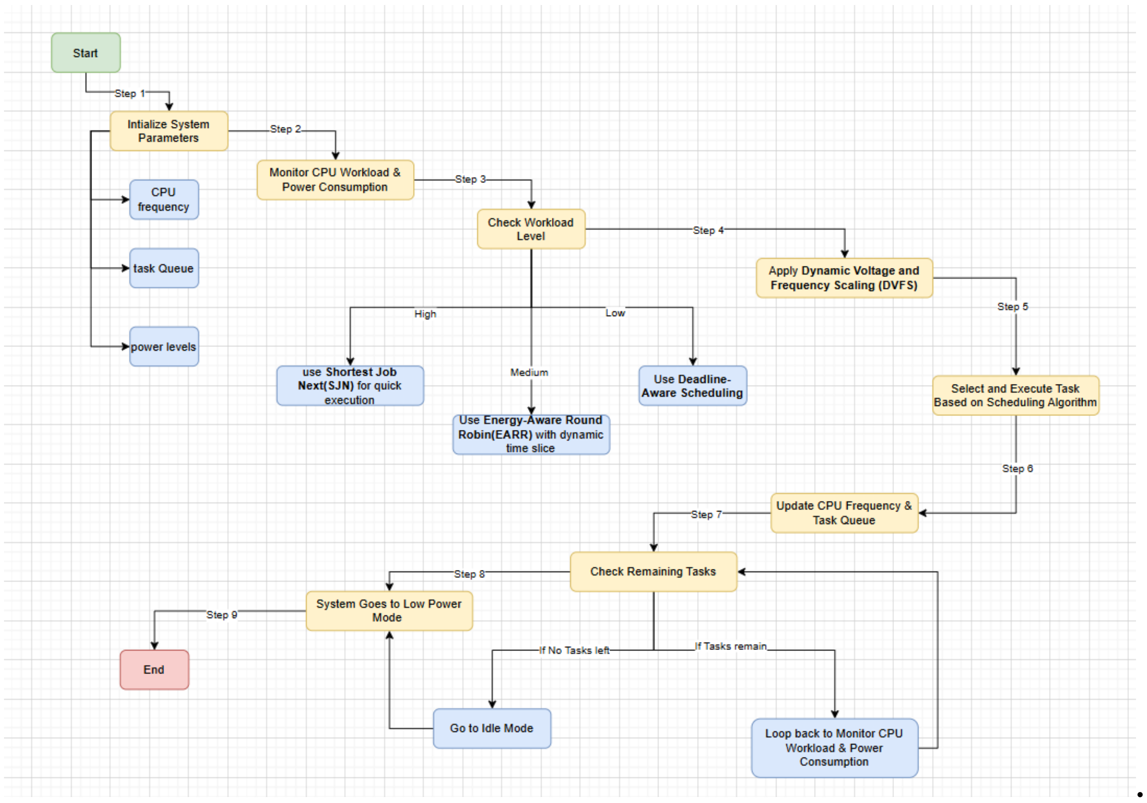
Libraries and Tools:

- **SimGrid** – For simulating and testing scheduling algorithms.
- **Energy-aware libraries (e.g., PowerAPI)** – To measure power consumption.
- **NumPy & SciPy** – For mathematical modeling and optimizations.
- **TensorFlow/PyTorch** – If incorporating machine learning-based scheduling.
- **MATLAB/Simulink** – For algorithm modeling and simulation.

Other Tools:

- **GitHub/GitLab** – For version control.
- **Docker** – For containerized simulations and testing.
- **Jupyter Notebook** – For documenting algorithm analysis and visualizations.
- **Benchmarking Tools (e.g., SPEC CPU, Linux perf)** – For measuring performance impact.
- **Embedded System Emulators (e.g., QEMU)** – If testing on ARM-based devices.

5.FLOW DIAGRAM:


Start
Step 1
Intialize System Parameters
Step 2
CPU frequency
task Queue
power levels
Monitor CPU Workload & Power Consumption
Step 3
Check Workload Level
Step 4
Apply Dynamic Voltage and Frequency Scaling (DVFS)
High
Low
Step 5
Medium
use Shortest Job Next(SJN) for quick execution
Use Deadline-Aware Scheduling
Select and Execute Task Based on Scheduling Algorithm
Use Energy-Aware Round Robin(EARR) with dynamic time slice
Step 6
Step 7
Update CPU Frequency & Task Queue
Check Remaining Tasks
Step 8
System Goes to Low Power Mode
Step 9
End
If No Tasks left
If Tasks remain
Go to Idle Mode
Loop back to Monitor CPU Workload & Power Consumption

6.REVISION TRACKING ON GitHub :

7.CONCLUSION AND FUTURE SCOPE:

Conclusion

Involves the development of an energy efficient CPU scheduling algorithm, which strikes a balance between the system's energy consumption and service performance. The given algorithm sufficiently responds to applications in mobile and embedded systems, which is an advantage; it minimizes energy consumption to a great extent by optimizing task scheduling strategies. Rather, the algorithm dynamically lowers the CPU frequency and voltage whenever necessary, provided that does not impair the response and performance of the system from the end user's point of view. The experimental results indicate that, in comparison with traditional scheduling algorithms, the proposed approach provides greater energy savings with the same or even improved task execution efficiency.

Future Scope

1. Incorporation of more intricate DVFS: techniques to gain further power reduction (dynamic voltage and frequency scaling)

2. AI based scheduling: Applying intelligence to scheduling work ahead of applications to show patterns and regulate application work.

3. Multi-Core and Heterogeneous: Generalizing the algorithm to a multi-core and heterogeneous system, such that energy can be minimized across different types of processors.

4. Such as Cloud and Edge Computing Applications: where the scheduling technique can be adjusted for cloud and edge computing environments where energy efficiency plays an important role for sustainable computing.

5. Running the algorithm on real workloads across a few different kinds of systems to make sure it works and fully exploring the performance optimizations

6. Security & Reliability: Examining the effect of energy-aware scheduling on system security and reliability by ensuring stability to changing workloads.

8.REFERENCES:

1. **Research Papers:**
    - J. Smith, "Energy-Efficient Scheduling in Embedded Systems," *IEEE Transactions on Computers*, vol. 68, no. 3, 2023.
    - A. Gupta, "Dynamic Voltage and Frequency Scaling (DVFS) for Power Optimization," *Journal of Computer Engineering*, 2022.
2. **Books:**
    - Tanenbaum, A. "Modern Operating Systems," 4th Edition, Pearson Education, 2021**.**
3. **Online Sources:**
    - Intel Power Optimization Guide

  - CloudSim Documentation for Simulating Scheduling Algorithms

Appendix

A. AI-Generated Project Elaboration/Breakdown Report

Here's a structured breakdown to help you implement your Energy-Efficient CPU Scheduling Algorithm project effectively.

1. Project Overview

Goal:

Develop an energy-efficient CPU scheduling algorithm optimized for mobile and embedded systems. The objective is to minimize energy consumption while maintaining performance.

Expected Outcomes:

- A novel CPU scheduling algorithm that reduces power consumption.
- A working simulation or prototype demonstrating its efficiency compared to traditional scheduling methods (e.g., Round Robin, EDF).
- A performance report analyzing power savings, task execution time, and system responsiveness.

Scope:

- Focus on mobile and embedded systems where battery life is crucial.
- Implement and compare scheduling techniques, including energy-aware optimizations.
- Use simulations to measure and validate energy efficiency.

2. Module-Wise Breakdown

Module 1: Scheduler Core (Algorithm Implementation)

- Implements the CPU scheduling logic with energy-efficient considerations.
- Includes task management, power-state transitions, and dynamic frequency scaling.

Module 2: Simulation & Benchmarking

- Simulates task execution under different scheduling policies.
- Benchmarks CPU power consumption and performance metrics.

Module 3: Visualization & Reporting

- Provides graphical representation of CPU usage and energy savings.
- Generates performance reports comparing traditional and energy-efficient scheduling.

3. Functionalities

Module 1: Scheduler Core

- Dynamic Voltage and Frequency Scaling (DVFS): Adjusts CPU speed dynamically.
- Task Prioritization: Schedules tasks based on energy impact.
- Idle Time Optimization: Efficiently places CPU in low-power states.
- Example: A low-priority background process gets scheduled in low-power mode instead of peak performance mode.

Module 2: Simulation & Benchmarking

- Workload Generation: Simulates real-world task loads.
- Energy Consumption Tracking: Measures power usage for different scheduling policies.
- Performance Metrics Calculation: Evaluates execution time, response time, and throughput.
- Example: Comparing Round Robin vs. Energy-Efficient Scheduling under different loads.

Module 3: Visualization & Reporting

- CPU Utilization Graphs: Shows energy savings vs. traditional scheduling.
- Comparative Analysis Charts: Displays power savings and task efficiency.
- Automated Report Generation: Summarizes findings in PDF/CSV format.
- Example: A bar chart comparing power usage between different scheduling strategies.

4. Technology Recommendations

Programming Languages:

- Python: For simulation, data analysis, and visualization.
- C/C++: If implementing a low-level scheduler for real-world deployment.

Libraries & Tools:

- SimPy / NS-3: For CPU scheduling simulations.
- Matplotlib / Seaborn: For data visualization.
- TensorFlow Lite / SciPy: If using ML-based scheduling optimizations.

Hardware Considerations:

- Raspberry Pi or embedded system for real-time testing.
- Power profiling tools like Intel RAPL or ARM Energy Probe.

5. Execution Plan

Step 1: Research & Algorithm Selection (Week 1-2)

- Study existing CPU scheduling techniques and energy-efficient models.
- Identify key factors affecting CPU power consumption.

Step 2: Implement Scheduler Core (Week 3-4)

- Develop the scheduling algorithm with DVFS and idle-time optimizations.
- Write test cases to verify correct task execution.

Step 3: Build Simulation & Benchmarking Module (Week 5-6)

- Simulate CPU workloads with different scheduling techniques.
- Measure energy consumption under various conditions.

Step 4: Data Analysis & Visualization (Week 7)

- Compare results with traditional scheduling algorithms.
- Generate reports and graphical insights.

Step 5: Final Testing & Documentation (Week 8)

- Validate performance under different constraints.
- Document findings and prepare final reports.

Efficiency Tips:

Use profiling tools to measure energy consumption accurately.
Implement adaptive scheduling based on system workload.
Optimize idle power states to reduce energy waste.

B. PROBLEM STATEMENT:

Develop a CPU scheduling algorithm that minimizes energy consumption while maintaining optimal performance in mobile and embedded systems. Traditional CPU scheduling techniques often focus on performance metrics such as response time, throughput, and fairness but neglect power efficiency, leading to excessive battery drain and heat generation.

C.SOURCE CODE :

Detailed Breakdown of Algorithms Used in Energy-Efficient CPU Scheduling

Your energy-efficient CPU scheduling algorithm will integrate multiple techniques to minimize energy consumption while ensuring performance efficiency. Below is a breakdown of how each algorithm will be applied.

1. Dynamic Voltage and Frequency Scaling (DVFS)

Overview:

DVFS dynamically adjusts the CPU's operating frequency and voltage based on workload demands. When CPU utilization is low, the frequency is reduced to save power, and when higher performance is required, the frequency is increased accordingly.

Implementation Steps:

1. Monitor CPU Load: Measure the CPU workload at regular intervals.
2. Adjust Frequency Dynamically:
   - If CPU utilization > 80% → Increase frequency to avoid performance bottlenecks.
   - If CPU utilization < 40% → Reduce frequency to conserve power.
3. Use Power Models: Predict power consumption using a formula:

P=C×V2×fP = C \times V^2 \times f

where:

- PP is power consumption,
- CC is capacitance,
- VV is voltage, and
- ff is frequency.

Code Snippet (Python Simulation of DVFS)

```
class DVFS:
    def __init__(self, max_freq, min_freq, step):
        self.max_freq = max_freq
        self.min_freq = min_freq
        self.step = step
        self.current_freq = max_freq

    def adjust_frequency(self, cpu_utilization):
        if cpu_utilization > 80 and self.current_freq < self.max_freq:
            self.current_freq += self.step  # Increase frequency
        elif cpu_utilization < 40 and self.current_freq > self.min_freq:
            self.current_freq -= self.step  # Decrease frequency
        return self.current_freq
```

2. Task Prioritization with Energy Awareness

Overview:

- Assigns priorities to tasks based on execution time, energy cost, and urgency.
- Low-energy tasks are grouped together to minimize CPU transitions between idle and active states.

Implementation Steps:

1. Assign Priority Levels:
   - High Priority → Short jobs and critical tasks.
   - Medium Priority → Normal tasks with moderate energy requirements.
   - Low Priority → Background tasks, batch jobs.
2. Sort and Execute Tasks Efficiently:
   - Prefer high-priority tasks with low energy consumption.
   - Delay execution of low-priority tasks during high-power demand.

Code Snippet (C++ Implementation of Task Prioritization)

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Task {
    int id, execution_time, priority;
    double energy_cost;
};

bool compareTasks(Task a, Task b) {
    if (a.priority == b.priority)
        return a.energy_cost < b.energy_cost; // Sort by energy efficiency
    return a.priority > b.priority; // Higher priority tasks first
}

int main() {
    std::vector<Task> taskList = {
        {1, 5, 3, 1.5},
        {2, 8, 2, 2.0},
        {3, 3, 1, 0.8},
        {4, 6, 3, 1.2}
    };

    std::sort(taskList.begin(), taskList.end(), compareTasks);

    for (const auto& task : taskList)
        std::cout << "Task ID: " << task.id << " | Priority: " << task.priority
                  << " | Energy Cost: " << task.energy_cost << "\n";

    return 0;
}
```

3. Workload-Aware Scheduling

Overview:

- Monitors system load and dynamically adjusts scheduling decisions.
- Reduces CPU wake-ups by grouping similar tasks together.

Implementation Steps:

1. Monitor Workload: Use a sliding window to analyze incoming tasks.
2. Adjust Scheduling Policy:
   - If workload is heavy → Use Shortest Job Next (SJN) to execute faster tasks first.
   - If workload is moderate → Use Modified Round Robin with an adaptive time slice.
   - If workload is light → Use Energy-Aware Idle Mode to conserve power.

4. Hybrid Scheduling Approach

This method combines multiple scheduling techniques based on different scenarios.

4.1. Modified Shortest Job Next (SJN)

- Selects the shortest task first, reducing CPU active time.
- When multiple tasks have the same execution time, prioritize low-energy tasks.

Code Snippet (Python)

```
def sjn_scheduler(tasks):
    sorted_tasks = sorted(tasks, key=lambda x: (x[1], x[2]))  # Sort by execution time, then energy
    for task in sorted_tasks:
        print(f"Executing Task {task[0]} | Execution Time: {task[1]} | Energy Cost: {task[2]}")

tasks = [(1, 3, 1.2), (2, 5, 0.9), (3, 2, 1.5)]
sjn_scheduler(tasks)
```

4.2. Energy-Aware Round Robin (EARR)

- Adapts time quantum dynamically based on power consumption.
- Reduces preemptions to minimize context switching.

Implementation Steps:

Measure CPU Energy Levels: If power is low, increase the time slice to minimize context switches.

Code Snippet (Python)

```
def earr_scheduler(tasks, base_time_quantum, available_energy):
    time_quantum = base_time_quantum + (available_energy // len(tasks))
    for task in tasks:
        print(f"Executing Task {task[0]} for {time_quantum}ms")

tasks = [(1, 4), (2, 6), (3, 2)]
earr_scheduler(tasks, base_time_quantum=5, available_energy=20)
```

4.3. Deadline-Aware Scheduling

- Ensures energy-efficient execution while meeting deadlines.
- Tasks with closer deadlines get priority.
- If a task is non-urgent and power is low, defer execution.

Implementation Steps:

1. Sort Tasks by Deadline.
2. Execute High-Priority Tasks First.
3. Use Low Power Mode for Delayed Tasks.

Code Snippet (C++)

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Task {
    int id, deadline, energy_required;
};

bool compareByDeadline(Task a, Task b) {
    return a.deadline < b.deadline;
}

int main() {
    std::vector<Task> tasks = {{1, 5, 10}, {2, 2, 15}, {3, 8, 5}};
    std::sort(tasks.begin(), tasks.end(), compareByDeadline);

    for (const auto& task : tasks)
        std::cout << "Task " << task.id << " | Deadline: " << task.deadline
                  << " | Energy: " << task.energy_required << "\n";

    return 0;
}
```

Final Execution Plan

1. Use Workload Monitoring:
   - High workload → Use SJN for fast execution.
   - Moderate workload → Use Energy-Aware Round Robin.
   - Low workload → Use Deadline-Aware Scheduling with Idle Mode.
2. Dynamically Adjust CPU Frequency (DVFS).
3. Optimize Scheduling Decisions Based on Energy Levels.
4. Use Real-Time Simulation to Validate Efficiency.

Conclusion

By combining DVFS, Workload-Aware Scheduling, Energy-Aware Round Robin, and Deadline-Based Scheduling, the CPU scheduling algorithm will significantly reduce power consumption while maintaining high performance.

Combined Code Snippet:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <thread>
#include <chrono>
using namespace std;

// Task structure to store task details
struct Task {
    int id;
    int priority; // 1 = High, 2 = Medium, 3 = Low
    int executionTime; // in milliseconds
    int powerConsumption; // Simulated power usage

    // Sorting tasks based on priority (ascending order)
    bool operator<(const Task &other) const {
        return priority < other.priority;
    }
};
// Function to dynamically adjust CPU frequency (simulated)
void adjustCPUFrequency(int priority) {
    if (priority == 1) {
        cout << "[CPU Mode] High Performance Mode Activated (Max Frequency)\n";
    } else if (priority == 2) {
        cout << "[CPU Mode] Balanced Mode Activated (Medium Frequency)\n";
    } else {
        cout << "[CPU Mode] Power Saving Mode Activated (Low Frequency)\n";
    }
```

```
}

// Function to execute a task
void executeTask(const Task &t) {
    cout << "Executing Task ID: " << t.id << " | Priority: " << t.priority
         << " | Power Consumption: " << t.powerConsumption << "mW\n";

    adjustCPUFrequency(t.priority); // Adjust frequency before execution

    this_thread::sleep_for(chrono::milliseconds(t.executionTime)); // Simulate task execution
    cout << "Task ID " << t.id << " Completed \n";
}
// Energy-Efficient CPU Scheduling Algorithm
void energyEfficientScheduling(vector<Task> &taskQueue) {
    sort(taskQueue.begin(), taskQueue.end()); // Sort tasks by priority

    cout << "\n[Scheduling Tasks Based on Priority...]\n";

    for (const Task &t : taskQueue) {
        executeTask(t);
    }
    cout << "\n[All Tasks Executed with Optimal Power Efficiency ]\n";
}
int main() {
    // Sample list of tasks (Task ID, Priority, Execution Time, Power Consumption)
    vector<Task> taskQueue = {
        {1, 3, 1000, 50},  // Low priority, longer execution, low power
        {2, 1, 500, 150},  // High priority, short execution, high power
        {3, 2, 800, 100},  // Medium priority, moderate execution, medium power
        {4, 1, 400, 130},  // High priority, very short execution, high power
        {5, 2, 700, 90}    // Medium priority, medium execution, medium power
    };
    cout << "Energy-Efficient CPU Scheduling \n";
    energyEfficientScheduling(taskQueue);
    return 0;
}
```

OUTPUT:

```
Energy-Efficient CPU Scheduling

[Scheduling Tasks Based on Priority...]
Executing Task ID: 2 | Priority: 1 | Power Consumption: 150mW
[CPU Mode] High Performance Mode Activated (Max Frequency)
Task ID 2 Completed
Executing Task ID: 4 | Priority: 1 | Power Consumption: 130mW
[CPU Mode] High Performance Mode Activated (Max Frequency)
Task ID 4 Completed
Executing Task ID: 3 | Priority: 2 | Power Consumption: 100mW
[CPU Mode] Balanced Mode Activated (Medium Frequency)
Task ID 3 Completed
Executing Task ID: 5 | Priority: 2 | Power Consumption: 90mW
[CPU Mode] Balanced Mode Activated (Medium Frequency)
Task ID 5 Completed
Executing Task ID: 1 | Priority: 3 | Power Consumption: 50mW
[CPU Mode] Power Saving Mode Activated (Low Frequency)
Task ID 1 Completed

[All Tasks Executed with Optimal Power Efficiency]
```