University of Essex

A Hardware and Software Task-Scheduling Framework Based on CPU+GPU Heterogeneous Architecture in Edge Computing Devices

Harshini Bangalore Amarnath

2311849

A thesis submitted for the degree of Mater of Science in Electronic Engineeering

Supervisor: **Dr. Sangeet Saha**

School of Computer Science and Electronic Engineering
University of Essex

August 27, 2024

Abstract

In the world of heterogeneous real-time systems, balancing result accuracy with power and time constraints presents a significant challenge. To combat the challenge, the researchers have proposed approximate computing tasks where the applications can be divided into two components, a mandatory part that produces an acceptable result, followed by an optional part that enhances the result's accuracy. This project proposes a scheduling algorithm for real-time dependent tasks on CPU-GPU based heterogeneous systems. The objective of the scheduling remains to maximize accuracy while considering both deadlines and power constraints. Given real-time applications, which are represented as Direct Acyclic Graph (DAG) of tasks, the algorithm topologically sorts tasks and identifies critical high (CH), and critical low (CL) tasks based on their execution time and task dependencies. These tasks are then scheduled appropriately across the Graphic Processing Unit (GPU) and Central Processing Unit (CPU) by considering power consumption while ensuring that all tasks meet the deadline. The effectiveness of this algorithm is evaluated by measuring the achieved result accuracy without exceeding power limits. Implementation of task scheduling on the NVIDIA Jetson Nano, which includes both CPU and GPU, demonstrates the practicality of the approach. Additionally, the results highlight the advantages of using heterogeneous systems over homogeneous systems, with an NQ of 81.1% being achieved.

Keywords- Real-time systems, heterogeneous systems, power consumption, task scheduling

Acknowledgement

I would like to express my deepest gratitude my project supervisor to Dr. Sangeet Saha, a Lecturer (Assistant Professor) at the School of Computer Science and Electronic Engineering, University of Essex, for his invaluable guidance throughout this project. His support was instrumental not only in helping me accomplish the tasks at hand but also in deepening my understanding of the underlying concepts essential to my research.

I would also like to extend my sincere thanks to Dr. Pal, a Senior Research Officer at the School of Computer Science and Electronic Engineering, University of Essex, for his guidance regarding the Jetson Nano and its usage. His expertise was crucial in navigating the technical aspects of this project.

Finally, I would like to thank the School of Computer Science and Electronic Engineering at the University of Essex for their unwavering support, providing the resources and environment necessary for the successful completion of this research.

Contents

CHAPTER 1

Introduction

Heterogeneous multicore systems, which integrate various types of processors such as CPUs, GPUs, ASICs, and FPGAs, are increasingly prevalent across a wide range of technological fields, particularly in time-critical systems. These systems leverage specialized hardware components, known as accelerators, which are designed to perform specific computational tasks more efficiently than general-purpose CPUs. Accelerators are essential because they offload and speed up demanding tasks like parallel processing, encryption, and data compression, resulting in faster overall processing and reduced power consumption [1]. In edge computing environments, where real-time data processing and energy efficiency are paramount, the use of accelerators is crucial to meet the escalating demands for computational power and to manage the growing volume of data effectively. Heterogeneous task scheduling algorithm are being used in wide applications, such as optimizing allocation of tasks in cloud computing devices [2], transport-pick agents task scheduling (TPTS) problem [3], fog computing (FC) and cloud computing [4].

One of the biggest challenges in heterogeneous multicore systems is achieving accurate and reliable performance while adhering to strict power and timing constraints. This is especially critical in real-time systems where the timing of task execution can be a matter of life and death. Task scheduling in these environments becomes even more complex when dealing with real-time applications. Real-time applications represented as Directed Acyclic Graphs (DAGs) consist of tasks (nodes) with specific execution

orders defined by dependencies (edges) between them. The DAG structure ensures that tasks are executed in a sequence that respects these dependencies, crucial for maintaining the timing and performance requirements of real-time systems [5].

To address these challenges, researchers have introduced the concept of approximate computing, where tasks are divided into two components. The first component, the mandatory part, ensures that the system meets its fundamental requirements by producing a minimally acceptable result within the specified deadline. The second component, the optional part, enhances the result's accuracy when additional resources and time are available. In this context, each task can have multiple versions, each representing different execution levels with varying degrees of accuracy and execution time. These task versions enable the system to balance accuracy with timing and power constraints by selecting the most suitable version based on the available resources. This approach optimizes the Quality of Service (QoS) while ensuring that deadlines are consistently met [6].

The effectiveness of this approach is demonstrated through its implementation on the NVIDIA Jetson Nano, a platform that includes both CPU and GPU [7]. The results of this implementation highlight not only the practicality of the proposed algorithm but also the inherent advantages of using heterogeneous systems over homogeneous ones. The introduction of this scheduling algorithm offers a promising approach to managing the complexities of task scheduling in heterogeneous real-time systems. More detailed explation is mentioned in 2.5. By effectively leveraging the unique strengths of different processing units and incorporating considerations of power and time constraints, this work contributes to the ongoing efforts to enhance the performance and reliability of critical real-time applications.

1.1 Problem definition

In today's complex computing environments, particularly in heterogeneous multicore systems, efficient task scheduling is critically challenging. These tasks must handle real-time applications where tasks have varying execution requirements and multiple versions, each with different computational demands and power consumptions.

The challenge lies in optimally assigning tasks to either the CPU or GPU while ensuring that all tasks and the scheduling process carefully balance performance and efficiency. The need to manage these constraints effectively supports the development of advanced scheduling algorithms that can navigate these complexities to optimize system performance, making this an essential area of research in real-time systems.

Given a set of tasks T represented as a Directed Acyclic Graph (DAG), which is a common representation of real-time systems, this DAG has different nodes or threads represented as T_i where i represents the task number that ranges from 1 to T_L, that is $i = \{1, 2, 3, \ldots, T_L\}$. The system under consideration is a multicore heterogeneous computing environment, specifically the NVIDIA Jetson Nano, which integrates both CPU and GPU capabilities. [6] Each task T_i may have multiple versions (signifying different degrees of accuracy) denoted as T_i^j, where j represents the specific version of the task, with j ranging from 0 to T_L, that is $j = \{0, 1, 2, \ldots, T_L\}$, indicating that a task can have one or more versions, or in some cases, no versions at all. Each task version T_i^j is characterized by its mandatory cycles M_i^j and optional cycles O_i^j, which define the essential and additional computation requirements of the task, respectively. The power consumption associated with these nodes is represented by the power factors Pow_i^j, L for critical low (C_L) tasks and Pow_i^j, H for critical high (C_H) nodes. The efficiency of [8] executing each task version in these critical modes is described by Y_i^j, L for C_L and Y_i^j, H for C_H. Additionally, each task must adhere to a specified deadline D, which defines the maximum allowable time for task completion. This structure presents a complex challenge in scheduling tasks in heterogeneous systems to optimize both performance and power consumption while ensuring all tasks meet their deadlines.

This research aims to develop an efficient scheduling algorithm for assigning tasks T_i in a heterogeneous computing system, specifically utilizing either the CPU or GPU of an NVIDIA Jetson Nano, within a given deadline D while optimizing power consumption Pow. The approach involves topologically sorting the tasks T_i to establish their execution order based on dependencies. Subsequently, the algorithm identifies critical low (C_L) and critical high (C_H) tasks by analyzing their execution times, represented as ET_i^j, L and ET_i^j, H for each task version T_i^j. By categorizing tasks into C_L and C_H, the algorithm can effectively manage power consumption and ensure that all tasks are

completed within the specified deadline. This scheduling method seeks to balance computational efficiency and power usage, leveraging the strengths of the heterogeneous system to achieve optimal performance.

Additionally, the problem involves optimizing task scheduling in heterogeneous multicore systems, where the number of available cores significantly impacts the Quality of Service (QoS) delivered by the system. The proposed solution will explore current methodologies and leverage topological sorting, along with the identification of critical low and high tasks, to ensure efficient task execution within stringent deadlines. By carefully balancing computational efficiency and power usage, this work seeks to contribute a robust approach to managing real-time tasks in heterogeneous computing environments, ultimately enhancing system reliability and performance.

1.2 Scope

The main goals of the thesis were:

- To ensure tasks are completed within the specified deadlines in real-time heterogeneous systems, particularly on both CPU and GPU components.
- To optimize the scheduling of time-critical, dependent tasks by balancing the trade-offs between execution speed and power consumption, ensuring that the system remains within specified power constraints.
- To conduct a comparative analysis that highlights the advantages of using heterogeneous systems, such as those integrating both CPU and GPU, over traditional homogeneous systems in handling the complex computational tasks.
- To investigate how the scalability of core counts within a heterogeneous system can improve the system's capacity to achieve near-ideal Quality of Service (QoS) in real-time applications, thereby enhancing overall systems performance.
- To evaluate the benchmark performance of the proposed scheduling in real-time applications, using the NVIDIA Jetson Nano as the representative heterogeneous systems in handling complex computational tasks.

1.3 Limitations

The scheduling algorithm developed in this project effectively optimizes the execution of a single set of tasks with six nodes on a heterogeneous system, balancing trade-offs between execution speed and power consumption. However, the algorithm faces significant challenges when applied to more complex, real-world scenarios. In practical applications, systems are often required to manage multiple sets of tasks, each with numerous nodes and varying task versions. The complexity of scheduling these tasks simultaneously across multiple cores, especially when considering dependencies and different versions, dramatically increases the computational load. This increase in complexity makes the current algorithm less efficient and difficult to scale. As the number of tasks and nodes grows, the algorithm may struggle to find optimal solutions within reasonable time frames, limiting its usefulness in larger, more dynamic environments.

Another limitation arises from the use of Python programming for data analysis and scheduling execution. In this project, data is managed using dictionaries and matrix methods, which are adequate for smaller datasets but may become inefficient when dealing with large-scale data typically encountered in real-world applications. The computational resources required to handle large volumes of data increase significantly, and the current approach may not be sufficient to process and analyze this data in a timely manner.

Moreover, in practical scenarios, data is often generated continuously and in large quantities, necessitating more advanced tools for real-time processing. The use of basic programming techniques without the support of AI or machine learning can hinder the algorithm's ability to adapt to changing conditions and optimize performance dynamically. For instance, AI-driven approaches could better handle the continuous data streams, predict task behaviors, and adjust scheduling in real-time, which is beyond the capabilities of the current implementation. The reliance on traditional programming methods thus represents a limitation when considering the scalability and adaptability required for real-time applications in more complex and data-intensive environments.

1.4 Work undertaken

The key achievements of this project were:

- Successfully implemented a scheduling algorithm using Python that ensures task completion within real-time constraints on heterogeneous systems, specifically utilizing the CPU and GPU components effectively.
- Optimized the scheduling of time-sensitive, dependent tasks by balancing between deadline and power efficiency, thereby maintain adherence to strict power consumption limits.
- Provided a detailed comparative analysis showcasing the significant benefits of heterogeneous systems over homogeneous ones, particularly in managing complex and computationally intensive tasks.
- Explored and confirmed that increasing the number of cores within a heterogeneous system significantly enhances its ability to approach near-ideal Quality of Service (QoS) in real-time scenarios, contributing to improved system performance.

1.5 Structure of work

The structure of this report is organized as follows: Chapter 2 presents a comprehensive literature review, exploring key areas such as the role of heterogeneous systems, Directed Acyclic Graphs (DAGs), task scheduling, Jetson Nano, and power and energy optimization strategies. Chapter 3 explains the core methodology, detailing the design of the task scheduling algorithm and presenting the associated pseudocode. Chapter 4 discusses the results and observations derived from the project, highlighting the key findings. Finally, Chapter 5 provides an evaluation of the work completed, discussing its implications and suggesting directions for future research.

CHAPTER 2

Literature review

This chapter will cover all the fundamental background knowledge required to understand the research topic. The first section explores the advantages of heterogeneous systems, emphasizing their ability to optimize performance and energy efficiency by utilizing different types of processors, each designed to handle specific tasks effectively. The second section delves into the use of Directed Acyclic Graphs (DAGs) for representing real-time applications, which provide a clear structure for modeling task dependencies and execution sequences. The third section discusses real-time parallel tasks, focusing on the challenges of scheduling these tasks to meet strict timing requirements while maximizing the utilization of available resources. The final section examines algorithms used in CPU-GPU systems for task scheduling, highlighting the importance of efficiently distributing workloads across different processing units. Additionally, this chapter introduces the concept of task versions, which allow for varying levels of accuracy and execution time, offering a flexible approach to balancing performance and resource consumption in complex computing environments.

2.1 The Role of Heterogeneous Systems in Modern Computing

Heterogeneous systems offer distinct advantages over homogeneous systems, particularly when addressing the nonlinear impacts of core frequency and cache partitioning

on task execution, as highlighted in [9]. By leveraging the diverse strengths of different core types, such as performance-oriented and energy-efficient cores, these systems can achieve significant energy savings that would be challenging to realize with a homogeneous architecture. This makes heterogeneous systems an optimal choice for applications where minimizing energy consumption while meeting strict deadlines is critical.

The application of heterogeneous systems is particularly evident in the context of Network Function Virtualization (NFV), as discussed in [10]. These systems integrate different types of cores, like Performance cores (P-cores) and Efficient cores (E-cores), to optimize the deployment of virtual network functions (VNFs). By matching VNFs with the most suitable core type, heterogeneous systems enhance performance and energy efficiency, reducing overall power consumption [11]. The ability to dynamically allocate VNFs based on their computational needs also improves load balancing, ensuring even distribution of workloads and reducing latency. This flexibility and scalability make heterogeneous systems ideal for adapting to varying workloads and service demands, ensuring better Quality of Service (QoS) and positioning them as a sustainable solution for modern network environments.

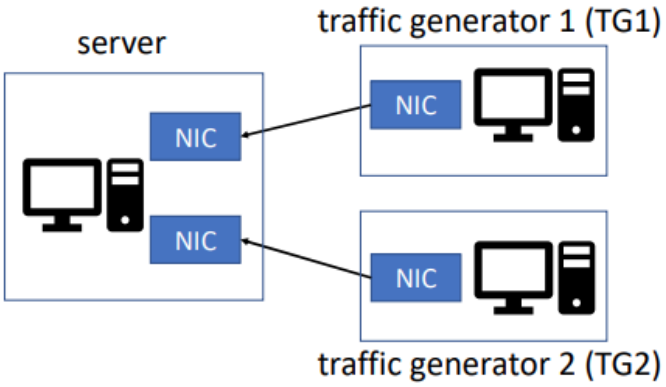

Figure 2.1: Hardware implementation [10]

Beyond NFV, heterogeneous systems play a crucial role in enhancing the computational power of edge computing systems, particularly within a CPU+FPGA heterogeneous architecture, as explored in [1]. In this architecture, FPGAs act as dynamic, reconfigurable accelerators that significantly improve system efficiency and perfor-

mance. The FPGA's reconfigurability allows it to adapt its functionality at runtime, making it suitable for various computational tasks that demand high performance and low latency. The dynamic partial reconfiguration (DPR) capability enables specific portions of the hardware to be reconfigured while the rest of the system continues to operate, thereby minimizing downtime and optimizing resource utilization. These accelerators, particularly the FPGA [12], are essential for handling tasks that require parallel processing, low power consumption, and high computational efficiency, which are vital for meeting the real-time performance demands of edge computing systems.

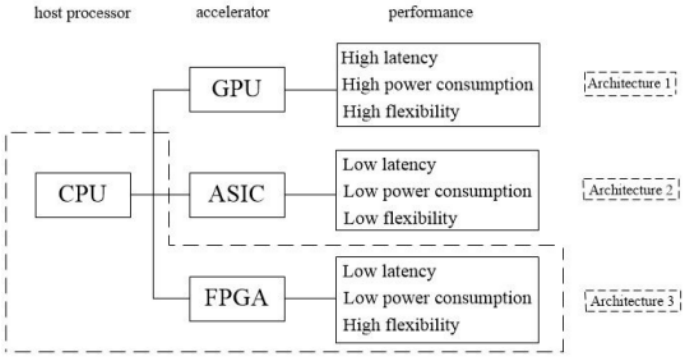

Figure 2.2: The characteristics of GPU, ASIC and FPGA[1]

Similarly, specialized hardware accelerators like Hotline, introduced in [13] , further illustrate the efficiency of heterogeneous systems in specific applications such as training recommendation systems. Hotline optimizes training by dynamically classifying embedding entries based on their access frequency and strategically utilizing both CPU and [14] GPU resources. By fragmenting mini-batches into smaller micro-batches and scheduling them efficiently, Hotline minimizes idle time and improves throughput. This example underscores the value of accelerators in heterogeneous systems, where targeted optimization can lead to substantial performance gains.

Finally, integrated CPU/GPU systems represent another significant advancement in computing efficiency, particularly for latency-sensitive applications like autonomous systems and edge intelligence, as discussed in [15]. These platforms combine the general-purpose processing power of CPUs with the parallel processing capabilities of GPUs, all within a single chip, enabling more efficient handling of complex tasks. The Unified Memory (UM) model simplifies memory management by allowing both

CPUs and GPUs to share the same memory space, reducing the overhead associated with data transfers. To further optimize performance, [16] proposes a framework that intelligently selects between CPU, GPU, and hybrid data initialization modes based on the application's specific needs. This approach reduces latency and ensures that data is processed as efficiently as possible, which is particularly beneficial for real-time applications.

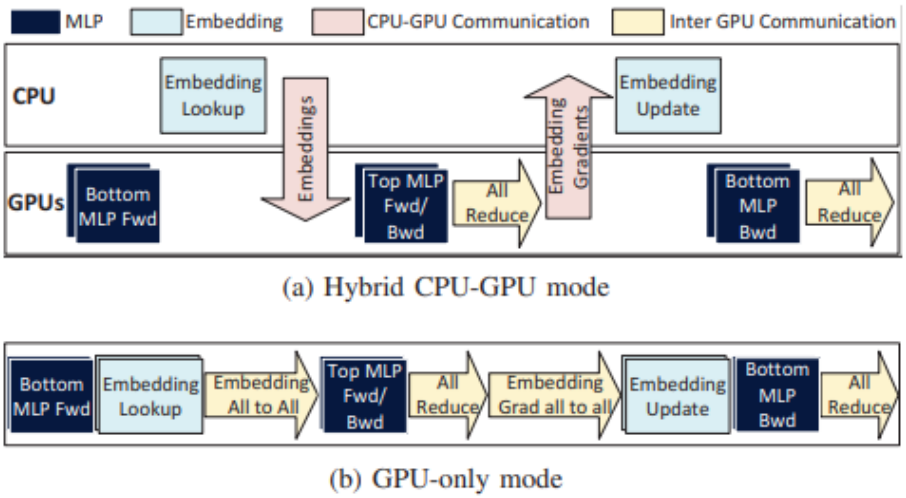

Figure 2.3: The execution flow of a typical recommendation model in the hybrid CPU-GPU and GPU-only. Due to their large sizes, the embedding tables are stored and processed on CPUs. The GPUs process the neural layers.[13]

In summary, heterogeneous systems, whether they integrate diverse cores, specialized accelerators, or CPU/GPU combinations, offer powerful solutions across various domains. Their ability to optimize resource utilization, improve energy efficiency, and enhance performance makes them indispensable in modern computing, from NFV and edge computing to autonomous systems.

2.2 Directed Acyclic Graph (DAG) and real-time systems

In various research studies, Directed Acyclic Graphs (DAGs) have been effectively used to represent real-time applications, demonstrating their utility in optimizing scheduling and resource allocation. For instance, [17] discusses the representation of several benchmark and real-world applications as DAGs, including: CyberShake,

Stencil, Gaussian Elimination, Epigenomics. Additionally, the paper features a case study on automotive control systems, where applications such as Adaptive Cruise Control (ACC), Traction Control (TC), and Electric Power Steering (EPS) are modeled as DAGs. In these models, tasks and their dependencies are represented as vertices and edges of the DAGs, respectively. The paper focuses on optimizing the scheduling of these DAGs on a distributed heterogeneous system to minimize energy consumption while ensuring that all tasks meet their real-time deadlines.

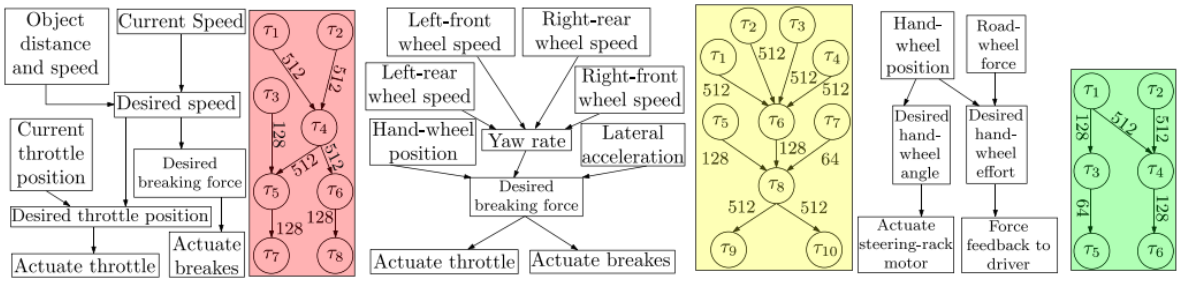

Figure 2.4: Representation of real-time applications as DAGs [17].

The [18] specifically considers a single periodic non-preemptive DAG running on a homogeneous multiprocessor platform. This setup is common in many real-time systems where tasks recur periodically and must be completed within specific time constraints. The non-preemptive nature means that once a task starts executing on a processor, it cannot be interrupted until it finishes. This is crucial in real-time systems to avoid overheads associated with task switching and to ensure predictability in task execution times.

In context of autonomous driving systems, [19] the real-time system represented as a Directed Acyclic Graph (DAG) is the Autoware framework, specifically a module related to the localization package used in autonomous driving systems. The paper models the Normal Distributions Transform (NDT) algorithm within Autoware as a DAG, showcasing different implementations of this algorithm on various processing engines, such as CPUs and GPUs, within a heterogeneous computing platform. The NDT algorithm is used to compute the precise position of an autonomous vehicle by matching Lidar data with offline map data. This DAG-based representation helps in scheduling tasks on a heterogeneous platform, ensuring real-time performance and efficient resource utilization.

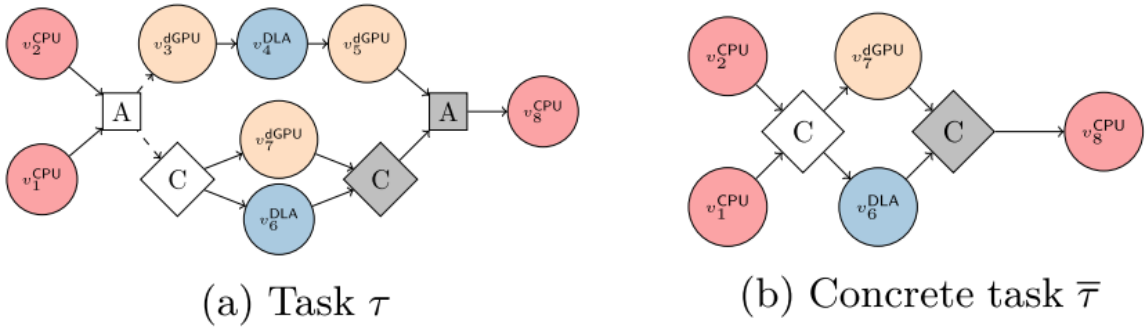

Figure 2.5: Task specification and concrete tasks [19].

The application of DAGs extends to OpenMP framework as well. In [20] the authors represent OpenMP tasks as Directed Acyclic Graphs (DAGs) to model the parallel execution structure. In this model, each task in OpenMP is represented as a vertex within the DAG. These tasks can be either implicit tasks, generated by constructs like omp parallel for or omp sections, or explicit tasks created by the omp task directive. Dependencies between tasks, such as those created by synchronization constructs like taskwait or depend clauses, are represented by edges in the DAG. These edges dictate the order in which tasks must be executed, ensuring that dependent tasks wait for their predecessors to complete. The DAG also incorporates execution constraints inherent in OpenMP. For example, the Thread Assignment (TA) constraint ensures that the number of threads assigned to a parallel region does not exceed what is specified in the OpenMP directive. Similarly, the Parallel-Region Execution (PRE) constraint ensures that once threads are assigned to a parallel region, only those threads can execute the tasks within that region. This structured representation allows for the analysis of critical properties like the longest path in terms of execution time (F-length) and total workload (F-volume), which are used to derive response time bounds for scheduling.

As per the above research, DAGs are instrumental in enhancing the understanding, optimization, and scheduling of real-time applications across various domains. This justifies my decision to implement DAGs in my project, leveraging their ability to represent complex task dependencies and ensure efficient, real-time performance.

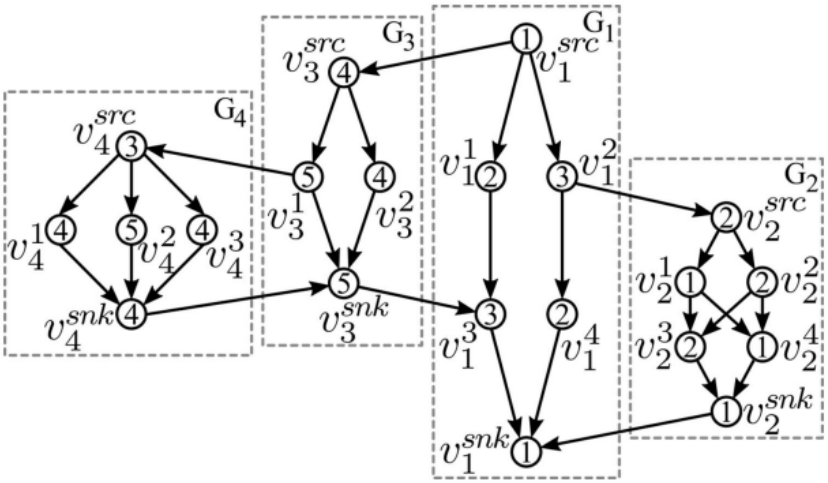

Figure 2.6: An example OpenMP DAG G (the label of each vertex is the worst-case execution time) [20].

2.3 Task Scheduling Algorithms in Heterogeneous CPU-GPU Systems

The realm of task scheduling in cloud and distributed computing environments has seen various innovative approaches aimed at optimizing energy efficiency, cost, and resource utilization, particularly in heterogeneous systems. The paper [21] presents an advanced scheduling method that prioritizes energy efficiency in a distributed green cloud (DGC) environment. It addresses the dynamic nature of power availability and costs across different geographical regions. By formulating the task scheduling as a constrained optimization problem, the study employs a novel Simulated-Annealing-Based Bees Algorithm (SBA) to intelligently allocate tasks among multiple green clouds. This method not only minimizes energy costs but also ensures that tasks meet their real-time deadlines by optimizing server allocation and operational speeds. However, the approach assumes homogeneity within data centers, which might not reflect the reality of mixed hardware environments, potentially leading to suboptimal scheduling decisions.

In a similar vein, the paper [22] introduces the Cost-Efficient Task Scheduling Strategy (CETSS), which focuses on managing workflow tasks in cloud environments with

strict deadline constraints. By modeling tasks as Directed Acyclic Graphs (DDAGs) and employing a critical path method, CETSS dynamically adjusts task deadlines and leverages a greedy algorithm to minimize execution costs. The strategy also reallocates tasks to maximize cost efficiency by optimizing VM usage. Despite its effectiveness, CETSS faces challenges in highly dynamic cloud environments where fluctuating resource availability and pricing can undermine its cost-saving strategies. Moreover, the complexity of handling heterogeneous resources and the reliance on accurate predictions for deadlines and power management introduce potential inefficiencies.

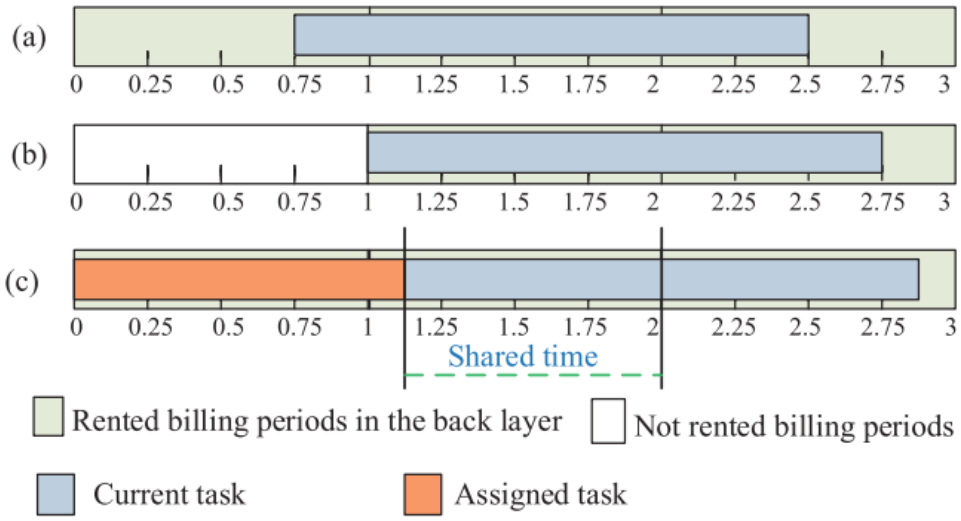

Figure 2.7: Example of task execution cost for a virtual machine [22]

The focus shifts to real-time parallel task scheduling on GPUs in paper [23], which proposes the RTGPU framework. This framework employs federated scheduling to allocate specific Streaming Multiprocessors (SMs) to tasks, ensuring predictable execution and preventing preemption. By using persistent threads and fine-grain GPU partitioning, RTGPU optimizes resource utilization and improves system throughput, particularly in scenarios where real-time deadlines are critical. However, the framework introduces complexities in managing the interdependencies between CPU, memory, and GPU segments, especially under high workloads. The non-preemptive nature of certain operations and the overhead of persistent thread management can further complicate the scheduling process, potentially leading to inefficiencies and underutilization of GPU resources.

In this [24], the authors explore the dynamic variability of CPU and GPU utilization in task scheduling, proposing a hybrid algorithm (H-PSO) that combines a heuristic greedy strategy with Particle Swarm Optimization. This approach aims to enhance both energy efficiency and resource utilization in heterogeneous systems. While the H-PSO algorithm shows significant improvements, it also comes with increased computational overhead and complexity, making it less suitable for real-time or latency-sensitive applications. Additionally, the reliance on specific models for energy consumption and CPU-GPU utilization may limit the algorithm's applicability across different hardware configurations and emerging computing technologies.

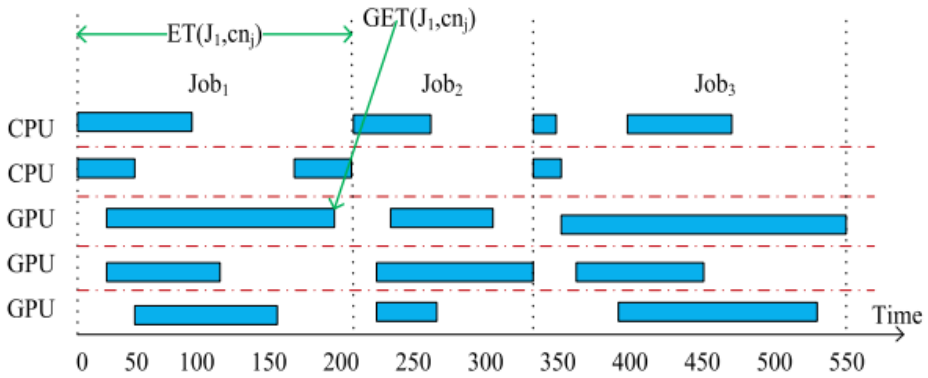

Figure 2.8: Computing node job execution [24].

In summary, real-time task scheduling in heterogeneous multicore systems, particularly those involving CPU-GPU architectures, is a complex challenge that requires balancing multiple factors such as energy efficiency, resource utilization, and meeting strict deadlines. While various algorithms have been proposed to address these challenges, each comes with its trade-offs, particularly in terms of computational complexity and the ability to adapt to diverse and dynamic computing environments. As these systems continue to evolve, future research will need to focus on developing more adaptable and efficient scheduling strategies that can operate effectively across a wider range of hardware configurations and application scenarios.

2.4 Power and Energy Optimization

In [8], the authors explore the use of task versions to optimize task deployment on heterogeneous multicore platforms whereas [6]. Task versions allow the system to adapt to

varying resource availability and constraints, such as energy consumption and real-time execution requirements. The approach involves modeling tasks using an imprecise computation (IC) task model, where each task is decomposed into a mandatory subtask that must meet real-time constraints and an optional subtask that can be executed to improve the quality of service (QoS) if resources allow. By utilizing dynamic voltage and frequency scaling (DVFS) [25], the system adjusts the power factor by selecting appropriate voltage/frequency levels, enabling energy-efficient task execution while meeting QoS demands. The deployment method simultaneously optimizes task allocation, scheduling, frequency assignment, and task migration, allowing for more flexible and efficient use of heterogeneous cores.

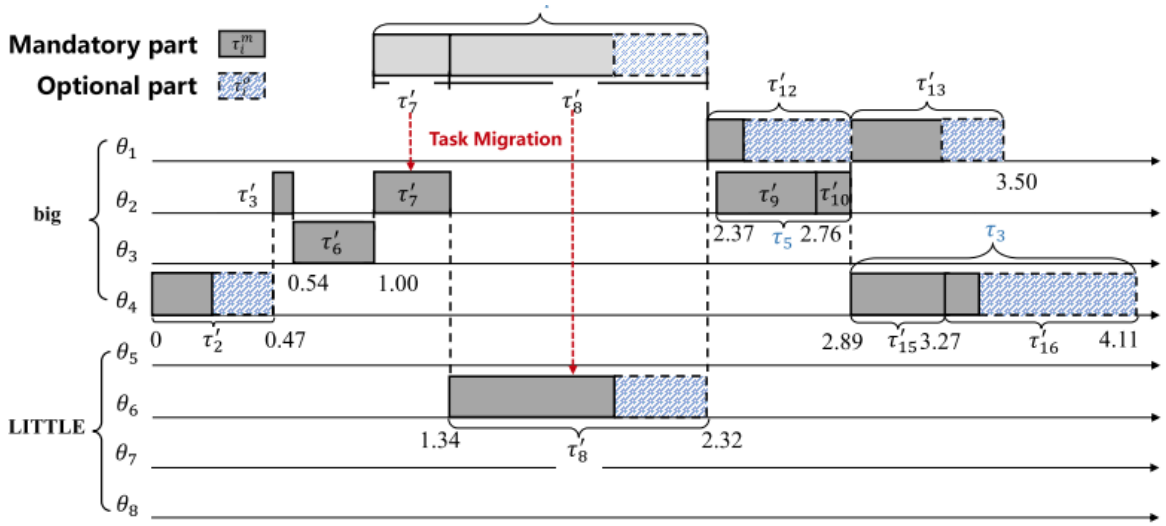

Figure 2.9: Task deployment results [8].

2.5 Jetson Nano

The Jetson Nano is a compact AI computing platform featuring a 128-core NVIDIA Maxwell GPU with 472 GFLOPS of AI performance and a quad-core ARM Cortex-A57 CPU with a maximum frequency of 1.43GHz. It includes 4GB of 64-bit LPDDR4 memory with 25.6GB/s bandwidth and 16GB of eMMC 5.1 storage. [7] The Nano supports video encoding and decoding up to 4K resolution, connects up to four cameras via 12 MIPI CSI-2 lanes, and offers a variety of I/O options including PCIe Gen2, USB 3.0, and Gigabit Ethernet. It operates within a power range of 5W to 10W and is housed in a

69.6mm x 45mm form factor with a 260-pin SO-DIMM connector.

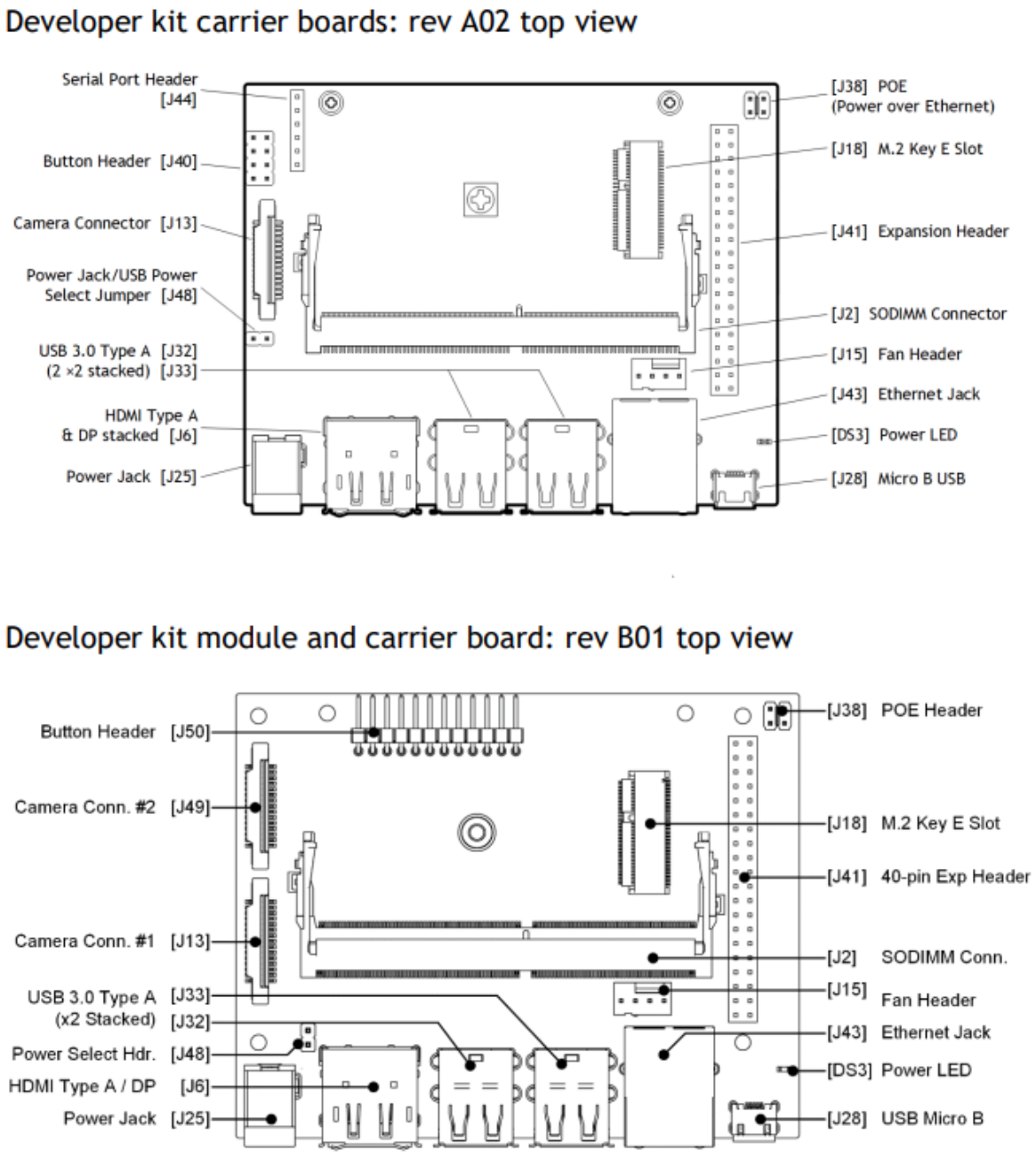

Figure 2.10: Jetson nano [26].

The NVIDIA Jetson Nano offers several advantages, particularly for developers looking to bring AI capabilities to edge devices. [26] It combines high performance with low power consumption, making it ideal for advanced robotics, IoT, and other autonomous applications. The Jetson Nano features a 128-core NVIDIA Maxwell GPU capable of 472 GFLOPS, providing substantial computational power for AI tasks. It

supports multiple camera inputs, making it suitable for vision-based applications. Additionally, it is part of the broader NVIDIA Jetson ecosystem, ensuring compatibility with a wide range of software tools like TensorRT, CUDA, and OpenCV [27]. The platform's compact size and cost-effective pricing further enhance its appeal, especially for projects requiring scalable, deployable AI solutions at the edge.

The Jetson Nano, with its quad-core ARM Cortex-A57 CPU and 128-core NVIDIA Maxwell GPU, offers a compact yet powerful platform for developers. Its ability to handle AI workloads with 472 GFLOPS of performance makes it ideal for testing and observing task scheduling algorithms in real-time. The built-in GPU accelerates complex computations, while the multi-core CPU ensures efficient processing of multiple tasks. This combination, along with its low power consumption and support for various interfaces, makes the Jetson Nano a versatile and accessible tool for developers to prototype and optimize their algorithms effectively.

CHAPTER 3

Methodology

The method used to accomplish the goals of this project involves a structured approach, beginning with the division of research into distinct groups to focus on specific tasks. The first crucial step was establishing the design specifications, which are essential for defining the objectives, methods, and inputs needed before starting any task. Following this, a basic architecture of the design flow was developed, represented as a block diagram with smaller components, allowing for a clear, step-by-step implementation process. The next step was designing the algorithm, which included writing the necessary code to ensure the research objectives were met. Finally, the project concluded with the simulation and implementation of the algorithm to achieve the desired outcomes.

3.1 Design specification

The first step is to thoroughly understand where and how to implement the design. The system used for writing, simulating, and checking results was `cseelab758.essex.ac.uk`, which has a 12th Gen Intel Core i7-12700 processor running at 2.10GHz, 16GB of RAM, and operates on a 64-bit Windows 11 Education (version 23H2) Operating System.

The algorithm is developed in Python, a language renowned for its simplicity, readability, and versatility, making it an ideal choice for both beginners and experienced developers. Python's extensive libraries such as NumPy, Pandas, and NetworkX, offer powerful tools for handling complex tasks like data analysis, machine learning,

and task scheduling [28]. The Jupyter Notebook is used as the the programming environment due to its interactive nature, which allows for the integration of live code, equations, visualizations, and narrative text into a single, coherent document [29]. This combination of Python and Jupyter Notebook provides a flexible and efficient environment for developing data-driven projects.

To design the algorithm, several key Python libraries were used. NumPy handled numerical computations, and Pandas managed and analyzed data in structured formats. Defaultdict from collections simplified dictionary operations with default values, while combinations from itertools generated task combinations. NetworkX was crucial for creating and analyzing task dependency graphs (DAGs). Matplotlib.pyplot and matplotlib.patches provided tools for visualizing data and adding graphical elements. Dequ managed efficient data structures for queue operations, Pprint made data structures more readable, and json and pickle enabled data serialization and storage. Each library played a vital role in building a flexible and efficient algorithm [30].

To perform the implementations, the NVIDIA Jetson Nnao was used as the core processing unit due to its capability to handle high-performance, real-time AI and machine learning task, such as object detection and image processing, in a compact and cost-effective manner [31]. Despite its small size and limited processing threads, the Jetson Nano efficiently runs critical algorithms like the SSD (Single Shot Multibox Detector) for managing flow and detecting violations. Its power-efficient design makes it ideal for implementing a smart traffic control system in resource-constrained environments.

3.2 Architecture

At the heart of the system are the initial inputs, which includes Directed Acyclic Graph (DAG) representing the real-time applications that maps out the dependencies between tasks, as well as critical parameters like the deadline, power factor, and the number of processing cores available. These inputs are crucial in shaping the scheduling strategy, particularly in environments like the NVIDIA Jetson Nano, where tasks must be allocated efficiently between CPU and GPU cores. The DAG ensures that tasks are executed in an order that respects their dependencies, which are essential for maintain the

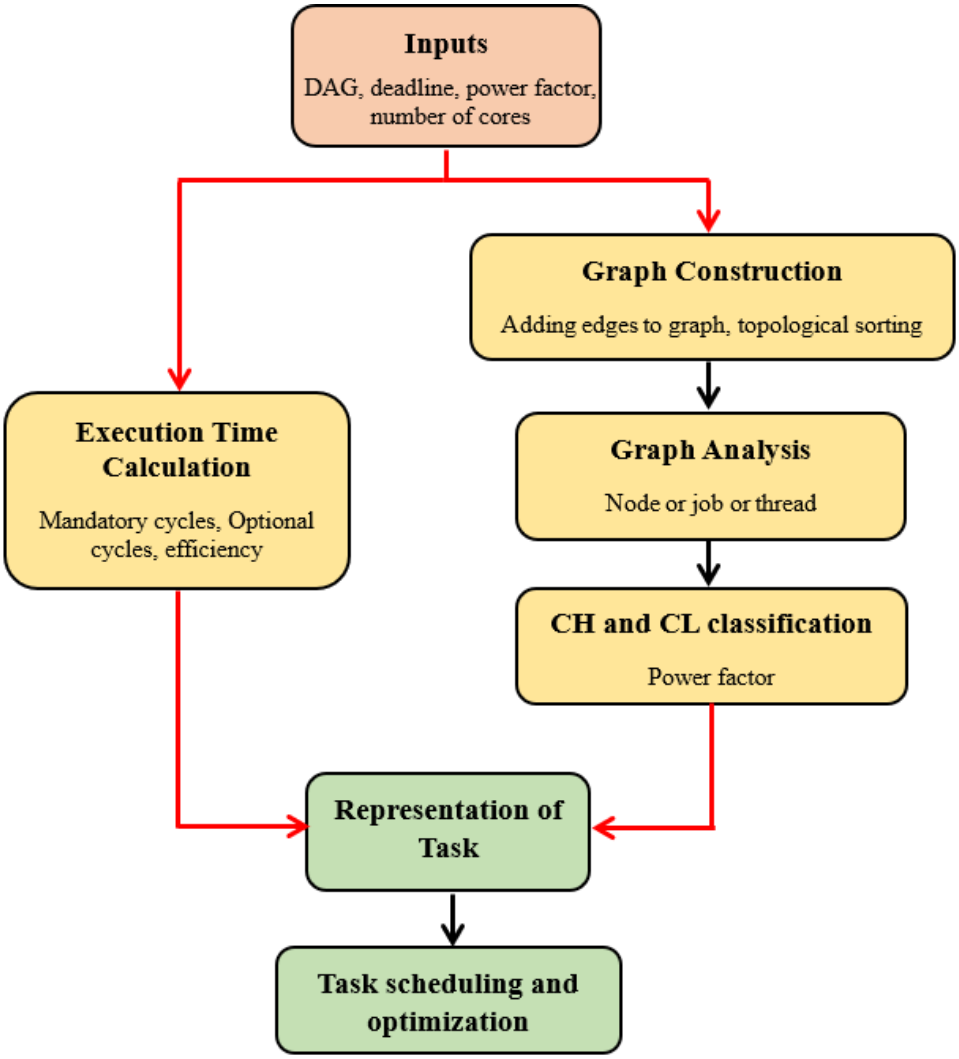

Figure 3.1: Architecture of the design

real-time requirements of the system. Also, considering power and deadlines ensures that the system not only meets the timing constraints but also operates within its power budget. Using these as inputs execution time and graphs are analysed.

In the Graph Construction phase, this involves adding edges to the DAG to represent task dependencies and performing topological sorting to determine the correct execution order of tasks. Topological sorting is a critical step as it organizes tasks in a sequence that ensures all dependencies are respected also, no task can be executed before all its prerequisites are completed. This phase is particularly important in heterogeneous systems, where tasks must be carefully scheduled across different types of processing units (CPU and GPU). The output of this phase directly influences the subsequent steps, as it determines the execution order that will be used for further analysis and

optimization. The ability to correctly construct and sort the graph is fundamental to the success of the overall scheduling process.

Once the graph is constructed and sorted, the system analyses the DAG to identify task dependencies and opportunities for parallel execution as required. Task are classified into critical high (CH) and critical low (CL) tasks based on their dependencies of tasks. Once they are classified power factors are taken into consideration by analysing versions of each task. This strategic allocation helps in balancing the load across the system, ensuring that tasks are executed efficiently while adhering to the power and timing constraints.

The next block is Execution Time Calculation, where the system calculates the execution times for each task version. This includes considering both mandatory cycles, which are essential for producing a minimally acceptable result, and optional cycles. This stage is crucial for determining whether tasks can be completed within the given deadlines and how much power will be consumed in the process. By calculating the execution times, the system can make informed decisions about which task versions to execute, balancing the need for accuracy with the constraints of time and power. This step directly feeds into the representation of tasks in a matrix, where tasks are organized by their start and finish times, aligned with the available cores.

In the Task Scheduling and Optimization phase, where the system optimizes the schedule of tasks across the available processing cores. The goal is to ensure that all tasks are completed within the specified deadline while minimizing power consumption. The representation of tasks in a matrix form provides a clear visualization of the scheduling process, helping the system to identify the best allocation of tasks to the available cores. The architecture also incorporates feedback loops, which allow the system to iterative refine the schedule by revisiting earlier steps if necessary. For example, if the execution time calculation reveals that a particular task cannot be completed within the deadline, the system might adjust the topological order or reclassify tasks to ensure that the schedule remains feasible.

3.3 Designing task scheduling algorithm

The algorithm design requires inputs such as essential libraries (3.1), node/job/thread data, deadlines, maximum power factor, fixed cores, and the DAG graph structure. Additionally, it considers which nodes/jobs/threads can be executed in parallel. These inputs collectively enable efficient task scheduling and execution. The schedule is generated by prioritizing tasks based on their As Late As Possible (ALAP) [6] times to ensure that precedence constraints are respected. If the initial schedule exceeds the deadline, the algorithm iterative downgrades the versions of selected tasks—those with the least impact on overall accuracy—until a feasible schedule is obtained.

3.3.1 Defining inputs

In a Directed Acyclic Graph (DAG) representing a real-time application, a single DAG encapsulates the structure of a task. The basic components of this structure are nodes and edges. Nodes, which can represent tasks, operations, threads, jobs, or various applications, are the key entities within the DAG. These nodes are interconnected by edges, also known as arcs, which define the relationships and dependencies between them. The direction of the edges is crucial, as it indicates the flow of execution or the precedence constraints between nodes. In essence, edges signify that one node (task or operation) must be completed before another can commence, establishing a clear sequence of execution. The acyclic nature of the DAG ensures that there are no cycles, meaning there is no way to return to a node once it has been visited, which is vital for accurately modelling task dependencies in real-time systems. DAG can be represented in different structures as shown in 3.2, in this thesis fork-join structure is used.

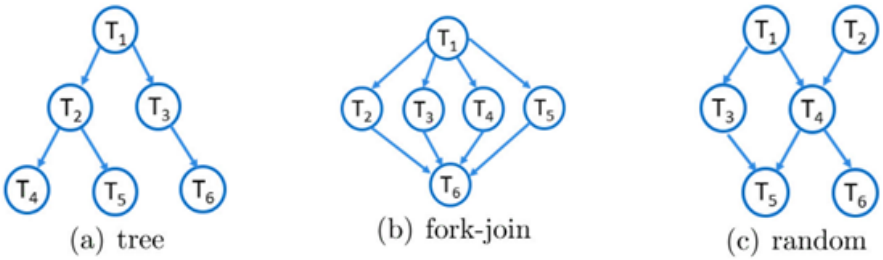

Figure 3.2: DAG Structures examples [5]

Given a set of tasks, each task is represented in a node (Node1, Node2, etc.) as T_i, each associated with multiple versions T_i^j of tasks. Each version contains two configurations: C_L (low configuration) and C_H (high configuration). These configurations specify parameters like M_i (mandatory cycles), O_i (optional cycles), Pow_i^j (power or energy consumption), and Y_i, L or Y_i, H (performance or efficiency metrics under low and high configurations, respectively). For example, Task1 has one version where the low configuration (C_L) has $M_i = 4$, $O_i = 2$, $\text{Pow}_i^j = 10$, and $Y_i, L = 0.5$, while the high configuration (C_H) increases Pow_i^j to 13 and Y_i, H to 0.7. The other tasks have similar versions with different values for these parameters, providing a comprehensive set of possible task configurations across different nodes. Other inputs include a deadline of 142 units, requiring the task to be completed within this time-frame, and a maximum power factor of 50 units, meaning no job should exceed 50 units of power. The number of cores is fixed at two, ensuring that complex tasks are efficiently scheduled.

Table 3.1: Defining Inputs

Tasks	**Versions**	**CL**				**CH**			
		Mi	**Oi**	**Pow_i**	**Yi,L**	**Mi**	**Oi**	**Pow_i**	**Yi,H**
Task1	1	4	2	10	0.5	4	2	13	0.7
Task2	1	10	5	20	0.4	10	5	26	0.6
	2	10	8	22	0.4	10	8	29	0.6
	3	10	8	24	0.4	10	10	30	0.6
Task3	1	10	2	12	0.5	10	2	12	0.6
	2	10	4	12	0.5	10	4	14	0.6
	3	10	6	15	0.4	10	6	17	0.6
Task4	1	28	16	15	0.8	28	16	20	0.8
	2	28	20	18	0.8	28	20	25	0.7
Task5	1	8	4	24	0.5	8	4	28	0.7
	2	8	4	24	0.5	8	4	28	0.7
	3	8	5	27	0.4	8	5	31	0.7
Task6	1	10	3	12	0.7	10	3	12	0.7
	3	10	9	14	0.7	10	9	16	0.7

3.3.2 Defining graph and sorting topologically

Once the above parameters are given as input, next is to define a class for constructing and managing DAG with functionalities to add edges, perform a non-recursive topological sort, and identify the successors of specific tasks. For this research task from [6] is taken as shown in 3.3. The graph is populated with tasks (represented as nodes) and their dependencies (represented as directed edges). The topological sort of method is implemented non-recursively, which is beneficial in avoiding the potential hidden recursion, such as stack overflow in cases where the graph is large or has a deep hierarchy. The non-recursive approach uses an explicit stack to manage the nodes, making the algorithm more robust and capable of handling larger graphs without risking the limitations of Python's recursion depth. Additionally, the code identifies parallel tasks and finds their successor tasks within the graph, providing insights into how tasks can be scheduled concurrently. The graph is then visualized using the 'networkx' and 'matplotlib' libraries, with the layout manually adjusted for the clarity.

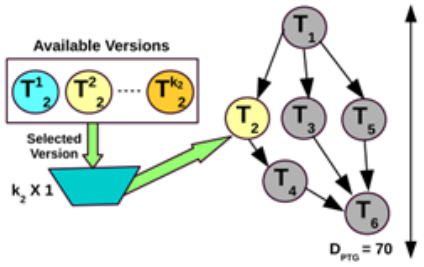

Figure 3.3: Task graph [6]

3.3.3 Execution time calculation

In the context of scheduling and resource management, particularly when working with tasks represented in clock cycles, it's important to convert these cycles into timing in seconds to achieve accurate planning and execution. The inputs and the structure of the task graph are outlined in Sections 3.3.1 and 3.3.2. After defining the task graph, the next crucial step is to calculate the execution times for each task and their respective versions, as these tasks include both mandatory and optional components, represented

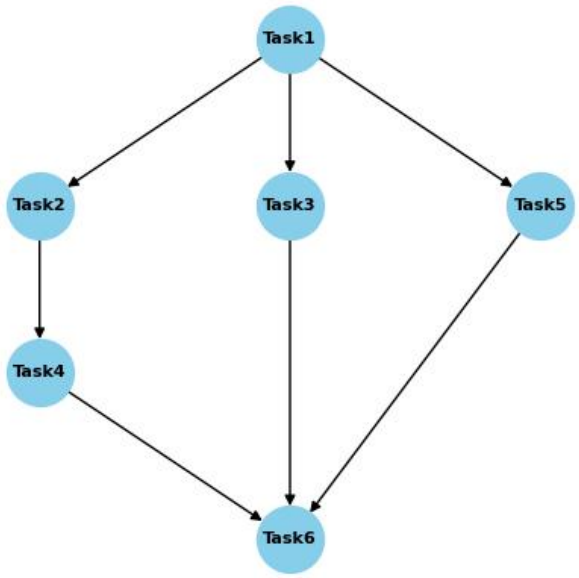

Figure 3.4: Task graph [6]

by clock cycles. The length of the j^{th} version of task T_i (len_i^j) is expressed as:

$$\text{len}_i^j = M_i + O_i^j[6] \tag{3.1}$$

This formula calculates the total number of clock cycles required for a specific version of a task, incorporating both its critical (mandatory) and additional (optional) components. Once the length in clock cycles is determined, the execution time in seconds can be calculated for different task configurations, such as critical low (C_L) and critical high (C_H). The execution time is crucial for understanding how long a task will take to complete under different configurations, which in turn affects how tasks are scheduled within the available resources, ensuring that deadlines are met without exceeding the system's power constraints. For example, the execution time for a critical low task T_i under a specific version is calculated using the following formula: [6]

$$\text{Execution time for task } T_i^j, C_L = \frac{\text{len}_i^j}{Y_i^j, L \times 0.5} \tag{3.2}$$

$$\text{Execution time for task } T_i^j, C_H = \frac{\text{len}_i^j}{Y_i^j, H \times 1} \tag{3.3}$$

These calculations are vital for determining the exact time required to execute each

task version, allowing for precise scheduling that considers both the time constraints (deadlines) and the system's power limitations. This approach helps in optimizing the overall performance and ensuring that the system runs efficiently without overloading any resources.

3.3.4 Classification of tasks as critical high and critical low

In task scheduling and optimization, tasks are categorized into two primary groups: Critical High (C_H) and Critical Low (C_L). This classification hinges on the number of successors a task has within a dependency graph. Tasks with multiple successors are deemed more critical because they significantly influence the execution of subsequent tasks, leading to their classification as C_H. Conversely, tasks with fewer or no successors are classified as C_L, as they have a lower impact on the overall sequence of task execution.

After classification, the next critical step is to select the most appropriate versions of these tasks while carefully considering power consumption constraints. Each task version comes with associated power consumption and execution time. Initially, values with the maximum optional parts are considered as the ideal case. But the goal is to choose versions that maximize system performance without exceeding a predefined power consumption limit. When parallel tasks are identified, such as Task5 and Task3 being set as C_H tasks and Task2 as a C_L task, a power factor formula is applied to ensure efficient resource use:

$$\text{Pow}(T_2^j), C_H + \text{Pow}(T_5^j), C_H \leq 50 \tag{3.4}$$

$$\text{Pow}(T_2^j), C_H + \text{Pow}(T_3^j), C_H \leq 50 \tag{3.5}$$

If the total power consumption exceeds the limit of 50 units, the algorithm adjusts by examining the versions of the C_L tasks. It downgrades [6] to the next lower version ($j - 1$) and recalculates the power factor. This iterative process continues until the combined power factors of the parallel tasks are within the acceptable range. The

outcome is an optimized task schedule where critical tasks are executed with maximum efficiency, and less critical tasks are adjusted to ensure that the system operates within its power and resource constraints. This balanced approach is vital in systems where both performance and power consumption are tightly controlled, ensuring tasks are completed within required timeframes without overloading the system's resources.

3.3.5 Representing tasks as matrix

In the initial stage of the project, parallel tasks assigned to Critical Low (C_L) and Critical High (C_H) categories, after considering power constraints, are stored in a matrix format. The matrix is structured with the number of columns equal to the number of levels in the Directed Acyclic Graph (DAG), and the number of rows corresponding to the number of available cores—in this case, two cores.

To populate this matrix, the topological sort of the tasks is considered. The tasks are processed sequentially from the topological list, where each element represents a task number. If a task is scheduled under the C_L category, the 'count-cl' variable tracks its position, and the task is placed in the corresponding row in the matrix. Similarly, if a task is scheduled under C_H, it is tracked by the 'count-ch' variable and placed in the respective row. If no task is scheduled for a particular level, the matrix column for that level is left as 'None'. For example, consider a scenario where tasks 2, 3, and 5 are running in parallel, and the DAG has 4 levels. Based on the topological sorting and the classification from the earlier section, the matrix looks as follows:

$$\begin{bmatrix} \text{None} & \text{Task } T_5^j & \text{Task } T_3^j & \text{None} \\ \text{None} & \text{Task } T_2^j & \text{None} & \text{None} \end{bmatrix}$$

In this matrix, Task5 and Task3 are allocated under the C_L row, while Task2 is allocated under the C_H row. The other values remain as 'None' since no tasks are assigned to those levels. This matrix provides a clear and structured representation of how tasks are distributed across cores and levels, reflecting the scheduling decisions based on criticality and power considerations.

To allocate the remaining tasks after the initial assignment, the process involves checking columns in the matrix that currently have 'None' values, which indicate unassigned tasks, and then focusing on their corresponding levels. For instance, if levels 1 and 4 have unassigned tasks, the tasks associated with these levels need to be identified and assigned to the appropriate columns (level 1 and level 4).

When determining which row (representing a core) to assign these tasks to, the decision is based on the execution times of the tasks, considering both the mandatory and optional parts, regardless of whether the task is classified as Critical Low (C_L) or Critical High (C_H). The approach involves calculating the total execution time for the tasks already assigned in each row by summing the execution times in that row. Next, compare the summed execution times across the rows to identify the row with the maximum total execution time. This total is then subtracted from the overall deadline to determine the remaining available time for scheduling the unassigned tasks. For example, consider the following matrix:

$$\begin{bmatrix} \text{None} & \text{Task } T_5^j = 50 & \text{Task } T_3^j = 20 & \text{None} \\ \text{None} & \text{Task } T_2^j = 30 & \text{None} & \text{None} \end{bmatrix}$$

Task $T_5^j(50) + T_3^j(20) = 70 > T_2^j(30)$

Therefore, Deadline $-70 =$ remaining deadline

Since 70 units is greater than 30 units, the row with the maximum execution time (row 1) is used to calculate the remaining deadline by subtracting 70 from the initial deadline. The remaining tasks in levels 1 and 4 are then considered for assignment to the row that allows their combined execution times to fit within the remaining available time, ensuring that the system remains within its operational constraints. This approach ensures an efficient and balanced distribution of tasks across the available cores while respecting the overall deadline.

Now, the algorithm checks for the remaining tasks and their respective level. A task is placed in the column which is equal to the level, and the row is decided based on the free space available ('None'). If the task is placed below another task, for example, if

Task_i^j is placed below Task_3^j, the power factor is calculated as explained above.

If a task is placed below another task in the same column (for instance, if Task_i^j is placed below Task_3^j), the algorithm recalculates the power factor, as previously described. This ensures that the combined power consumption of the tasks in the same column does not exceed the predefined power limit. The algorithm continues this process of placing tasks and recalculating power factors until all tasks are assigned appropriately, ensuring that the system remains within its operational constraints and the tasks are efficiently distributed across the available cores.

This is how the scheduled tasks are organized and displayed in a matrix format for further analysis and representation. The matrix structure provides a clear and systematic way to visualize how tasks are allocated across different levels and cores, ensuring that all scheduling constraints, such as power factors and execution times, are met. Outputs are shown in the results section, where they provide a detailed overview of the scheduling outcomes, including how effectively the system meets its deadlines and resource limitations. This representation not only facilitates easy tracking of task distribution but also helps in identifying potential bottlenecks or inefficiencies, ensuring that tasks are efficiently managed and system resources are fully optimized.

3.4 Pseudo code

This section presents the pseudocode for task scheduling in heterogeneous multicore real-time systems modeled as Directed Acyclic Graphs (DAGs). In such systems, tasks must be efficiently managed to ensure they meet strict deadlines while operating within constraints like power consumption and core availability. The pseudocode outlines the key steps required to organize and execute tasks across multiple cores, considering the specific characteristics and requirements of each task version. This approach aims to optimize system performance and guarantee that all tasks are completed within the designated timeframe.

Algorithm 1 Task Scheduling Algorithm

Inputs:
SET DAG, Deadline $D = 142$, $\text{Pow}_{\text{max}} = 50$, Cores $C = 2$
DEFINE tasks with M_i, O_i^j, Pow_i^j, $Y_{i,L}$, $Y_{i,H}$
Topological Sort:
INITIALIZE T_{sorted}
  FOR each task T_i:
    IF no predecessors:
      ADD T_i to T_{sorted}
      FOR each successor T_s:
        DECREASE in-degree of T_s
        IF in-degree is 0, ADD to T_{sorted}
  OUTPUT T_{sorted}
Classify Tasks:
  FOR each task T_i in T_{sorted}:
    IF successors > 1, CLASSIFY as C_H
    ELSE, CLASSIFY as C_L
Calculate Execution Times:
  FOR each T_i^j:
    SET Exec_Time $C_L = \frac{M_i + O_i^j}{Y_{i,L} \times 0.5}$
    SET Exec_Time $C_H = \frac{M_i + O_i^j}{Y_{i,H}}$
Assign to Matrix:
  INITIALIZE $M[2][L]$
  FOR each task T_i in T_{sorted}:
    IF C_L, ASSIGN to $M[0][\text{count}_{C_L}]$
    IF C_H, ASSIGN to $M[1][\text{count}_{C_H}]$
Check Power Constraints:
  FOR each pair (T_{i1}^j, T_{i2}^j) in M:
    IF Pow_combined $> \text{Pow}_{\text{max}}$, DOWNGRADE C_L version
Schedule Tasks:
  FOR each T_i in M:
    SET $t_s = 0$ if first, ELSE $t_s =$ predecessor finish time
    SET $t_f = t_s +$ Exec_Time
Output:
  VALIDATE all tasks meet D and Pow_{max}
  PRINT final schedule

CHAPTER 4

Results and Observations

4.1 Task Scheduling Ideal case vs. Actual case

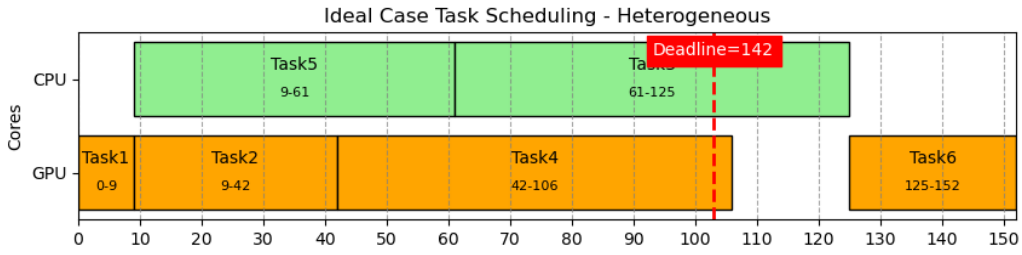

Figure 4.1: Ideal case task scheduling

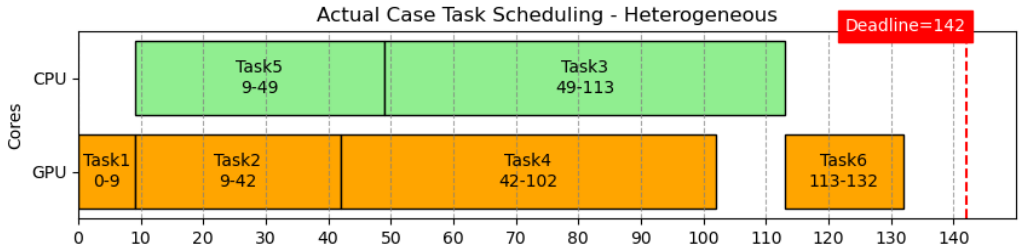

Figure 4.2: Actual case task scheduling

In an ideal scenario as shown in 4.1 where the maximum optional cycles are considered for each task, it is observed that the deadline is not met even with 2 cores, and scheduling tasks remains challenging despite topological sorting. However, by employing an algorithm that schedules tasks based on dependencies and power optimization,

it is possible to successfully schedule tasks within the given deadline using just 2 cores which is shown in 4.2. This demonstrates that, with efficient processor utilization and power optimization, tasks can be effectively scheduled even with a minimal number of processors.

4.1.1 Task Scheduling in Homogeneous system

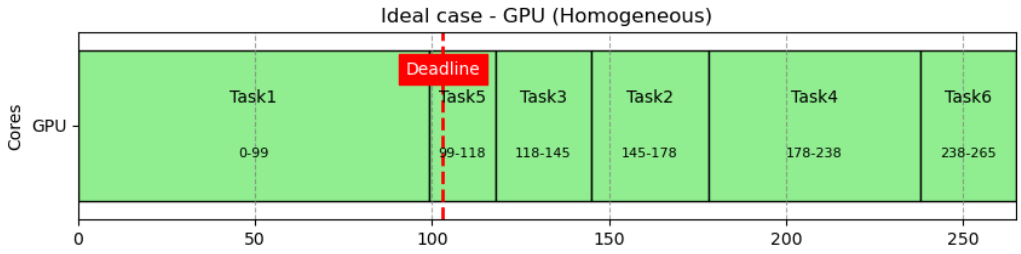

Figure 4.3: Ideal case GPU

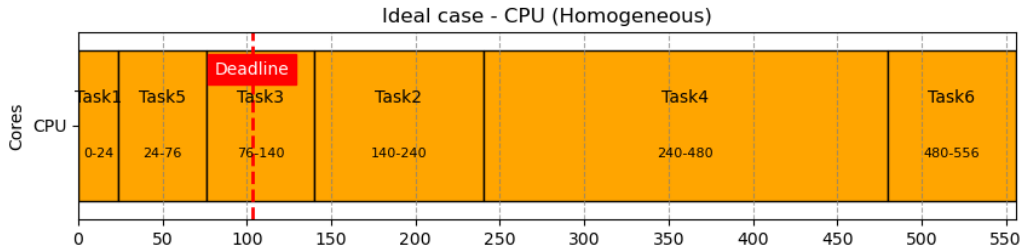

Figure 4.4: Ideal case GPU

The figures 4.4 and 4.3 clearly demonstrate that scheduling tasks within the minimum deadline is virtually impossible. While it is feasible to schedule these tasks in homogeneous systems, this requires extending the deadline. However, increasing the deadline entails prolonging both the mandatory and optional cycle times, which ultimately leads to a reduction in overall efficiency.

4.2 System Utilization in Homogeneous vs. Heterogeneous Systems

The first two charts represent ideal scenarios where either the CPU or GPU as shown in 4.5 is fully utilized without any idle capacity, likely pushing the system to its limits.

In contrast, the third chart illustrates a more practical heterogeneous scenario where both CPU and GPU are used together, leaving some idle capacity, which suggests a more balanced and potentially more efficient system. This comparison highlights the difference between theoretical maximum utilization and actual resource usage in a combined CPU-GPU setup.

In the actual case, the utilization of both the GPU and CPU is calculated separately by summing the execution times (ET) of all tasks for each processor and dividing by the system's deadline. The total system utilization is then determined by averaging the utilization of the GPU and CPU, and multiplying by 100 to express it as a percentage. This calculation provides a measure of how effectively the entire system's resources are being utilized within the given deadline.

$$\text{Utilization of GPU} = \frac{\sum \text{ET}_i^j}{\text{Deadline}}$$

$$\text{Utilization of CPU} = \frac{\sum \text{ET}_i^j}{\text{Deadline}}$$

$$\text{Total utilization} = \frac{(\text{Utilization of GPU} + \text{Utilization of CPU})}{2} \times 100$$

In ideal homogeneous systems, where tasks are scheduled exclusively on either the CPU or GPU, the utilization can exceed 100%. This over utilization indicates that the system is being pushed beyond its optimal capacity, which is not a desirable or efficient solution. It highlights the limitations of relying solely on one type of processor, as it can lead to performance degradation and potential system instability.

4.3 Power Optimization and Consumption

In the ideal case, where the maximum optional parts of each version of the tasks are considered, we observe a significant issue when multiple tasks are running in parallel:

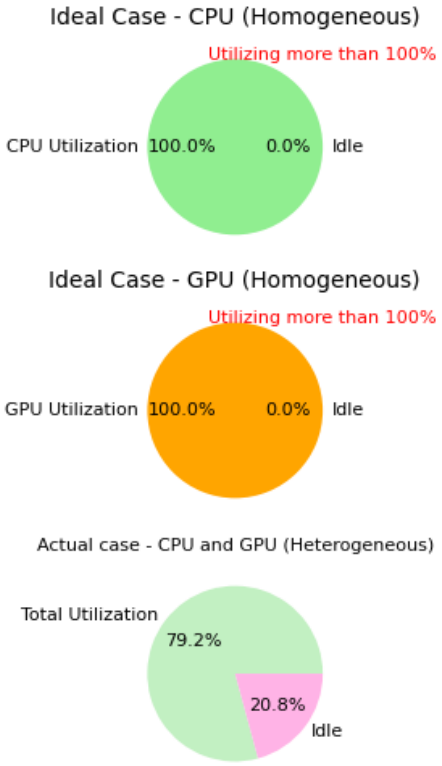

Figure 4.5: Utilization

the power consumption exceeds the defined maximum power limit. This situation, as illustrated in the graph, indicates that the combined power demands of certain task groupings, such as "Task2 + Task5," surpass the threshold of 50 units shown in 4.8. When power consumption exceeds this limit, it suggests that the system is operating inefficiently, consuming excessive energy, which could lead to overheating, reduced system lifespan, or failure to meet energy compliance standards. This is clearly not an optimal scenario for power management.

In contrast, the actual case employs an optimized approach where the algorithm carefully selects task versions, considering both their mandatory and optional parts, to ensure that power consumption remains within or at the maximum allowable limit 4.9. By strategically managing the task executions and choosing versions with lower power demands, the algorithm effectively reduces the overall power consumption. This

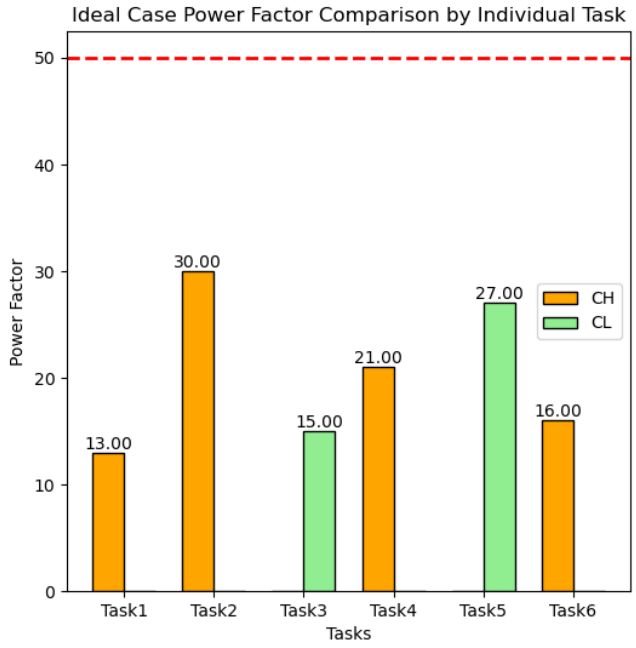

Figure 4.6: Ideal case power comparison by individual task

optimization is critical because it allows the system to run efficiently without exceeding power limits, which is crucial for maintaining system stability, preventing overheating, and ensuring the system's longevity.

The success of this optimization highlights the effectiveness of the algorithm used in this project. It demonstrates that, by intelligently scheduling tasks and selecting appropriate versions, it is possible to maintain a balance between performance and power consumption. The algorithm not only ensures that all tasks are completed within the given constraints but also optimizes power usage, making it a robust solution for managing complex, power-sensitive systems.

This optimized approach is particularly beneficial in environments where power resources are limited or where energy efficiency is a priority. By ensuring that the power consumption does not exceed the maximum limit, the algorithm contributes to a more sustainable and cost-effective operation. It reduces the risk of system failures due to power overloads and contributes to the overall reliability and efficiency of the system. This careful balance between task scheduling and power management is a key achievement of the project, demonstrating how advanced algorithms can be used to

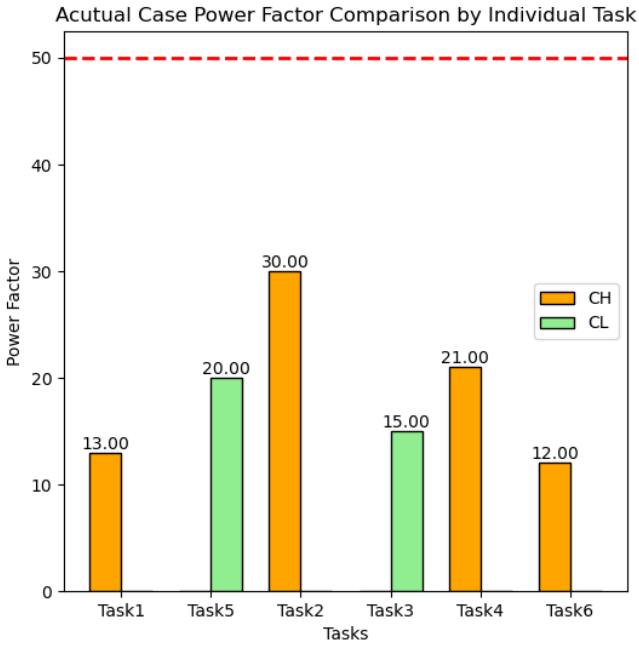

Figure 4.7: Actual case power comparison by individual task

address real-world challenges in system design and operation.

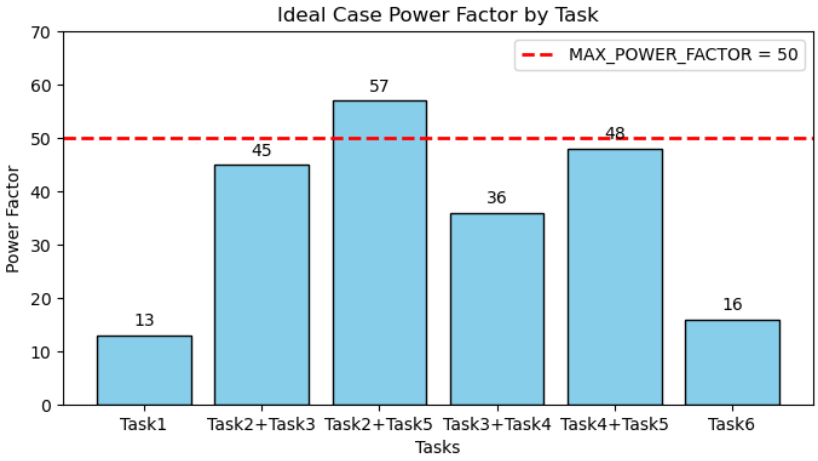

Figure 4.8: Ideal case power by task

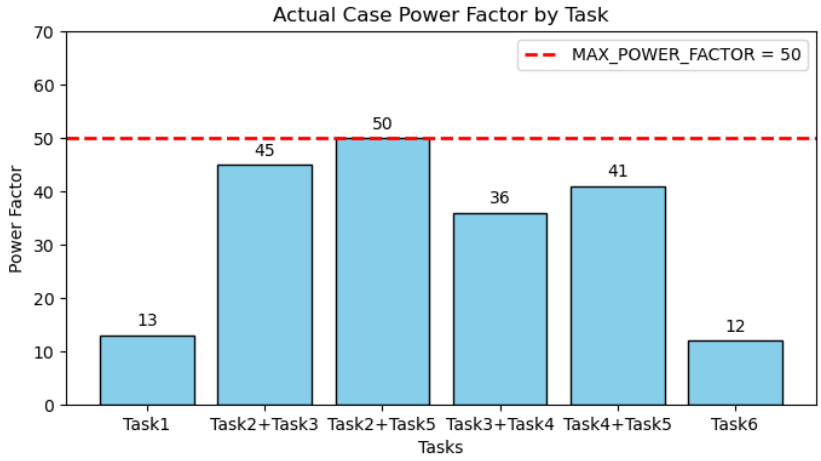

Figure 4.9: Actual case power by task

4.4 Accuracy Analysis

Accuracy in this project is quantified by the metric NQ, where NQ is defined as the ratio of the sum of Actual Quality of Service (QoS) to the sum of Ideal Quality of Service (QoS). The equation:

$$NQ = \frac{\sum \text{Actual QoS}}{\sum \text{Ideal QoS}} = \frac{\sum \text{Actual Oi}}{\sum \text{Ideal Oi}}$$

[6]

serves as a critical measure for evaluating the Quality of Service (QoS) in the context of task scheduling and execution within a system. QoS is essentially a measure of the system's performance, specifically focusing on how well tasks adhere to their expected performance targets, such as meeting deadlines and achieving optimal execution efficiency. The "Actual QoS" refers to the performance level that is actually achieved by the system during execution, taking into account the sum of optional cycles (Oi) that were successfully executed for each task. In contrast, the "Ideal QoS" represents the theoretical best-case scenario, where all possible optional cycles are executed without any limitations. The ratio $\frac{\sum \text{Actual QoS}}{\sum \text{Ideal QoS}}$ thus provides a comparison of the system's real-world performance against this ideal scenario. Similarly, the equation can be interpreted in terms of optional instructions or cycles, where "Actual Oi" refers to the sum of optional

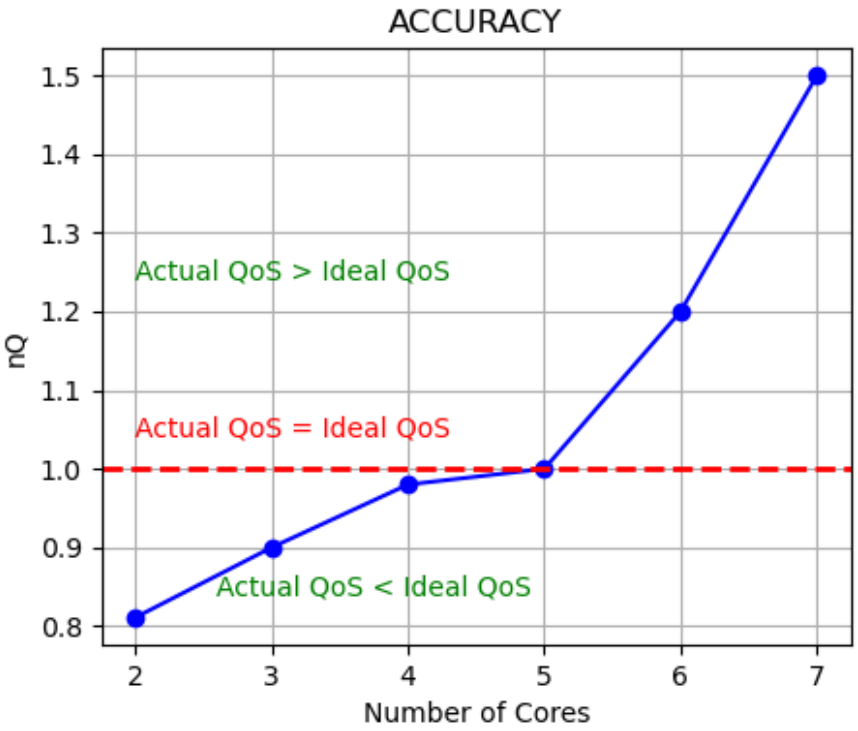

Figure 4.10: Accuracy

instructions that were executed, and "Ideal Oi" refers to the total possible optional instructions that could have been executed under optimal conditions. This metric, NQ, effectively captures the efficiency and effectiveness of the task scheduling algorithm by comparing actual performance to the ideal, thereby identifying how closely the system operates to its maximum potential.

In this project, NQ is calculated as:

$$NQ = \frac{\sum \text{Actual Oi}}{\sum \text{Ideal Oi}} = \frac{2+10+6+2+20+3}{2+10+6+20+5+9} \times 100 = 81.1\%$$

In addition to defining NQ, a graph plotting NQ versus the number of cores is created to further analyze system performance. This graph illustrates that as the number of cores increases, the system's availability also rises, allowing for better scheduling of tasks, particularly parallel tasks, with their maximum optional cycles. [17] The increase in the number of cores provides more computational resources, thereby enabling the system to more closely match the Ideal QoS. As the number of cores grows, the Actual

QoS approaches the Ideal QoS, indicating that the system is able to handle tasks more efficiently and effectively, especially under parallel execution conditions. This behavior demonstrates that with sufficient cores, the system can optimize the execution of tasks to their full potential, minimizing the gap between actual and ideal performance and thereby improving overall system accuracy.

CHAPTER 5

Conclusions

Overall, the project successfully achieved its primary goal of developing a task scheduling algorithm for heterogeneous multicore systems, specifically targeting CPU-GPU real-time environments. The algorithm effectively scheduled tasks based on their dependencies, ensuring real-time performance while optimizing power consumption. Critical high tasks were efficiently managed on the GPU, while critical low tasks were allocated to the CPU, demonstrating a well-balanced approach to resource utilization. The implementation on the NVIDIA Jetson Nano provided a practical platform for testing, with the system maintaining power levels within acceptable limits and achieving an 80 percentage near-quality (nq) threshold. With lessons learned in time management and a deeper understanding of heterogeneous systems, future work could further refine and expand the capabilities of this scheduling algorithm.

5.1 Future work

Future work should address the limitations encountered in scaling the scheduling algorithm for larger and more complex real-world applications. Enhancing the algorithm's ability to efficiently manage multiple sets of tasks with numerous nodes and varying versions is critical. This could involve refining the algorithm to reduce computational load and increase efficiency, particularly when dealing with large-scale data and multiple cores.

In addition, integrating AI and machine learning techniques could provide significant improvements in real-time adaptability and performance optimization. These technologies could help the algorithm predict task behaviors, manage continuous data streams more effectively, and adjust scheduling dynamically to meet varying demands.

Further, the current data management approach, which relies on Python and basic programming techniques, may not suffice for handling extensive data sets common in practical applications. Future efforts should explore more advanced data processing tools and real-time analysis techniques to ensure that the system can maintain high performance even under more demanding conditions.

By addressing these areas, the scheduling algorithm can be made more robust, scalable, and adaptable, making it suitable for a wider range of complex, real-world heterogeneous computing environments.

Bibliography

[1] Z. Zhu, J. Zhang, J. Zhao, J. Cao, D. Zhao, G. Jia, and Q. Meng, "A hardware and software task-scheduling framework based on cpu+ fpga heterogeneous architecture in edge computing," *IEEE Access*, vol. 7, pp. 148975–148988, 2019.

[2] I. Behera and S. Sobhanayak, "Task scheduling optimization in heterogeneous cloud computing environments: A hybrid ga-gwo approach," *Journal of Parallel and Distributed Computing*, vol. 183, p. 104766, 2024.

[3] H. Wang and W. Chen, "Task scheduling for heterogeneous agents pickup and delivery using recurrent open shop scheduling models," *Robotics and Autonomous Systems*, vol. 172, p. 104604, 2024.

[4] M. Hosseinzadeh, E. Azhir, J. Lansky, S. Mildeova, O. H. Ahmed, M. H. Malik, and F. Khan, "Task scheduling mechanisms for fog computing: A systematic survey," *IEEE Access*, vol. 11, pp. 50994–51017, 2023.

[5] J. Xu, K. Li, and Y. Chen, "Real-time task scheduling for fpga-based multicore systems with communication delay," *Microprocessors and Microsystems*, vol. 90, p. 104468, 2022.

[6] S. Saha, S. Chakraborty, S. Agarwal, R. Gangopadhyay, M. Själander, and K. McDonald-Maier, "Delicious: Deadline-aware approximate computing in cache-conscious multicore," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 718–733, 2022.

[7] NVIDIA, "Jetson nano product development," 2024. Accessed: 2024-08-26.

[8] X. Li, L. Mo, A. Kritikakou, and O. Sentieys, "Approximation-aware task deployment on heterogeneous multicore platforms with dvfs," *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, vol. 42, no. 7, pp. 2108–2121, 2022.

[9] S. Z. Sheikh and M. A. Pasha, "Energy-efficient cache-aware scheduling on heterogeneous multicore systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 206–217, 2021.

[10] J. C. Kao, G. H. Ma, C. Y. Lee, C. F. Kuo, and J. H. Hong, "Load-balancing and prudent deployment of vnfs for heterogeneous multicore systems," in *2024 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 01–06, IEEE, April 2024.

[11] R. Jeyaraj and A. Paul, "Optimizing mapreduce task scheduling on virtualized heterogeneous environments using ant colony optimization," *IEEE Access*, vol. 10, pp. 55842–55855, 2022.

[12] A. Dorflinger, M. Albers, J. Schlatow, B. Fiethe, H. Michalik, P. Keldenich, and P. S'andor, "Hardware and software task scheduling for arm-fpga platforms," in *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 66–73, IEEE, IEEE, 2018.

[13] M. Adnan, Y. E. Maboud, D. Mahajan, and P. J. Nair, "Heterogeneous acceleration pipeline for recommendation system training," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 1063–1079, IEEE, June 2024.

[14] Z. Han, R. Zhou, C. Xu, Y. Zeng, and R. Zhang, "InSS: An Intelligent Scheduling Orchestrator for Multi-GPU Inference with Spatio-Temporal Sharing," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[15] Z. Jin and J. S. Vetter, "Evaluating unified memory performance in hip," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 562–568, IEEE, May 2022.

[16] Z. Wang, Z. Jiang, Z. Wang, X. Tang, C. Liu, S. Yin, and Y. Hu, "Enabling latency-aware data initialization for integrated cpu/gpu heterogeneous platform," *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, pp. 3433–3444, 2020.

[17] D. Senapati, A. Sarkar, and C. Karfa, "Energy-aware real-time scheduling of multiple periodic dags on heterogeneous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 8, pp. 2447–2460, 2022.

[18] S. Zhao, X. Dai, I. Bate, A. Burns, and W. Chang, "Dag scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, pp. 128–140, IEEE, December 2020.

[19] H.-E. Zuhair, N. Capodieci, R. Cavicchioli, G. Lipari, and M. Bertogna, "The hpc-dag task model for heterogeneous real-time systems," *IEEE Transactions on Computers*, vol. 70, no. 10, pp. 1747–1761, 2020.

[20] J. Sun, N. Guan, F. Li, H. Gao, C. Shi, and W. Yi, "Real-time scheduling and analysis of openmp dag tasks supporting nested parallelism," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1335–1348, 2020.

[21] H. Yuan, M. Zhou, Q. Liu, and A. Abusorrah, "Fine-grained resource provisioning and task scheduling for heterogeneous applications in distributed green clouds," *IEEE/CAA Journal of Automatica Sinica*, vol. 7, no. 5, pp. 1380–1393, 2020.

[22] X. Tang, W. Cao, H. Tang, T. Deng, J. Mei, Y. Liu, C. Shi, M. Xia, and Z. Zeng, "Cost-efficient workflow scheduling algorithm for applications with deadline constraint on heterogeneous clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 9, pp. 2079–2092, 2021.

[23] A. Zou, J. Li, C. D. Gill, and X. Zhang, "Rtgpu: Real-time gpu scheduling of hard deadline parallel tasks with fine-grain utilization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1450–1465, 2023.

[24] X. Tang and Z. Fu, "Cpu–gpu utilization aware energy-efficient scheduling algorithm on heterogeneous computing systems," *IEEE Access*, vol. 8, pp. 58948–58958, 2020.

[25] A. Yeganeh-Khaksar, M. Ansari, S. Safari, S. Yari-Karin, and A. Ejlali, "Ring-dvfs: Reliability-aware reinforcement learning-based dvfs for real-time embedded systems," *IEEE Embedded Systems Letters*, vol. 13, no. 3, pp. 146–149, 2020.

[26] NVIDIA, *Jetson Nano Developer Kit User Guide*, 2024. Accessed: 2024-08-26.

[27] G. Kuncham, R. Vaidya, and M. Barve, "Performance study of gpu applications using sycl and cuda on tesla v100 gpu," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, IEEE, 2021.

[28] S. Thorgeirsson, T. B. Weidmann, K. H. Weidmann, and Z. Su, "Comparing cognitive load among undergraduate students programming in python and the visual language algot," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, pp. 1328–1334, ACM, March 2024.

[29] L. Quaranta, F. Calefato, and F. Lanubile, "Kgtorrent: A dataset of python jupyter notebooks from kaggle," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 550–554, IEEE, May 2021.

[30] M. Ravasi and I. Vasconcelos, "Pylops—a linear-operator python library for scalable algebra and optimization," *SoftwareX*, vol. 11, p. 100361, 2020.

[31] M. I. Uddin, M. S. Alamgir, M. M. Rahman, M. S. Bhuiyan, and M. A. Moral, "Ai traffic control system based on deepstream and iot using nvidia jetson nano," in *2021 2nd International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST)*, pp. 115–119, IEEE, January 2021.

APPENDIX A

A Long Proof

Python Program for Task Scheduling

```
import numpy as np
import pandas as pd
from collections import defaultdict
from itertools import combinations
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from collections import deque
import pprint
import json
import pickle

# INPUT SECTION - Task data including versions and their attributes for multiple tasks
tasks_data = {
    'Task1': {
        'Versions': [
            {'CL': {'Mi': 4, 'Oi': 2, 'Pow_i': 10, 'Yi,L': 0.5}, 'CH': {'Mi': 4, 'Oi': 2, 'Pow_i': 13, 'Yi,H': 0.7}},
        ]
    },
    'Task2': {
```

```
        'Versions': [
            {'CL': {'Mi': 10, 'Oi': 5, 'Pow_i': 20, 'Yi,L': 0.4}, 'CH': {'Mi': 10, 'Oi': 5, 'Pow_i': 26, 'Yi,H': 0.6}},
            {'CL': {'Mi': 10, 'Oi': 8, 'Pow_i': 22, 'Yi,L': 0.4}, 'CH': {'Mi': 10, 'Oi': 8, 'Pow_i': 29, 'Yi,H': 0.6}},
            {'CL': {'Mi': 10, 'Oi': 10, 'Pow_i': 24, 'Yi,L': 0.4}, 'CH': {'Mi': 10, 'Oi': 10, 'Pow_i': 30, 'Yi,H': 0.6}},
        ]
    },
    'Task3': {
        'Versions': [
            {'CL': {'Mi': 10, 'Oi': 2, 'Pow_i': 10, 'Yi,L': 0.5}, 'CH': {'Mi': 10, 'Oi': 2, 'Pow_i': 12, 'Yi,H': 0.6}},
            {'CL': {'Mi': 10, 'Oi': 4, 'Pow_i': 12, 'Yi,L': 0.5}, 'CH': {'Mi': 10, 'Oi': 4, 'Pow_i': 14, 'Yi,H': 0.6}},
            {'CL': {'Mi': 10, 'Oi': 6, 'Pow_i': 15, 'Yi,L': 0.5}, 'CH': {'Mi': 10, 'Oi': 6, 'Pow_i': 17, 'Yi,H': 0.6}},
        ]
    },
    'Task4': {
        'Versions': [
            {'CL': {'Mi': 28, 'Oi': 16, 'Pow_i': 15, 'Yi,L': 0.4}, 'CH': {'Mi': 28, 'Oi': 16, 'Pow_i': 18, 'Yi,H': 0.8}},
            {'CL': {'Mi': 28, 'Oi': 20, 'Pow_i': 18, 'Yi,L': 0.4}, 'CH': {'Mi': 28, 'Oi': 20, 'Pow_i': 21, 'Yi,H': 0.8}},
        ]
    },
    'Task5': {
        'Versions': [
            {'CL': {'Mi': 8, 'Oi': 2, 'Pow_i': 20, 'Yi,L': 0.5}, 'CH': {'Mi': 8, 'Oi': 2, 'Pow_i': 25, 'Yi,H': 0.7}},
            {'CL': {'Mi': 8, 'Oi': 4, 'Pow_i': 24, 'Yi,L': 0.5}, 'CH': {'Mi': 8, 'Oi': 4, 'Pow_i': 28, 'Yi,H': 0.7}},
            {'CL': {'Mi': 8, 'Oi': 5, 'Pow_i': 27, 'Yi,L': 0.5}, 'CH': {'Mi': 8, 'Oi': 5, 'Pow_i': 31, 'Yi,H': 0.7}},
        ]
    },
    'Task6': {
        'Versions': [
```

```
            {'CL': {'Mi': 10, 'Oi': 3, 'Pow_i': 10, 'Yi,L': 0.5}, 'CH': {'Mi': 10, 'Oi': 3, 'Pow_i': 12, 'Yi,H': 0.7}},
            {'CL': {'Mi': 10, 'Oi': 6, 'Pow_i': 12, 'Yi,L': 0.5}, 'CH': {'Mi': 10, 'Oi': 6, 'Pow_i': 14, 'Yi,H': 0.7}},
            {'CL': {'Mi': 10, 'Oi': 9, 'Pow_i': 14, 'Yi,L': 0.5}, 'CH': {'Mi': 10, 'Oi': 9, 'Pow_i': 16, 'Yi,H': 0.7}},
        ]
    }
}

# Constants
Deadline = 142  # Deadline for completing the tasks
Number_of_types = 2  # Number of types (CL, CH)
types = ['CL', 'CH']  # Labels for types
MAX_POWER_FACTOR = 50  # Maximum allowed sum of power factors for task combinations

# GRAPH AND TOPOLOGICAL SORT

# Class representing a directed graph using adjacency list
class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)  # Dictionary containing adjacency lists
        self.V = vertices  # Number of vertices

    def addEdge(self, u, v):
        # Add an edge from vertex u to vertex v
        self.graph[u].append(v)

    def neighbor_gen(self, v):
        # Generate neighbors of vertex v
        for k in self.graph[v]:
            yield k

    def nonRecursiveTopologicalSortUtil(self, v, visited, stack):
        # Non-recursive function to perform topological sort
        working_stack = [(v, self.neighbor_gen(v))]  # Initialize stack with current node
```

```
82
83          while working_stack:
84              v, gen = working_stack.pop()  # Pop the vertex and generator
    from stack
85              visited[v] = True  # Mark the current node as visited
86
87              for next_neighbor in gen:  # Iterate through neighbors
88                  if not visited[next_neighbor]:  # If neighbor not visited
89                      working_stack.append((v, gen))  # Push the current
    state back to stack
90                      working_stack.append((next_neighbor, self.neighbor_gen(
    next_neighbor)))  # Add the neighbor to stack
91                      break
92              else:
93                  stack.append(v)  # Push the vertex to stack when all
    neighbors are processed
94
95      def nonRecursiveTopologicalSort(self):
96          # Perform topological sort on all vertices
97          visited = [False] * self.V  # Initialize all vertices as not
    visited
98          stack = []  # Stack to store the topological order
99
100         for i in range(1, self.V):
101             if not visited[i]:  # If vertex is not visited
102                 self.nonRecursiveTopologicalSortUtil(i, visited, stack)  #
    Call the util function
103
104         stack.reverse()  # Reverse the stack to get the topological order
105         return stack
106
107     def find_successors(self, start_task):
108         # Find all successors of a given task
109         visited = set()  # Set to track visited nodes
110         successors = []  # List to store successors
111
112         def dfs(v):
113             if v in visited:  # If node already visited, return
114                 return
```

```
            visited.add(v)  # Mark node as visited
            for neighbor in self.graph[v]:  # Iterate through neighbors
                if neighbor not in visited:  # If neighbor not visited
                    successors.append(neighbor)  # Add neighbor to
    successors
                    dfs(neighbor)  # Recursively visit neighbor

        dfs(start_task)  # Start DFS from the given task
        return successors

# Create and populate the graph with edges
g = Graph(7)
g.addEdge(1, 2)
g.addEdge(1, 3)
g.addEdge(1, 5)
g.addEdge(2, 4)
g.addEdge(3, 6)
g.addEdge(4, 6)
g.addEdge(5, 6)

# Get the topological sort of tasks
topological_sort = g.nonRecursiveTopologicalSort()
task_list = [f'Task{num}' for num in topological_sort]  # Convert to task
    format

# Print the topological sort result
print("\nTopological Sort of Tasks:")
print(topological_sort)

# SCHEDULING PARALLEL TASKS

# Define parallel tasks (tasks that can be done in parallel)
parallel_tasks = [
    [2, 3, 5]  # Example set of parallel tasks
]

# Initialize a dictionary to store the results for parallel task successors
Sorted_results = {"Parallel Tasks Successors": {}}

```

```
# Loop through each set of parallel tasks
for index, task_set in enumerate(parallel_tasks, start=1):
    # Initialize a dictionary for the current set of parallel tasks
    Sorted_results["Parallel Tasks Successors"][f"Parallel Task Set {index}"] = {}

    # Loop through each task in the current set of parallel tasks
    for task in task_set:
        successors = g.find_successors(task)  # Find successors of the current task

        # Store the number of successors and the successor tasks in the result dictionary
        Sorted_results["Parallel Tasks Successors"][f"Parallel Task Set {index}"][task] = {
            "Number of Successors": len(successors),
            "Successor Tasks": successors
        }

# Print the successors for each set of parallel tasks
print("\nSuccessors of Parallel Tasks:")
pprint.pprint(Sorted_results)

# EXECUTION TIME CALCULATIONS

# Function to calculate execution time for each version of each task
def calculate_execution_time(data):
    # Loop through each task in the data
    for task, task_data in data.items():
        # Loop through each version of the task
        for version in task_data['Versions']:
            cl = version['CL']  # Get the 'CL' type data
            # Calculate execution time for 'CL' version
            cl_execution_time = round((cl['Mi'] + cl['Oi']) / (0.5 * cl['Yi,L']))
            cl['Execution_Time'] = cl_execution_time  # Store the calculated execution time

```

```
            ch = version.get('CH', {})  # Get the 'CH' type data, if it exists
            if ch:
                # Calculate execution time for 'CH' version
                ch_execution_time = round((ch['Mi'] + ch['Oi']) / ch['Yi,H'])
                ch['Execution_Time'] = ch_execution_time  # Store the calculated execution time

# Call the function to calculate execution times for all tasks
calculate_execution_time(tasks_data)

# Print tasks with calculated execution times
print("\nTasks with Calculated Execution Times:")
pprint.pprint(tasks_data)

# CLASSIFICATION BASED ON SUCCESSORS (CH/CL)

# Initialize dictionaries to classify tasks into 'CH' and 'CL'
task_classification = {'CH': [], 'CL': []}

# Loop through each set of parallel tasks in the sorted results
for task_set, tasks in Sorted_results['Parallel Tasks Successors'].items():
    # Loop through each task in the parallel task set
    for task, details in tasks.items():
        num_successors = details['Number of Successors']  # Get the number of successors
        task_name = f"Task{task}"  # Convert task number to task name
        if num_successors > 1:  # If more than one successor
            task_classification['CH'].append(task_name)  # Classify as 'CH'
        else:
            task_classification['CL'].append(task_name)  # Classify as 'CL'

# Print task classification into 'CH' and 'CL'
print("\nTask Classification (CH/CL):")
pprint.pprint(task_classification)

# POWER FACTOR AND VERSION SELECTION

```

```
# Initialize a dictionary to store the selected versions and their details
result = {}

# Loop through each 'CH' task
for ch_task in task_classification['CH']:
    if ch_task in tasks_data:  # Check if the task exists in the data
        ch_versions = tasks_data[ch_task]['Versions']  # Get the versions of the task
        highest_ch_version = len(ch_versions) - 1  # Get the index of the highest version
        ch_data = ch_versions[highest_ch_version]['CH']  # Get the 'CH' data for the highest version
        ch_power_factor = ch_data['Pow_i']  # Get the power factor for the 'CH' version
        ch_execution_time = ch_data.get('Execution_Time', None)  # Get the execution time for the 'CH' version

        # Store the selected version and its details for the 'CH' task
        result[ch_task] = {
            'CH': {
                'Version': f'Version {highest_ch_version + 1}',
                'Power_Factor': ch_power_factor,
                'Execution_Time': ch_execution_time
            }
        }

        # Loop through each 'CL' task
        for cl_task in task_classification['CL']:
            if cl_task in tasks_data:  # Check if the task exists in the data
                cl_versions = tasks_data[cl_task]['Versions']  # Get the versions of the task
                chosen_version_index = len(cl_versions) - 1  # Start with the highest version index
                cl_data = cl_versions[chosen_version_index]['CL']  # Get the 'CL' data for the chosen version
                cl_power_factor = cl_data['Pow_i']  # Get the power factor for the 'CL' version
```

```
                cl_execution_time = cl_data.get('Execution_Time', None)  # Get the execution time for the 'CL' version

                # Reduce the version index until the sum of power factors is within the limit
                while (ch_power_factor + cl_power_factor) > MAX_POWER_FACTOR and chosen_version_index > 0:
                    chosen_version_index -= 1  # Decrease the version index
                    cl_data = cl_versions[chosen_version_index]['CL']  # Update the 'CL' data
                    cl_power_factor = cl_data['Pow_i']  # Update the power factor
                    cl_execution_time = cl_data.get('Execution_Time', None)  # Update the execution time

                # Store the selected version and its details for the 'CL' task
                result[cl_task] = {
                    'CL': {
                        'Version': f'Version {chosen_version_index + 1}',
                        'Power_Factor': cl_power_factor,
                        'Execution_Time': cl_execution_time
                    }
                }

# Print the selected versions with power factor considerations
print("\nSelected Versions with Power Factor Considerations:")
pprint.pprint(result)

# UPDATED TASKS

# Initialize a dictionary to store the final updated tasks with selected versions
final_result = {}

# Loop through each task in the result
for task, details in result.items():
    task_type, task_data = list(details.items())[0]  # Get the task type (CL/CH) and its data
```

```
277     version_number = int(task_data['Version'].split(' ')[1]) - 1  # Extract
        the version number
278     updated_task_data = tasks_data[task]['Versions'][version_number][
        task_type]  # Get the updated task data
279     final_result[task] = {task_type: updated_task_data}  # Store the
        updated task data in the final result
280
281 # Print the final updated tasks with selected versions
282 print("\nFinal Updated Tasks with Selected Versions:")
283 pprint.pprint(final_result)
284
285 # NUMBER OF LEVELS & TASKS AT EACH LEVEL
286
287 # Function to calculate the number of levels in the task graph and tasks at
        each level
288 def calculate_levels(graph):
289     topo_sort = list(nx.topological_sort(graph))  # Get the topological
        sort of the graph
290     levels = {node: 0 for node in graph.nodes}  # Initialize levels for
        each node
291     tasks_by_level = {}  # Dictionary to store tasks by their level
292
293     # Loop through each node in the topological order
294     for node in topo_sort:
295         current_level = levels[node]  # Get the current level of the node
296         # Update the level of each successor to be one higher than the
        current node
297         for successor in graph.successors(node):
298             levels[successor] = max(levels[successor], current_level + 1)
299
300     # Organize tasks by their levels
301     for node, level in levels.items():
302         if level not in tasks_by_level:
303             tasks_by_level[level] = []
304         tasks_by_level[level].append(node)
305
306     max_level = max(levels.values())  # Get the maximum level in the graph
307     total_levels = max_level + 1  # Calculate the total number of levels
308
```

```
    return total_levels, tasks_by_level

# Initialize a directed graph and add nodes and edges for tasks
task_graph = nx.DiGraph()
nodes = range(1, 7)
task_graph.add_nodes_from(nodes)
edges = [
    (1, 2),
    (1, 3),
    (1, 5),
    (2, 4),
    (3, 6),
    (4, 6),
    (5, 6)
]
task_graph.add_edges_from(edges)

# Calculate the total levels and tasks at each level in the graph
total_levels, tasks_by_level = calculate_levels(task_graph)

# Print the number of levels and tasks at each level
print("\nTotal Number of Levels:", total_levels)
print("Tasks at Each Level:")
pprint.pprint(tasks_by_level)

# ASSIGNING TASKS INTO MATRICES

# Initialize matrices for execution times and task names
execution_time_matrix = np.zeros((2, total_levels), dtype=object)
task_name_matrix = np.full((2, total_levels), None, dtype=object)

# Initialize counters for 'CL' and 'CH' tasks
count_cl = 0
count_ch = 0

# Loop through the tasks in the topological order
for task in task_list:
    if task in final_result:  # Check if the task is in the final result
```

```
347         task_type = list(final_result[task].keys())[0]  # Get the task type
    (CL/CH)
348         execution_time = final_result[task][task_type]['Execution_Time']  #
    Get the execution time
349 
350         if task_type == 'CL':
351             count_cl += 1  # Increment the 'CL' task counter
352             if count_cl < total_levels:  # Ensure we stay within matrix
    bounds
353                 execution_time_matrix[0][count_cl] = execution_time  #
    Assign execution time to matrix
354                 task_name_matrix[0][count_cl] = task  # Assign task name to
    matrix
355         elif task_type == 'CH':
356             count_ch += 1  # Increment the 'CH' task counter
357             if count_ch < total_levels:  # Ensure we stay within matrix
    bounds
358                 execution_time_matrix[1][count_ch] = execution_time  #
    Assign execution time to matrix
359                 task_name_matrix[1][count_ch] = task  # Assign task name to
    matrix
360 
361 # Set the initial column (first task) to zero
362 execution_time_matrix[0][0] = 0
363 execution_time_matrix[1][0] = 0
364 task_name_matrix[0][0] = None
365 task_name_matrix[1][0] = None
366 
367 # Print the initial matrices for execution times and task names
368 print("\nInitial Execution Time Matrix:")
369 print(execution_time_matrix)
370 print("\nInitial Task Name Matrix:")
371 print(task_name_matrix)
372 
373 # Determine which columns in the matrix are empty
374 zero_columns = []
375 for col in range(total_levels):
376     if execution_time_matrix[0][col] == 0 and execution_time_matrix[1][col]
    == 0:
```

```
            zero_columns.append(col + 1)  # Store empty column indices

# HANDLE MISSING TASKS

# Convert zero columns to levels
zero_levels = [col - 1 for col in zero_columns]
tasks_at_zero_levels = []

# Identify tasks at levels that correspond to zero columns
for level in zero_levels:
    if level in tasks_by_level:
        tasks_at_zero_levels.extend(tasks_by_level[level])

# Assign missing tasks to the matrices based on the best available version
for task in tasks_at_zero_levels:
    task_key = f"Task{task}"  # Convert task number to task name
    if task_key in tasks_data:
        versions = tasks_data[task_key]['Versions']  # Get all versions for
    the task
        max_oi = -1  # Initialize maximum Oi value
        max_version = None

        # Loop through versions to find the one with the maximum Oi value
        for version in versions:
            for type_key, data in version.items():
                if data['Oi'] > max_oi:
                    max_oi = data['Oi']
                    max_version = data

        least_execution_time = float('inf')  # Initialize least execution
    time
        best_type = None

        # Loop through versions again to find the one with the least
    execution time
        for version in versions:
            for type_key, data in version.items():
                if data['Execution_Time'] < least_execution_time:
                    least_execution_time = data['Execution_Time']
```

```
                    best_type = type_key

        # Assign the task to the matrix based on its type (CL/CH)
        index = zero_levels[tasks_at_zero_levels.index(task)]
        if best_type == 'CL':
            execution_time_matrix[0][index] = least_execution_time
            task_name_matrix[0][index] = task_key
        elif best_type == 'CH':
            execution_time_matrix[1][index] = least_execution_time
            task_name_matrix[1][index] = task_key

# Print matrices after handling missing tasks
print("\nExecution Time Matrix After Handling Missing Tasks:")
print(execution_time_matrix)
print("\nTask Name Matrix After Handling Missing Tasks:")
print(task_name_matrix)

# FIND AND HANDLE MISSING TASKS IN MATRICES

# Identify tasks that are missing from the matrices
tasks_in_matrix = set(task_name for row in task_name_matrix for task_name
    in row if task_name is not None)
task_list_set = set(task_list)
missing_tasks = task_list_set - tasks_in_matrix

# Assign missing tasks to the appropriate rows and columns
for missing_task in missing_tasks:
    task_number = int(missing_task.replace('Task', ''))  # Extract task
    number

    task_level = None
    # Determine the level of the missing task
    for level, tasks in tasks_by_level.items():
        if task_number in tasks:
            task_level = level
            break

    # If the task level is identified, find an empty slot in the matrices
    if task_level is not None:
```

```
450         respective_column = task_level  # The corresponding column in the
    matrix
451         if execution_time_matrix[0][respective_column] == 0:
452             available_row = 0  # Assign to row 0 (CL)
453             row_type = 'CL'
454         elif execution_time_matrix[1][respective_column] == 0:
455             available_row = 1  # Assign to row 1 (CH)
456             row_type = 'CH'
457         else:
458             available_row = None  # No available row
459 
460         # Assign the task to the matrix based on the selected version
461         if available_row is not None:
462             max_oi = -1  # Initialize maximum Oi value
463             selected_version = None
464             # Loop through versions to select the one with the highest Oi
    value
465             for version in tasks_data[missing_task]['Versions']:
466                 if version[row_type]['Oi'] > max_oi:
467                     max_oi = version[row_type]['Oi']
468                     selected_version = version[row_type]
469 
470             # Assign the selected task version to the matrix
471             if selected_version:
472                 execution_time_matrix[available_row][respective_column] =
    selected_version['Execution_Time']
473                 task_name_matrix[available_row][respective_column] =
    missing_task
474 
475 # Print final matrices after assigning missing tasks
476 print("\nFinal Execution Time Matrix:")
477 print(execution_time_matrix)
478 print("\nFinal Task Name Matrix:")
479 print(task_name_matrix)
480 
481 # Initialize the result dictionary for storing final task selections
482 result = {}
483 
484 # Iterate through the task_name_matrix and execution_time_matrix
```

```
rows, cols = task_name_matrix.shape
for row in range(rows):
    for col in range(cols):
        task_name = task_name_matrix[row, col]  # Get the task name from the matrix
        if task_name:
            execution_time = execution_time_matrix[row, col]  # Get the corresponding execution time
            versions = tasks_data[task_name]['Versions']  # Get all versions for the task

            # Determine whether the task is CL or CH based on the execution time
            selected_version = None
            for version in versions:
                cl_data = version.get('CL', {})
                ch_data = version.get('CH', {})

                if cl_data.get('Execution_Time') == execution_time:  # Match execution time with CL
                    selected_version = cl_data
                    result[task_name] = {'CL': selected_version}
                elif ch_data.get('Execution_Time') == execution_time:  # Match execution time with CH
                    selected_version = ch_data
                    result[task_name] = {'CH': selected_version}

# Print the final result dictionary containing selected versions for tasks
print("\nFinal Task Version Selections:")
pprint.pprint(result)
print()

# PROCESSING TASK EXECUTION SEQUENCE

# Initialize a dictionary to store the start and finish times of tasks
start_finish_times = {}

# Process each column in the matrices to understand the task execution flow
for col in range(task_name_matrix.shape[1]):
```

```
    for row in range(task_name_matrix.shape[0]):
        task_name = task_name_matrix[row, col]  # Get the task name from the matrix
        execution_time = execution_time_matrix[row, col]  # Get the corresponding execution time

        if task_name is not None:  # If there is a task in this cell
            task_type = 'CH' if row == 1 else 'CL'  # Determine the task type based on the row

            if row == 1 and col == 0:  # Task1 is executed first
                start_time = 0
            else:  # For other tasks, determine the start time based on predecessor tasks
                if task_name == 'Task6':
                    # Task6 starts after Task3, Task5, and Task4 are completed
                    predecessor_tasks = ['Task3', 'Task5', 'Task4']
                elif task_name == 'Task5':
                    # Task5 starts after Task1 is completed
                    predecessor_tasks = ['Task1']
                elif task_name == 'Task3':
                    # Task3 starts after Task5 is completed
                    predecessor_tasks = ['Task5']
                elif task_name == 'Task2':
                    # Task2 starts after Task1 is completed
                    predecessor_tasks = ['Task1']
                elif task_name == 'Task4':
                    # Task4 starts after Task2 is completed
                    predecessor_tasks = ['Task2']

                # Find the maximum finish time among the predecessor tasks
                start_time = max(
                    start_finish_times[predecessor][2]
                    for predecessor in predecessor_tasks
                    if predecessor in start_finish_times
                )

```

```
551             finish_time = start_time + execution_time  # Calculate the
    finish time for the task
552             start_finish_times[task_name] = (task_type, start_time,
    finish_time)  # Store start and finish times
553
554 # Store task execution sequence in a dictionary with the requested format
555 task_execution_sequence = {}
556
557 # Loop through the start and finish times to create the execution sequence
558 for task, (task_type, start, finish) in start_finish_times.items():
559     task_execution_sequence[task] = {
560         'type': task_type,
561         'Start_Time': start,
562         'Finish_Time': finish
563     }
564
565 # Print the dictionary to see the final task execution sequence
566 print("Task Execution Sequence Dictionary:")
567 print(task_execution_sequence)
568 print()
569
570 # Identify columns with no zeros (valid columns)
571 valid_columns = np.all(execution_time_matrix != 0, axis=0)
572 columns_with_zeros = ~valid_columns
573
574 # Filter the matrices to get only the valid columns
575 filtered_execution_time_matrix = execution_time_matrix[:, valid_columns]
576 filtered_task_name_matrix = task_name_matrix[:, valid_columns]
577
578 # Initialize a list to store the unique task combinations and their
    corresponding power factors
579 parallel_task_power_factors = []
580
581 # Use a set to keep track of seen combinations (unordered pairs)
582 seen_combinations = set()
583
584 # Function to get the power factor from the result dictionary
585 def get_power_factor(task, execution_time):
586     task_data = result.get(task)
```

```
    if task_data:
        for task_type, details in task_data.items():
            if details['Execution_Time'] == execution_time:
                return details['Pow_i']
    return None

# Generate all combinations of row1 + row2 across valid columns using power factors
for i in range(filtered_execution_time_matrix.shape[1]):
    for j in range(i, filtered_execution_time_matrix.shape[1]):
        task1_row1 = filtered_task_name_matrix[0, i]
        task2_row2 = filtered_task_name_matrix[1, j]

        if task1_row1 and task2_row2:
            pow_i1 = get_power_factor(task1_row1, filtered_execution_time_matrix[0, i])
            pow_i2 = get_power_factor(task2_row2, filtered_execution_time_matrix[1, j])
            if pow_i1 is not None and pow_i2 is not None:
                tasks = tuple(sorted([task1_row1, task2_row2]))
                if tasks not in seen_combinations:
                    combined_power = pow_i1 + pow_i2
                    parallel_task_power_factors.append((f'{tasks[0]} + {tasks[1]}', combined_power))
                    seen_combinations.add(tasks)

        # Also add the reverse combination (row2 + row1)
        task1_row2 = filtered_task_name_matrix[1, i]
        task2_row1 = filtered_task_name_matrix[0, j]

        if task1_row2 and task2_row1:
            pow_i1 = get_power_factor(task1_row2, filtered_execution_time_matrix[1, i])
            pow_i2 = get_power_factor(task2_row1, filtered_execution_time_matrix[0, j])
            if pow_i1 is not None and pow_i2 is not None:
                tasks = tuple(sorted([task1_row2, task2_row1]))
                if tasks not in seen_combinations:
                    combined_power = pow_i1 + pow_i2
```

```
                    parallel_task_power_factors.append((f'{tasks[0]} + {tasks[1]}', combined_power))
                    seen_combinations.add(tasks)

# Handle columns with zeros separately
for col in range(execution_time_matrix.shape[1]):
    if columns_with_zeros[col]:
        task = task_name_matrix[1, col]  # Take the task from row 2, as row 1 has zero
        if task:
            sum_value = get_power_factor(task, execution_time_matrix[1, col])
            if sum_value is not None:
                parallel_task_power_factors.append((task, sum_value))

# Sort the combined list of tasks and power factors by task name
parallel_task_power_factors.sort()

# Print the sorted list of parallel tasks and their corresponding power factors
print("Parallel tasks and their corresponding power factors (sorted):")
for task in parallel_task_power_factors:
    print(task)
```

Listing A.1: Python Program for Task Scheduling

Python Program for Graph generations

```

# Draw the graph
plt.figure(figsize=(3, 2))
pos = nx.spring_layout(task_graph)  # Position nodes using the spring layout
nx.draw(task_graph, pos, with_labels=True, node_color='red', node_size=500,
        edge_color='black', font_size=11), #font_weight='bold')
#nx.draw_networkx_edge_labels(task_graph, pos, edge_labels={(u, v): f'{u}->{v}' for u, v in edges})
plt.title("Task Dependency Graph")
```

```
plt.show()

print()

# Set up the figure and axis
fig, ax = plt.subplots(figsize=(10, 2))

# Define y-axis positions for CPU and GPU
y_positions = {'CH': 1, 'CL': 2}
colors = {'CH': 'orange', 'CL': 'lightgreen'}

# Plot each task as a rectangle
for task, details in task_execution_sequence.items():
    task_type = details['type']
    start_time = details['Start_Time']
    finish_time = details['Finish_Time']

    # Determine the y position based on the task type
    y_position = y_positions[task_type]

    # Calculate width of the box
    width = finish_time - start_time

    # Draw the rectangle with a height of 0.8 units
    rect = patches.Rectangle((start_time, y_position - 0.4), width, 0.8,
   edgecolor='black', facecolor=colors[task_type])
    ax.add_patch(rect)

    # Add task label and execution time inside the box
    ax.text(start_time + width / 2, y_position, f'{task}\n{start_time}-{
   finish_time}', ha='center', va='center', fontsize=10, color='black')

# Set y-axis limits and labels
ax.set_ylim(0.5, 2.5)  # y-axis to accommodate both CPU and GPU stacked
ax.set_yticks([1, 2])
ax.set_yticklabels(['GPU', 'CPU'])

# Set x-axis limits and labels
ax.set_xlim(0, 150)  # Set x-axis limit
```

```
ax.set_xticks(range(0, Deadline + 1, 10))  # Mark every 5 units on the x-axis
plt.axvline(x=142, color='red', linestyle='--', label=''); plt.text(142, plt.ylim()[1], 'Deadline=142', color='white', backgroundcolor='red', ha='right')


# Add grid lines for better readability
ax.grid(True, which='both', axis='x', linestyle='--', color='gray', alpha=0.7)

# Add labels and title
ax.set_xlabel('Execution Time')
ax.set_ylabel('Cores')
ax.set_title('Actual Case Task Scheduling - Heterogeneous')

plt.savefig('actualcasetaskscheduling.png')
# Show the plot
plt.tight_layout()
plt.show()


#################################################################################

import matplotlib.pyplot as plt
import matplotlib.patches as patches

# Define y-axis positions for CPU and GPU
y_positions = {'CPU': 2, 'GPU': 1}
colors = {'CPU': 'lightgreen', 'GPU': 'orange'}

# Task details manually set according to the specified start and end times
tasks = {
    'CPU': [(9, 61), (61, 125)],
    'GPU': [(0, 9), (9, 42), (42, 106), (125, 152)]
}

# Task names corresponding to each time segment
```

```
task_names = {
    'CPU': ['Task5', 'Task3'],
    'GPU': ['Task1', 'Task2', 'Task4', 'Task6']
}

# Set up the figure and axis
fig, ax = plt.subplots(figsize=(10, 2))

# Plot each task segment on the CPU and GPU
for core_type, core_tasks in tasks.items():
    for i, (start_x, end_x) in enumerate(core_tasks):
        y_position = y_positions[core_type]
        color = colors[core_type]

        # Draw the rectangle representing the task's execution time
        rect = patches.Rectangle((start_x, y_position - 0.4), end_x -
    start_x, 0.8, edgecolor='black', facecolor=color)
        ax.add_patch(rect)

        # Add the task name and the execution time range inside the box
        task_name = task_names[core_type][i]
        ax.text((start_x + end_x) / 2, y_position + 0.15, task_name, ha='
    center', va='center', fontsize=10, color='black')
        ax.text((start_x + end_x) / 2, y_position - 0.15, f'{start_x}-{
    end_x}', ha='center', va='center', fontsize=8, color='black')

# Set y-axis limits and labels
ax.set_ylim(0.5, 2.5)  # y-axis to accommodate both CPU and GPU rows
ax.set_yticks([1, 2])
ax.set_yticklabels(['GPU', 'CPU'])  # Label y-axis as 'CPU' and 'GPU'

# Set x-axis limits and labels
total_time = max(end for core_tasks in tasks.values() for _, end in
    core_tasks)
ax.set_xlim(0, total_time)  # Set x-axis limit based on the total time
ax.set_xticks(range(0, total_time + 1, 10))  # Mark every 10 units on the x
    -axis

# Draw a red vertical line at x=103 for the deadline
```

```
113 ax.axvline(x=103, color='red', linestyle='--', linewidth=2)
114 ax.text(103, 2.3, 'Deadline=142', color='white', ha='center', va='center',
        fontsize=10, rotation=0, backgroundcolor='red')
115
116 # Add grid lines for better readability
117 ax.grid(True, which='both', axis='x', linestyle='--', color='gray', alpha
        =0.7)
118
119 # Add labels and title
120 ax.set_ylabel('Cores')
121 ax.set_xlabel('Execution Time')
122 ax.set_title('Ideal Case Task Scheduling - Heterogeneous')
123
124 plt.savefig('idealcasetaskscheduling.png')
125 # Show the plot
126 plt.tight_layout()
127 plt.show()
128
129 #
        ##############################################################################################

130 import matplotlib.pyplot as plt
131 import matplotlib.patches as patches
132 import numpy as np
133
134 # Define the input array
135 execution_times = [24, 52, 64, 100, 240, 76]
136 task_names = ['Task1', 'Task5', 'Task3', 'Task2', 'Task4', 'Task6']  # Task
        names corresponding to each segment
137
138 # Set up the figure and axis
139 fig, ax = plt.subplots(figsize=(10, 2))
140
141 # Define y-axis position for CPU
142 y_position = 1
143 color = 'orange'
144
145 # Initialize the starting point of the x-axis
146 current_x = 0
```

```
147
148 # Plot each segment based on the execution times
149 for i, value in enumerate(execution_times):
150     start_x = current_x
151     end_x = current_x + value
152
153     # Draw the rectangle with a height of 0.8 units
154     rect = patches.Rectangle((start_x, y_position - 0.4), value, 0.8,
    edgecolor='black', facecolor=color)
155     ax.add_patch(rect)
156
157     # Add task name and execution time inside the box
158     ax.text((start_x + end_x) / 2, y_position + 0.15, task_names[i], ha='
    center', va='center', fontsize=10, color='black')
159     ax.text((start_x + end_x) / 2, y_position - 0.15, f'{start_x}-{end_x}',
     ha='center', va='center', fontsize=8, color='black')
160
161     # Update current_x to the end_x for the next segment
162     current_x = end_x
163
164 # Set y-axis limits and labels
165 ax.set_ylim(0.5, 1.5)  # y-axis to accommodate CPU row
166 ax.set_yticks([1])
167 ax.set_yticklabels(['CPU'])  # Label y-axis as 'CPU'
168
169 # Set x-axis limits and labels
170 ax.set_xlim(0, np.sum(execution_times))  # Set x-axis limit based on the
    total value
171 ax.set_xticks(range(0, int(np.sum(execution_times)) + 1, 50))  # Mark every
     50 units on the x-axis
172
173 # Draw a red vertical line at x=103 for the deadline
174 ax.axvline(x=103, color='red', linestyle='--', linewidth=2)
175 ax.text(103, 1.3, 'Deadline', color='white', ha='center', va='center',
    fontsize=10, rotation=0, backgroundcolor='red')
176
177 # Add grid lines for better readability
178 ax.grid(True, which='both', axis='x', linestyle='--', color='gray', alpha
    =0.7)
```

```
179
180 # Add labels and title
181 ax.set_ylabel('Cores')
182 ax.set_xlabel('Execution Time')
183 ax.set_title('Ideal case - CPU (Homogeneous)')
184
185 plt.savefig('idealcasecpu.png')
186 # Show the plot
187 plt.tight_layout()
188 plt.show()
189 #
    ##############################################################################
190 import matplotlib.pyplot as plt
191
192 # Define the tasks and their corresponding power factors
193 parallel_task_power_factors = [
194     ('Task1', 13),
195     ('Task2+Task3', 45),
196     ('Task2+Task5', 50),
197     ('Task3+Task4', 36),
198     ('Task4+Task5', 41),
199     ('Task6', 12)
200 ]
201
202 # Separate the tasks and power factors into two lists
203 tasks = [task for task, _ in parallel_task_power_factors]
204 power_factors = [power for _, power in parallel_task_power_factors]
205
206 # Create the bar chart
207 plt.figure(figsize=(8, 4))
208 plt.bar(tasks, power_factors, color='skyblue', edgecolor='black')
209
210 # Add labels and title
211 plt.xlabel('Tasks')
212 plt.ylabel('Power Factor')
213 plt.title('Actual Case Power Factor by Task')
214
215 # Set y-axis limit to 70
```

```
plt.ylim(0, 70)

# Draw a red line at MAX_POWER_FACTOR = 50
plt.axhline(y=50, color='red', linestyle='--', linewidth=2, label='
    MAX_POWER_FACTOR = 50')

# Annotate the bars with their power factor values
for i, value in enumerate(power_factors):
    plt.text(i, value + 1, f'{value}', ha='center', va='bottom')

# Display the legend
plt.legend()

plt.savefig('actualcasepowerfactorbytask.png')

# Show the plot
plt.tight_layout()
plt.show()


#
    ##############################################################################################

import matplotlib.pyplot as plt

# Define the tasks and their corresponding power factors
parallel_task_power_factors = [
    ('Task1', 13),
    ('Task2+Task3', 45),
    ('Task2+Task5', 57),
    ('Task3+Task4', 36),
    ('Task4+Task5', 48),
    ('Task6', 16)
]

# Separate the tasks and power factors into two lists
tasks = [task for task, _ in parallel_task_power_factors]
power_factors = [power for _, power in parallel_task_power_factors]

```

```
# Create the bar chart
plt.figure(figsize=(8, 4))
plt.bar(tasks, power_factors, color='skyblue', edgecolor='black')

# Add labels and title
plt.xlabel('Tasks')
plt.ylabel('Power Factor')
plt.title('Ideal Case Power Factor by Task')

# Set y-axis limit to 70
plt.ylim(0, 70)

# Draw a red line at MAX_POWER_FACTOR = 50
plt.axhline(y=50, color='red', linestyle='--', linewidth=2, label='
    MAX_POWER_FACTOR = 50')

# Annotate the bars with their power factor values
for i, value in enumerate(power_factors):
    plt.text(i, value + 1, f'{value}', ha='center', va='bottom')

# Display the legend
plt.legend()

plt.savefig('idealcasepowerfactorbytask.png')

# Show the plot
plt.tight_layout()
plt.show()

#
    ##############################################################################

import matplotlib.pyplot as plt

# Data for the graph
number_of_cores = [2, 3, 4, 5, 6, 7]
nQ_values = [0.811, 0.9, 0.98, 1, 1.2, 1.5]

# Create the line plot
```

```
plt.figure(figsize=(5, 4))
plt.plot(number_of_cores, nQ_values, marker='o', linestyle='-', color='blue')

# Add labels and title
plt.xlabel('Number of Cores')
plt.ylabel('nQ')
plt.title('ACCURACY')

# Draw a horizontal red line at y=1
plt.axhline(y=1, color='red', linestyle='--', linewidth=2)

# Annotate the red line
plt.text(2.0, 1.05, 'Actual QoS = Ideal QoS', color='red', ha='left', va='center', fontsize=10)
plt.text(2.6, 0.85, 'Actual QoS < Ideal QoS', color='green', ha='left', va='center', fontsize=10)
plt.text(2.0, 1.25, 'Actual QoS > Ideal QoS', color='green', ha='left', va='center', fontsize=10)

# Add grid lines for better readability
plt.grid(True)

plt.savefig('Accuracy.png')

# Show the plot
plt.tight_layout()
plt.show()
```

Listing A.2: Python Program for generating graphs