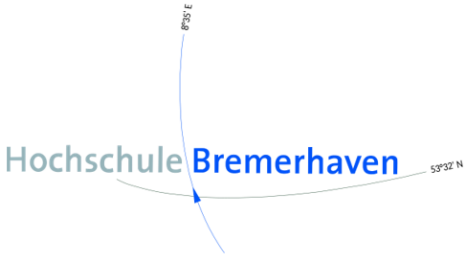


MASTER THESIS

Optimized Task Allocation to Enhanced Multi-core Real-time Scheduling

Author:
Khushbu Gajera

Supervisor:
Dr.-Ing. Zain Alabedin Haj Hammadeh

Supervising tutor: Prof. Dr.-Ing. Oliver Prenzel

A thesis submitted in fulfilment of the requirements for the degree of Master of Science in the

Embedded Systems Design

BREMERHAVEN UNIVERSITY OF APPLIED SCIENCES

February 11, 2024

Declaration of Authorship

I, Khushbu Gajera, declare that this thesis titled, "Optimized task allocation to enhanced multi-core real-time scheduling" and all its work is my own. I confirm that:

- This work was completed entirely or mostly while pursuing a research degree at this university.
- I always give the source when I quote from other people's work.
- Whenever I refer to external published works, I conscientiously acknowledge their contributions.
- I have duly recognized all significant forms of support received.
- I independently authored and composed this present thesis, relying exclusively on the literature and resources referenced within.

Signed: 

Date: February 11, 2024

Abstract

On-boarding software system

Deutsches Zentrum für Luft- und Raumfahrt - DLR

Master of Science

BREMERHAVEN UNIVERSITY OF APPLIED SCIENCES

Optimized task allocation to enhanced multi-core real-time scheduling

by Khushbu Gajera

To handle the increasing computational needs of onboard data processing and complicated control algorithms, modern autonomous systems require high-performance computer resources. With the rise of embedded real-time systems in a variety of industries, including as automotive, avionics, and aerospace, multi-core platforms have evolved as a compromise between performance and power efficiency. However, due to their intrinsic complexity, scheduling and schedulability analysis offer difficulties when switching from single-core to multi-core systems. As a result of the intensive research, global, partitioned and clustered scheduling were proposed.

Arbitrary Processor Affinity (APA) scheduling has been proven to achieve a better schedulability for periodic tasks. APA is a clustered scheduling in which the clusters, i.e., affinity sets, are not necessarily disjoint. APA is implemented in Linux via the pull/push scheduling method, and it is also implemented in the Real-Time Executive for Multiprocessor Systems or RTEMS which is an open source Real-time Operation System (RTOS) by assigning an affinity set to a tasks. A schedulability analysis has been already proposed for APA scheduling. However, optimizing affinity sets for a set of real-time tasks such that to achieve the best scheduling is still an open question.

This research project aims to provide an effective method for defining affinity sets and task assignments, with the ultimate goal of improving schedulability in multi-core real-time systems.

Keywords: multi-core real-time systems, scheduling, affinity sets, schedulability analysis, APA, clustered scheduling, partition-first methodology, bin-packing heuristics, Integer Linear Programming (ILP), resource optimization

Acknowledgements

I would like to extend my heartfelt appreciation to my colleagues at DLR for their invaluable support throughout the duration of my thesis work. Additionally, I want to acknowledge the members of the On-boarding software system for their contributions.

I am deeply grateful to my external supervisor, Dr.-Ing. Zain Alabedin Haj Hammadeh, for believing in my capabilities and entrusting me with the execution of this thesis. His exceptional guidance and unwavering belief have been instrumental to my progress. His practical and theoretical insights have significantly enhanced my understanding of the subject matter, and his enthusiastic assistance has been a constant source of motivation.

I extend my thanks to Prof. Dr.-Ing. Oliver Prenzelat Hochschule Bremerhaven for his mentorship and guidance throughout the course of my thesis. His expertise and support have been crucial to my academic journey.

Engaging in programming activities during work has significantly enhanced my understanding of how embedded systems works. These hands-on experiences have played a pivotal role in deepening my knowledge in this domain.

Lastly, I wish to convey my appreciation to my friends at Hochschule Bremerhaven for their consistent encouragement and motivation.

I am truly grateful to everyone who has been a part of my academic and personal journey.

Thank you everyone...

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Full Form
APA	Arbitrary Processor Affinity
BF	Best Fit
CPU	Central Processing Unit
DMS	Deadline Monotonic Scheduling
DLR-OSS	DLR-On Board Software System
EDF	Earliest Deadline First
FF	First Fit
FPP	Fixed Priority Preemptive
gAPA	Global First Arbitrary Processor Affinity
ILP	Integer Linear Programming
LP	Linear Programming
MIAT	Minimum Inter Arrival Time
NF	Next Fit
OBC	On-Board Computer
pAPA	Partitioned First Arbitrary Processor Affinity
ReFEx	Reusability Flight Experiment
RTOS	Real Time Operating System
RTEMS	Real-Time Executive for Multiprocessor Systems
RTA	Response Time Analysis
SMP	Symmetric Multiprocessing
WCRT	Worst Case Response Time
WF	Worst Fit
WRR	Weighted Round-Robin

Chapter 1

Introduction

As the use of multi-core systems spread across all domains, efficiently using the promise of hardware parallelism for real-time applications has arisen as a critical concern. The optimization of task assignment algorithms for multi-core real-time systems is a critical issue in this regard. Over the last decade, significant efforts have been dedicated to addressing this issue, leading to the proposal of diverse scheduling algorithms.

The performance improvements with a multi-core system are dependent on both the nature of the application and its software implementation. It's important to use the right methods for dividing software into tasks (threads) and fairly distributing these tasks across processors to maximize overall performance in order to take use of the explicit parallelism given by multi-core architectures. Real-time systems are frequently multi-threaded, which makes them more adaptable to multi-core platforms than sequential programs that are only one threaded. Simultaneous execution can be used to improve performance when tasks are independent. Many legacy real-time systems that are now in use are large and complex, frequently including millions of lines of code that have been years in the making and maintenance. The option of getting rid of these legacy systems and starting a new one from scratch is often not practical due to the significant expenditures made in them.

In the modern technological world, there are two fundamental methods used for scheduling real-time tasks on multi-core systems, which are 1) Partitioned approach, 2) Global approach (Detail discussion in Chapter 2). In the partitioned approach, each task is statically allocated to a single core, and migration between cores is prohibited. Conversely, the global approach allows tasks to migrate freely and be executed on any available core. However, when dealing with extensive multi-core platforms, both these approaches encounter limitations that limit the levels of core utilization that can be achieved.

Additionally, researchers have also extensively investigated hybrid techniques. Clustered scheduling is a well-known hybrid strategy in which processors are divided into disjoint clusters, each task is statically allocated to a particular cluster, and a "global" scheduling policy is carried out inside each cluster. The multiprocessor platform is divided into clusters, with each cluster containing cores. Tasks are assigned to clusters in a static manner, resembling partitioning, while being subjected to global scheduling within each individual cluster. This method provides a good solution for addressing the limitations of traditional partitioned and global scheduling strategies.

Since, real-time operating systems for multi-core computers differ from the conventional scheduler solutions that have been discussed in the literature [27]. Instead, they acquire a more adaptable migration method based on the concept of "processor affinity". Processor affinity allows tasks or threads to be bound to certain subsets of available processors, limiting their execution on unaffiliated ones [38]. With the help of this dynamic approach, heterogeneous platforms with various core types can be optimized. The application performance in throughput-oriented computing is enhanced, real-time and non-real-time tasks are split up onto different cores, and the approach can be tailored to meet the needs of particular tasks (like those that must be completed quickly and with little consideration for cache) [44].

Moreover, It makes it easier to implement different scheduling concepts like global, partitioned, and clustered scheduling, with the added benefit of allowing Arbitrary Processor Affinity (APA) to be assigned on a task-by-task basis. This provides greater flexibility and versatility in migration strategies when compared to traditional scheduling methodologies.

In 2013, Brandenburg released a paper outlining the benefits of APA scheduling, demonstrating how it provides better schedulability and dominates global, partitioned, and clustered scheduling [27]. The Linux push and pull scheduler was examined by the authors [27]. Although more prominent two schedulers based on APA were added to Real-Time Executive for Multiprocessor Systems (RTEMS) in 2015. Nevertheless, the 2020 version of the RTEMS space profile does not have this functionality [45]. The focus of this thesis is an in-depth analysis of the partitioned first APA technique originally presented by the DLR-On Board Software System (DLR-OSS) group. This strategy proposed for assigning tasks using a partitioned first APA approach, which aims to improve task assignments by utilizing various bin packing methods. Furthermore, the thesis attempts to examine the impacts of pAPA technique when combined with other priority assignment methods and to give better understanding of the combined effect of bin packing algorithms, and priority assignment techniques on partitioned first APA and its impact on system scheduling and performance optimization.

This thesis explores the topic of optimised task allocation to enhance multi-core scheduling in real-time systems using Integer Linear Programming (ILP) for better task assignments for APA scheduling. The key focus is on developing a better method of task assignment using ILP, with a particular emphasis on core affinities using divide and conquer strategies. This will be explained more in details in the following chapters. This method is to provide a complete structure that optimizes their assignment based on core affinities, hence improving the overall performance and responsiveness of real-time applications operating on multi-core architectures.

1.1 Use Case Scenario

In the world of multi-core platforms, task scheduling efficiency is critical. This study explores several options for reducing over-provisioning and optimizing task allocation within this computational framework.

The German Aerospace Center's project *Reuseability Flight Experiment (ReFEx)* aims to perform an autonomous re-entry into earth's atmosphere from hypersonic speeds in high altitudes down to subsonic speeds in low altitudes [12].

In this project, the necessary state estimation and control algorithms needed to fuse the information of multitudes of sensors and derive control signals for all actuators will become complex and computationally demanding. Therefore, a multi-core platform has been selected for the on-board computers to handle these control tasks with high real-time accuracy, while collecting high frequency science data and processing general command and data handling tasks at the same time [47].

Figure 1.1 depicts the ReFEx mission sequence, including basic information such as, time and altitude. Based on the basic requirements, Real-Time Executive for Multiprocessor Systems (RTEMS) was selected as operating system [12]. RTEMS, an open-source real-time operating system, is compatible with a wide range of

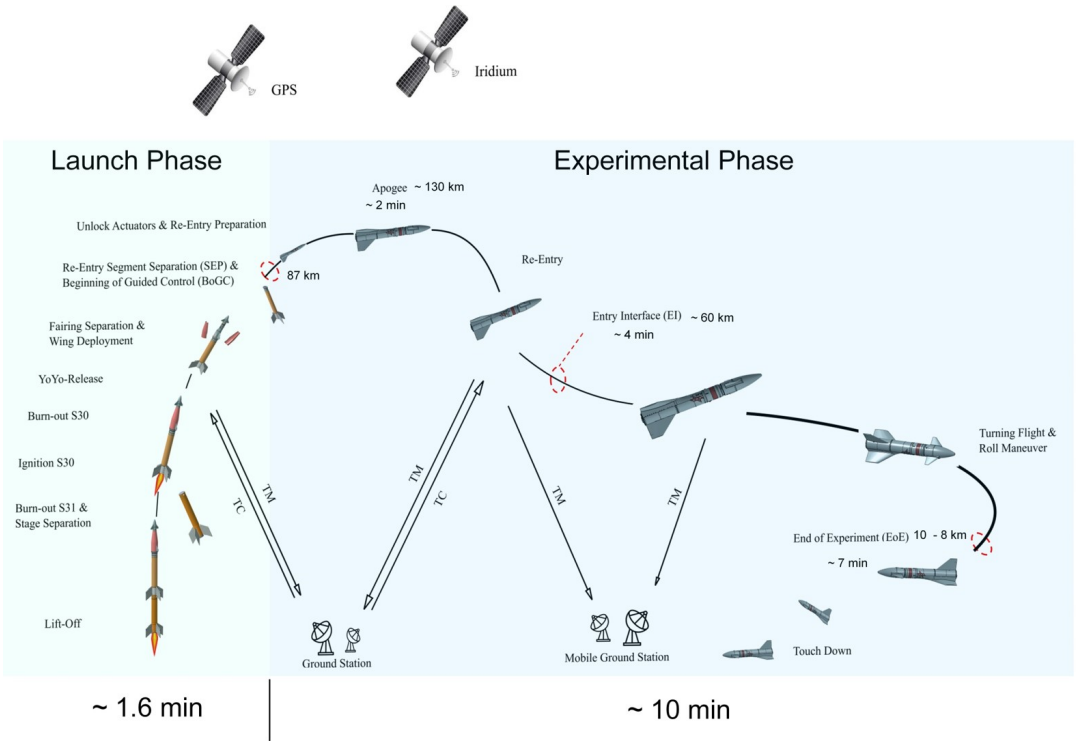


FIGURE 1.1: Overview of the ReFEx mission sequence [12].

hardware architectures and has a long track record in space applications. In its most recent version, RTEMS includes a real-time executive and real-time schedulers for Symmetric Multiprocessing (SMP) on multi-core systems. In particular, the programming interface allows numerous scheduling groups and thread pinning, allowing for the strategic allocation of tasks among processor cores while following to certain limitations [3]. The RTEMS version for x86-based hardware platforms, which has a 4-core multi-core system, requires that tasks be assigned to cores in a way that maximizes schedulable tasks. RTEMS port for x86-based hardware platform has been extended to support SMP for the x86- based On-Board Computer (OBC) of ReFEx. However, this allocation method uses a global scheduling technique for task scheduling, which may not be the most efficient option.

This work draws inspiration from the ReFEx project, with the goal of providing a less conservative or pessimistic solution. The objective is to improve schedulability and decrease over-provisioning. In order to achieve this goal, APA were used as a strategy.

1.2 Chapter Overview

This master thesis is divided into eight different chapters. This section of the introduction offers a quick summary of the following chapters:

Chapter 2: Chapter 2 describes the system model and problem statement.

In this chapter, system model is comprehensively defined. Additionally, this chapter provides a real-time task structure with several sub-tasks that are assigned and executed. Furthermore, the specific problem that the research intends to address

is clearly stated, highlighting the gap in existing solutions and and motivating the need for the proposed approach.

Chapter 3: Chapter 3 explains the prior work and background.

This chapter presents a critical examination of the literature on real-time systems, scheduling approaches, multiprocessor architectures, and multi-core technologies. It also outlines the expected consequences of the full background understanding, highlighting important concepts and insights necessary for understanding the next discussions. It explains the types of schedulability analysis for single-core and multi-core real-time systems.

Chapter 4: This chapter explains task allocation methods.

Building upon the groundwork established in Chapter 3, this section delves deeper into the complexities of schedulability analysis within real-time systems and show types of task allocation methods. It offers an exploration of the existing solutions and techniques for assessing the feasibility of task schedules, identifying their strengths and limitations. Also, it is concentrating on the use of the Partitioned First Arbitrary Processor Affinity (pAPA) method and Global First Arbitrary Processor Affinity (gAPA). This approach's theoretical basis and practical implementation are discussed, demonstrating its potential to improve real-time task scheduling.

Chapter 5: Chapter 5 describes the optimized affinity sets for APA approach.

The heart of the thesis lies in this chapter, where the proposed approach is presented. The concept of optimized affinity sets is introduced, with a primary emphasis on employing the ILP strategy. This chapter provides a full examination of the theoretical foundation and actual implementation of this technique, clarifying its power to significantly improve real-time task scheduling with detailed explained it's mathematical formulations.

Chapter 6: Following that, chapter 6 highlights the development of experimental investigation.

In this empirical phase of the research, Chapter 6 undertakes a comprehensive examination of the proposed approach. Diverse scenarios are considered, varying factors such as the number of cores, tasks, utilization levels, priorities, and task periods. Through carefully experimentation, the chapter offers a thorough evaluation of the approach's effectiveness and performance.

Chapter 7 and 8: Conclusion and future directions.

The concluding chapter draws together the findings from the previous chapters. It presents a brief summary of the results achieved through the proposed approach and experimentation. Additionally, the chapter reflects on the broader effects of the research, suggesting potential directions for future investigations in the areas of optimal task allocation and real-time task scheduling on multi-core systems.

Chapter 2

System Model & Problem Statement

This chapter introduces the system model and formally outlines the problem addressed in its main contribution. The chapter includes formal definitions of the key concepts on which the thesis is based, such as task assignments, task scheduling, and minimum inter arrival time.

2.1 System Model

This section will introduce the basic modeling framework required for this thesis. Table 2.1 (page no. 7) contains on an extensive list of notations that are required to understand the next topics.

Key Components for Schedulability Analysis:

Schedulability analysis involves several key components which are followings:

a. Task Arrival Times: The timing at which tasks task becomes available for execution in the system. It is the time when a task is submitted to the scheduler and is ready for execution on one of the processor cores.

b. Task Execution Times: The duration of time it takes for a task to complete its execution after it begins executing on a CPU core. This statistic is critical for predicting overall performance in a multicore system. This is illustrated in Figure 2.1.

c. Task Preemption: Task preemption in scheduling is the ability to temporarily pause a lower-priority task's execution, allowing a higher-priority task to run. Preemption is critical for supporting tasks with varying degrees of priority and maintaining balanced resource allocation in a multicore system.

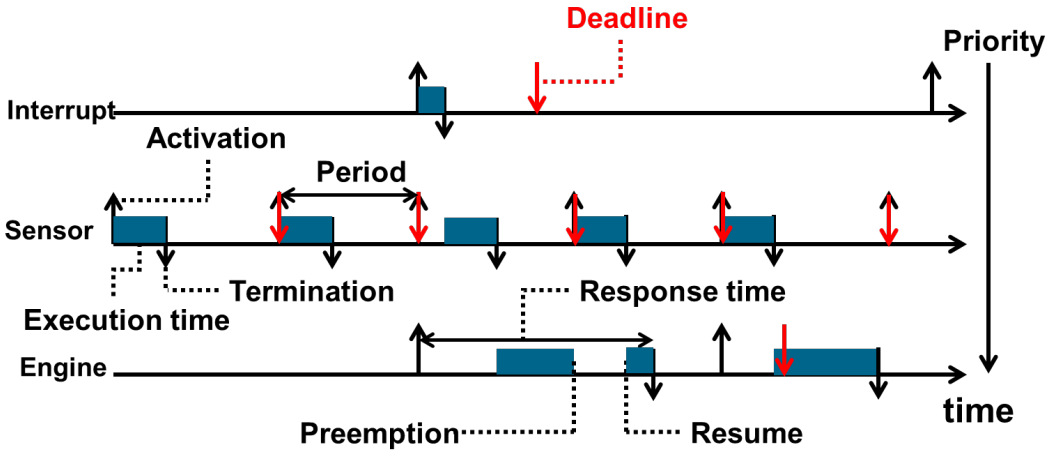


FIGURE 2.1: Basics of schedulability analysis.

Definition 2.1 (Task): A task is defined as a tuple $\tau_i = (prio_i, T_i, D_i, C_i, \alpha_i)$ having tasks with task's priority ($prio_i$), a minimum inter-arrival time (MIAT) or period (T_i),

a relative deadline (D_i), a worst-case execution time (WCET) (C_i), and task's core affinities (α_i).

Definition 2.2 (Periodic task): A periodic task is a type of task in real-time systems that occurs at regular intervals and follows a predictable pattern or schedule. Mathematically, a periodic task is characterized by having a fixed inter-arrival time between successive occurrences of the task. This work considers the *periodic* task model.

Definition 2.3 (Sporadic task): A task is classified as sporadic when it is triggered by events that happen infrequently and at unpredictable intervals. Mathematically, this unpredictability is denoted by signifies the smallest time gap between successive occurrences of these events [28].

Definition 2.4 (MIAT or Period): It represents the time interval or duration between the successive run of execution or activation of those tasks. If the activation and execution times of two tasks denote as t_1 and t_2, respectively, then the period (T) between these tasks can be defined as:

$$T = t_2 - t_1$$

Definition 2.5 (Relative deadline): The relative deadline, denoted as D_i, for a task τ_i specifies a time window in which the task must be completed.

The relative deadline can be implicit, constrained, or arbitrary. The deadline is implicit when it is equal to the period of the task (T_i) ($D_i = T_i$).

When $D_i \leq T_i$, then the deadline is constrained. However, if the specific task deadlines might be shorter, equal to, or longer than their duration $D_i \geq T_i$, then the deadline is arbitrary [20].

In this work, implicit deadlines are considered.

Definition 2.6 (Worst Case Execution Time (WCET)): The WCET refers to the longest duration that a task requires to complete its execution on a particular hardware platform [32].

Definition 2.7 (Core affinity): Each task τ_i also has a core affinity α_i, where α_i is the set of cores affinities that τ_i may be scheduled on. The joint core affinity of a taskset as the set of cores on which at least one task in the taskset can be scheduled. Similarly, a taskset is defined as the set of tasks that may be scheduled on at least a single affinity among a set of affinities.

Core affinity is the capability that allows the assignment of a process or multiple processes to a particular CPU cores, ensuring that these processes exclusively run on that specified cores. This strategy enhances CPU utilization by effectively utilizing the available cores for concurrent processing [2].

The determination of core affinities involves a pivotal formula that establishes the number of available affinities, denoted as Number of affinities. This calculation is mathematically expressed as number of affinities $2^M - 1$ where M signifies the number of processor cores within the system.

Definition 2.8 (Utilization): The task's utilization is:

$$u_i = \frac{C_i}{T_i} \tag{2.1}$$

In this thesis, the challenge of assigning a set of n independent periodic real-time tasks $\tau_1, \tau_2, \ldots \tau_n$ is considered which are running on a set of M identical cores $\pi_1, \pi_2, \ldots \pi_M$ where tasks are assigned with fixed priorities where the priorities of the tasks doesn't change during run-time.

Definition 2.9 (Interfering taskset): A task τ_k can (directly) interfere with another task τ_i, i.e., delay τ_i's execution, only if α_k overlaps with α_i. In general, the exact collection of conflicting tasks is determined by the scheduling policies. Therefore, the interfering taskset is defined if τ_i is scheduled under any scheduling algorithm. For example, in an fixed priority scheduler, only higher-priority tasks can interfere with τ_i. Considering $prio_k$ to be τ_k's fixed priority, where $prio_k > prio_i$ indicates that τ_k has a greater priority than τ_i (i.e., τ_k can preempt τ_i).

The interfering taskset I_i is defined as the set of tasks that can potentially interfere with T_i when planned using any scheduling method. For example, in an fixed priority scheduler, only tasks with higher priorities can be included in the interfering taskset [27].

$$I_i = \{T_k \mid prio_k > prio_i \wedge \alpha_k \cap \alpha_i \neq \emptyset\} \text{ [27]} \tag{2.2}$$

Symbol	Description
$\boldsymbol{M}$	Set of identical cores
$\boldsymbol{\pi}$	Number of cores
$\boldsymbol{\tau}$	Taskset
$\boldsymbol{\tau_k}$	K$^{\text{th}}$ task in $\boldsymbol{\tau}$
$\boldsymbol{T_i}$	Period or minimum inter-arrival time of $\boldsymbol{\tau_i}$
$\boldsymbol{C_i}$	Worst-case execution time of $\boldsymbol{\tau_i}$
$\boldsymbol{D_i}$	Arbitrary deadline of $\boldsymbol{\tau_i}$
$\boldsymbol{\alpha_i}$	Processor affinities
$\boldsymbol{u_i}$	Utilization of $\boldsymbol{\tau_i}$, $\frac{C_i}{T_i}$
$\boldsymbol{U}$	Total utilization
$\boldsymbol{I_i}$	Interference on $\boldsymbol{\tau_i}$

TABLE 2.1: Overviews of system model.

2.2 Problem Statement

This section describes the problem scenery and outlines the thesis objectives for improving task allocation in a multi-core real-time scheduling context. The thesis aims to optimize task allocation within a multi-core real-time scheduling, focusing

specifically on the task set τ, comprised solely of periodic tasks denoted as $\tau_i = (prio_i, T_i, D_i, C_i, \alpha_i)$.

The goal is to maximize resource utilization, adhere strictly to timing constraints, and ultimately enhance the performance and reliability of real-time systems.

- **Maximizing Resource Utilization:** Using available computational resources on several cores as efficiently as possible.
- **Strict Adherence to Timing Constraints:** Consistently ensuring that tasks are completed within their specified time frames.
- **Enhancing Performance and Reliability:** Improving the overall performance and dependability of real-time systems.

Within the context of a task $\tau_i \in \tau$, This thesis focuses on the challenge of effectively assigning a growing number of tasks without missing deadlines within the system. Because of the fundamental complexity of job assignment in real-time systems, it is a difficult challenge. In order to resolve these problems, the thesis will conduct a detailed analysis of the fundamental approaches which are current in use for task assignment. Understanding the details of task allocation in order to provide creative ways that meet the expanding demands on multi-core systems while maintaining the integrity of real-time constraints is important.

In summary, following studies will look at the difficulties of task scheduling in order to provide solutions and insights that improve the performance and dependability of real-time systems.

Our contributions to this report include the following:

- The implementation of an efficient task allocation technique aimed at reducing task blocking time and enhancing task schedulability within the system architecture.
- Presenting a complete comparison of outcomes with gAPA, pAPA, partitioned, and global scheduling algorithms.
- Comparative evaluation of execution times across different approaches to examine the scalability of approaches.

2.3 Related Work

Scheduling more tasks to the embedded system can reduce the over-provisioning which reduces the cost and the power consumption. The most efficient utilization of a processor is to reach 100% by using an optimal scheduler. In single-core platforms that can be achieved by the earliest deadline first scheduling algorithm (EDF) [22]. EDF suffers from high number of preemption, i.e., context switching, [17]. Another direction of improving the schedulability is to exploit tolerable deadline misses. These tasks are called weakly-hard real-time tasks [13]. In this direction, A Linear Programming (LP) based weakly-hard schedulability analysis has been presented in [49] for overloaded systems. It is extendable for more scheduling policies.

Fixed Priority Preemptive (FPP) and non-preemptive are covered in [49], Weighted Round-Robin (WRR) in [29], and EDF in [30].

In multi-core platforms, the challenge is harder. Similar to the single-core, there were efforts to propose optimal scheduling algorithms. Some global scheduling approaches are optimal such as the pfair algorithms [4, 11] and U-EDF [40]. The optimal global scheduling approaches have the same problem that the EDF has for single-core platform, namely the high scheduling overhead. there is no RTOS implement any of them. Since the partitioning problem is NP-hard, i.e., no optimal partitioning algorithm exists, plenty algorithms have been proposed, e.g., [24, 31, 34, 35]. Most of the proposed partitioning algorithms depend on the standard bin-packing heuristics of first-fit, worst-fit and best-fit [10]. APA scheduling is a promising approach to improve the schedulability on the multi-core platforms. As mentioned earlier in this chapter, this thesis aims to propose an allocation approach for APA scheduling to improve the schedulability.

Chapter 3

Background on Schedulability Analysis

As noted in the introduction, APA scheduling works on the idea of processor affinities to provide a flexible migration approach [27]. As a result, we begin by categorizing real-time scheduling algorithms based on various migration techniques which are already proposed, then compare them to the APA scheduling. Subsequently, it will describe which task assignment strategies have been applied in the proposed scheduling algorithms.

3.1 Scheduling Approaches for Multi-core Systems

Real-time scheduling techniques can allow unrestricted migrations, no migrations, or a hybrid approach with intermediate migrations. In order to effectively manage the distribution of tasks or processes among various cores in a multi-core system, multi-core scheduling is a crucial component of contemporary operating systems. There are primarily two ways for scheduling real-time systems on multi-core [9], [19] which are the following.

- Global scheduling
- Partitioned scheduling

These approaches serve different purposes and are applied based on the characteristics and requirements of the system. The choice between global and partitioned scheduling depends on factors such as the system architecture, timing constraints and resource availability.

3.1.1 Global Scheduling Approach

The Global scheduling approach employs a single scheduler to manage tasks, enabling each task to execute on any core [41]. As depicted in Figure 3.1, it is a centralized strategy in which a single scheduler is in charge of distributing tasks to all of the system's available processing cores. The scheduler makes choices depending on the workload and priorities of the entire system. A task can be preempted on a core and resumed on another core, i.e., migration of tasks among cores is permitted.

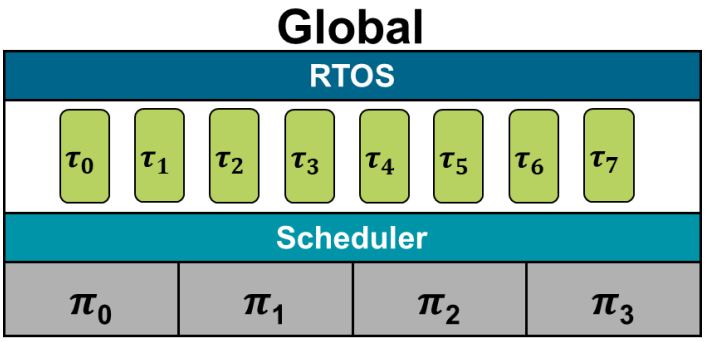


FIGURE 3.1: Global scheduling, grey color box represent as a cores (π), green box's are represent as a tasks (τ) and the scheduler specified in between tasks and cores.

In global scheduling, a central scheduler manages the task assignment across all cores in a multi-core system. This means that the scheduler can reallocate tasks to cores based on real-time system conditions, such as changes in workload or resource availability. This dynamic assignment approach enables global schedulers to adapt to changing system dynamics and maintain optimal performance.

3.1.2 Partitioned Scheduling Approach

Under partitioned scheduling tasks are statically assigned to the cores, and the tasks within each core are scheduled by a single-processor scheduling algorithm [41]. Each task is assigned to a core where it will run. Each core has its own ready queue for scheduling tasks (As shown in Figure 3.2).

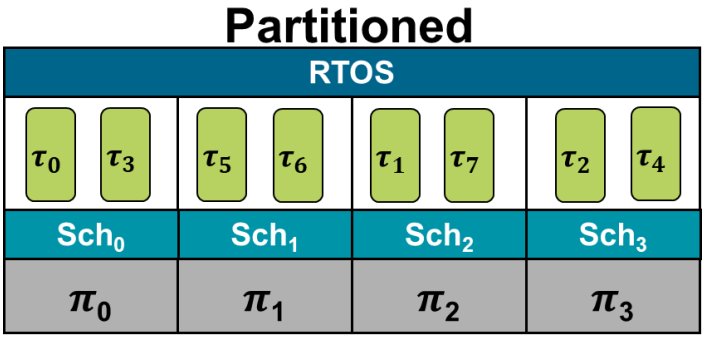


FIGURE 3.2: Partitioned scheduling approach.

Task assignment is static, which is one of the key characteristics of partitioned scheduling. This means that the task assignment remains fixed throughout the execution of the application. To assign tasks to cores statically, partitioned scheduling often employs bin packing algorithms. In the context of partitioned scheduling, bin packing algorithms are used to efficiently allocate tasks to cores, minimizing the number of cores required or balancing the load on the available cores [42] which is explained in detail in Chapter 4, specifically in Sub-section 4.1.1.

Depending on the number of migrations permitted, researchers have also explored hybrid techniques in details that has been proposed with an intermediate degree

of migration include the clustered scheduling [18], semi-partitioned scheduling [5], and restricted-migration scheduling [5].

3.1.3 Cluster Scheduling Approach

Cluster scheduling, another approach for scheduling can be defined as a generalization of partitioned and global scheduling methods. It is a middle-of-the-road technique that divides cores into clusters. The system can contain more than one cluster, and each cluster has its own scheduler. Tasks are statically assigned to clusters and tasks within each cluster are globally scheduled [48] (As shown in Figure 3.3).

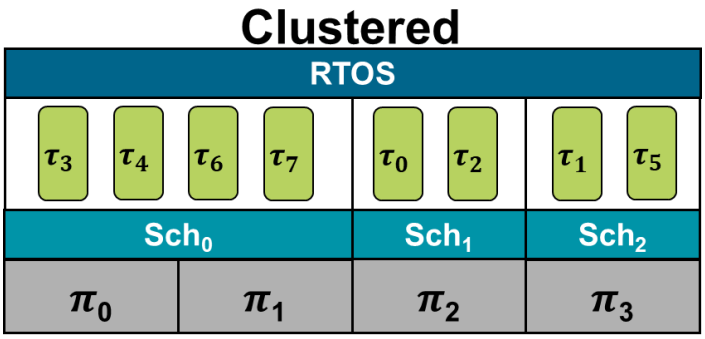


FIGURE 3.3: Clustered scheduling approach.

Global scheduling is appropriate for systems that require optimal load balancing and global priority management yet can handle the additional complexity. Partitioned scheduling is suited for applications that require strict isolation, whereas cluster scheduling provides a balance between isolation and adaptability. Unlike the conventional descriptions found in literature, many contemporary multiprocessor real-time operating systems, such as VxWorks, LynxOS, QNX, and real-time variants of Linux, implement scheduling algorithms that are not strictly based on the traditional methods [27]. Instead, they rely on processor affinity to provide a more flexible migration technique. Processor affinities improve application performance in throughput-oriented computing and separate real-time and non-real-time tasks by allocating them to independent cores [25, 33, 39, 46].

3.1.4 Arbitrary Processor Affinities (APA) Scheduling

APA, in the context of multi-core systems, refers to the ability to explicitly assign a specific task, or process to execute on a particular processor core [27]. This affinity setting allows the programmer or system administrator to dictate where a particular task or workload should be executed, rather than relying on the system's default scheduler.

APA refers to the ability to manually establish or alter the assignment of certain tasks to individual cores. In APA scheduling, developers take command and direct where each process executes. This can be effective in some cases when the user is familiar with the workload and the system's design.

Global, partitioned, and clustered scheduling are all generalized by APA scheduling. In other words, APA scheduling forces each task to migrate only among a limited

group of cores determined by the task's processor affinity [27]. For this reason, a taskset can be described as a global, clustered, or partitioned taskset using the proper processor affinity assignment.

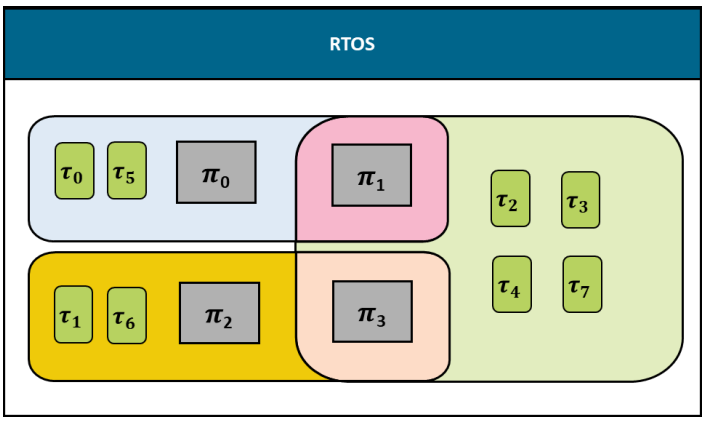


FIGURE 3.4: APA, illustrated that tasks τ_0, τ_5 are allocated to π_0 and π_1 cores affinity, whereas core affinity π_1 and π_3 also has different tasks which are $\tau_2, \tau_3, \tau_4, \tau_7$ shows that π_1 is common for some affinities.

The Figure 3.4 illustrate the concept of APA scheduling, a technique that allows for the explicit assignment of tasks to specific cores or clusters based on their processor affinity. APA scheduling is a powerful tool for optimizing performance in multi-core systems by controlling the allocation of tasks on specific cores or clusters.

Here, the execution plan based on the processor affinity assignments.

- Seven tasks are denoted as τ_0, τ_1, τ_2, τ_3, τ_4, τ_5,τ_6 and τ_7. Each task represents a unit of taskset in a multi-core environment.
- τ_0, τ_5 tasks are assigned to the processor affinity set consisting of cores π_0 and π_1.
- tasks τ_2, τ_3, τ_4, and τ_7 will execute on cores π_1 and π_3, forming a dedicated cluster for these tasks.
- Tasks τ_1 and τ_6 will execute on cores π_2 and π_3, creating another dedicated cluster.

This example demonstrates how APA scheduling allows tasks to be explicitly assigned to processor affinity sets, leading to fine-tuned control over task assignments and the potential for improved performance in multi-core systems.

While APA scheduling offers performance benefits, it introduces complexity and management overhead. Careful planning to define processor affinities is required to ensure that core assignments are optimal for the workload. Additionally, tasks assigned to the same cluster may need to communicate efficiently with each other.

Understanding APA scheduling's complexities help to see that, despite it has many performance advantages, this strategy adds a layer of complexity and requires careful planning for the best tasks assignment. When global, partitioned, and

clustered scheduling fail, APA scheduling can make a taskset feasible by carefully selecting processor affinities. In 2013, Brandenburg's work showed that APA scheduling problems can be efficiently reduced to "global-like" sub-problems, which allows us to reuse the large body of literature on global schedulability analysis. This makes APA scheduling a powerful and flexible scheduling algorithm that can be used to schedule real-time tasks on multi-core systems [27] (Detail explanation in Sub-section 3.2.3).

3.2 Schedulability Analysis

The importance of schedulability analysis

Schedulability analysis is the process of determining whether a set of real-time tasks can be scheduled to meet their deadlines. This analysis is essential to guarantee that all tasks, each with its own execution time and deadline, can be accommodated without violating timing constraints in the worst-case scenario, which cannot be covered by testing. There are two main categories of schedulability tests upon the processor:

- schedulability analyses for single-core systems, which include partitioned and cluster scheduling.
- schedulability analyses for multi-core systems and that covers global, cluster and APA scheduling.

The scheduling of periodic tasks with hard deadlines on a single-core, focusing on the rate monotonic scheduling method developed by Liu and Layland in 1973 [37]. Extending the rate monotonic scheduling analysis, in 1990 J. P. Lehoczky provide worst-case bounds that generalize the original Liu and Layland bounds, allowing flexibility in task deadlines [36]. The author suggest a practical approach for distributed tasks into resource sequences and end-to-end deadlines, using traditional rate monotonic theory which is known as busy window analysis. In this experiment, busy-window analysis to determine the schedulability of a single-core system is used, which includes partitioned and cluster scheduling evaluations.

3.2.1 Busy-window Analysis

Busy window analysis is a key component of schedulability analysis, primarily applied in single-core real-time systems [36], since it helps determine whether tasks can be scheduled and helps make sure deadlines are fulfilled. It involves the examination of the core's execution timeline to identify intervals, known as "busy windows," during which the core is fully occupied with executing tasks. These busy windows are critical because they represent periods during which new tasks must wait to be scheduled.

Understanding the worst-case scenario for task scheduling requires a knowledge of busy windows. During a busy window, the core is already committed to executing operations, making it difficult to fit other tasks into the remaining time. If the execution time and deadline of a task occur inside a busy window, it may not be schedulable, resulting in missed deadlines and system failure.

Consider a collection of n periodic tasks $\tau_1, \tau_2, \tau_3 \ldots, \tau_n$, each of which has four critical characteristics (C_i,T_i,D_i,I_i):

- C_i: Execution of task τ_i
- T_i: Period of τ_i
- D_i: Deadline of τ_i
- I_i: τ_i interface with respect to a fixed time origin

Fixed priority scheduling algorithms are frequently used in real-time systems to enforce task deadlines. The main goal is to find a scheduling algorithm that can ensure all task deadlines are met. To determine if a scheduling algorithm can meet all deadlines, it's important to identify the task scheduling that results in the longest response time for every assignment of a certain task τ_i.

To calculate the worst-case response time for a task with priority level i, the proposed idea involves the concept of a $level - i$ busy period. The time frame in which all tasks with priority levels less than or equal to i are being executed is known as a $level - i$ busy period. The longest $level - i$ busy period is equivalent to the worst-case response time for a task with priority level i [36].

To perform a busy window analysis, the busy windows must be calculated or estimated by taking into account the task set's arrival times, execution times, and deadlines. Various scheduling methods and strategies are used to correctly compute these busy window. Equation 3.1 is often used to calculate busy window for task in scheduling algorithms to manage resources or tasks within a system.

$$\mathrm{BW}_i^{(n+1)} = C_i + \sum_{j<i} \left\lceil \frac{\mathrm{BW}_i^n}{T_j} \right\rceil C_j, \quad \mathrm{BW}_i^0 = C_i \text{ [36]} \tag{3.1}$$

where:

$$\begin{aligned} \mathrm{C}_i &= \text{ Execution time of task } \tau_i \\ \mathrm{C}_j &= \text{ Execution time of task } \tau_j \\ \mathrm{BW}_i^{(n+1)} &= \text{ Busy window of task i in the (n+1)}^{\text{th}} \text{ iteration} \\ \mathrm{T}_j &= \text{ Period of task } \tau_j \end{aligned}$$

In practical terms, this equation facilitates the determination of the busy window for a specific task, such as the 4^{th} task, by considering the busy windows of all preceding tasks. The initial condition, $BW_i^0 = C_i$, establishes that the busy window of a task in the initial iteration is equivalent to its execution time.

Busy window analysis, a crucial component in schedulability analysis algorithms within computational systems, involves several key elements. The term BW_i^{n+1} defines the busy window at time $n + 1$, provides insight into the predicted resource utilization within the system. C_i symbolizes the computational time required for task i, a pivotal factor in assessing the workload of individual tasks. The expression

$\sum_{j<i} \lceil \frac{BW_i^n}{T_j} \rceil C_j$ carefully calculates the overall load exerted by tasks arriving before task i.

By studying task characteristics and scheduling methods, the conditions under which all task deadlines can be fulfilled may be identified. The study gives useful insights for schedulability analysis of the tasks in single-core real-time systems. The transition from single-core to multi-core systems introduces the concept of global schedulers, which maintain a single scheduling queue for dynamic task execution across available cores. This centralized decision-making technique at the system level is useful for efficiently handling dynamic loads and temporary overload scenarios. Efficient schedulability tests for the partitioned case, which use above proven single-core approaches, presently outperform those for the global case [14].

Bertogna and Cirinei developed a unique technique in 2007 to analyze the timely features of real-time systems planned on identical multi-core platforms, coinciding with an increase in interest in scheduling analysis. Their research tackles the limits of previous findings and eliminates important shortcomings by improving methodologies typically used in single-core schedulability analysis. Specifically, they apply the Response Time Analysis (RTA) to systems with many cores, proposing schedulability tests that outperform all existing approaches. This establishing methodology bridges the gap between single-core and multi-core schedulability assessments, providing improved methodologies for evaluating the real-time performance of systems running on identical multiprocessor platforms.

3.2.2 Response Time Analysis (RTA) for Global Scheduling

RTA for real-time applications usually rely on fixed-point iteration algorithms to determine upper limits on task response times [8, 14, 26, 43]. Busy window Analysis primarily focuses on predicting task allocation based on computational requirements, RTA extends this framework to ascertain the responsiveness and predictability of tasks in a global scheduling context. The RTA will be applied to multi-core systems, allowing for schedulability tests that significantly outperform all previously approached techniques.

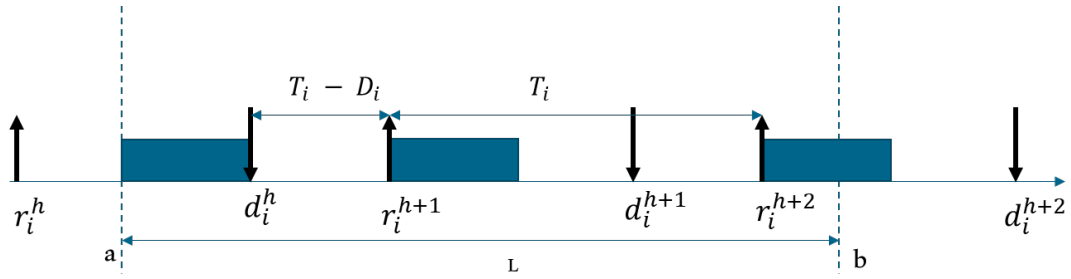


FIGURE 3.5: Scenario described for Equation 3.2 [14].

The maximum response time (r_i) of a task (τ_i) is characterized as the longest duration taken by any job of task τ_i to complete its execution [27]. The analysis conducted by Bertogna and Cirinei in 2007 establishes upper bounds on this response time, relying on the principles of workload and interference [14]. Task τ_i workload ($\widehat{W}_i(L)$) is defined as the maximum duration during which the task can execute within any interval of length L (As shown in Figure 3.5).

To compute the workload of task τ_i in an interval $[a, b)$ of length L, the first job of τ_i after the task's arrival time, is released at the time $a + C_i + T_i - D_i$. The next jobs are then released periodically every T_i time unit [14]. Therefore, the workload is determined by the number of tasks $N_i(L)$, that contribute to a complete WCET in any period of length L, which is given by

$$N_i(L) = \left\lfloor \frac{L + D_i - C_i}{T_i} \right\rfloor \text{ [14]} \tag{3.2}$$

where:

$\mathrm{N}_i(L) =$ Number of job instances of task j
$\mathrm{C}_i =$ Computational or processing time required by task j
$\mathrm{L} =$ Time frame
$\mathrm{T}_i =$ Task's period

The contribution of the terminated task can then be bounded by $\min(C_i, L + D_i - C_i - N_i(L)T_i)$. Equation 3.2 provides an estimation of the number of job instances of task j within the specified time frame L considering the timing constraints and periodic behavior of the task.

The workload of a task τ_i is determined based on $N_i(L)$ as follows:

$$\widehat{W}_i(L) = N_i(L) \cdot C_i + \min\left(C_i, (L + D_i - C_i - N_i(L)T_i)\right) \text{ [14]} \tag{3.3}$$

The equation 3.3 for $\widehat{W}_i(L)$ provides an estimation of the busy window for task i up to time L. It incorporates the number of job instances ($N_i(L)$) that fit within the time frame L, multiplied by the computational time of task i. Additionally, it considers the interference caused by higher-priority tasks, including their computational times and the remaining time in the time frame L after considering the response time and execution of task i.

In the context of task scheduling, the interference ($I_i(t)$) imposed by a higher-priority task (τ_j) on the lower-priority task (τ_i) is defined as the overall duration of sub-intervals during which τ_i is backlogged but unable to be scheduled on any core while τ_j is in execution. This interference is intricately linked to the workload of the higher-priority task, indicating how τ_i's execution affects the timely execution of τ_j within intervals of length L. Therefore, the expression of worst-case interference which is depends on the workload of interfering task given by,

$$\mathrm{WCRT}_j = C_j + \left\lfloor \frac{1}{m} \sum_{i<j} \min\left(\widehat{W}_i(\mathrm{WCRT}_j), \mathrm{WCRT}_j - C_j + 1\right) \right\rfloor \tag{3.4}$$

where:

$\widehat{W}_i(\mathrm{WCRT}_j) =$ Busy window or workload ($\widehat{W}_i$) of task j

The relationship between WCRT_i and $\widehat{W}_i(L)$ can be observed in how the interference

caused by higher-priority tasks (as seen in Equation 3.4) affects the estimation of the busy window $\widehat{W}_i(L)$ for task i within a given time frame L.

3.2.3 Schedulability Analysis for APA Scheduling

According to the paper [27], the RTA for APA scheduling is a simple extension of the reduction-based schedulability analysis which is explained here with one example. When a set of tasks assigned to the certain core affinity is given, and those tasks can globally schedulable on that affinity or on a subset of that affinity, then it can also be schedulable using APA scheduling.

As shown in Figure 3.6, consider a scenario with four cores $\pi_0, \pi_1, \pi_2, \pi_3$ and five tasks $\tau_0, \tau_1, \ldots, \tau_4$. The APA scheduling technique assigns tasks to affinities, resulting in improving efficiency. In this scenario, task τ_0 has the highest priority, while task τ_4 has the lowest priority. The affinity assignment is such that some tasks can easily be scheduled on their preferred cores.

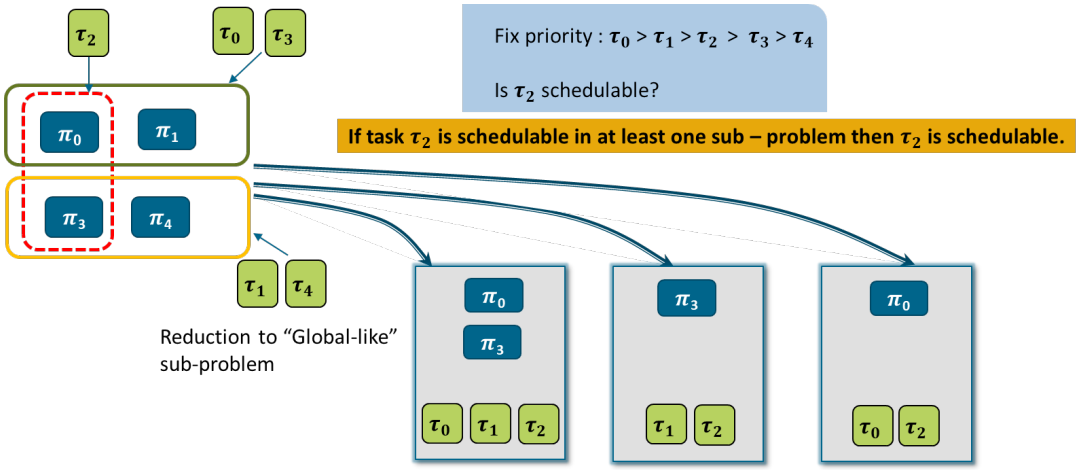


FIGURE 3.6: Elaboration of gAPA approach, where can see how task are assign to the core. It's global-like sub-problem.

However, the true test of this approach is in a worst-case scenario, where all potentially interfering tasks are included. Task τ_2, in this case, plays a important role as it may interfere with other tasks. According to author, if task τ_2 can be scheduled under any processor affinities, then it is globally schedulable [27].

The approach to analyzing the schedulability of task τ_2 is based on a reduction to a "global-like sub-problem." The reduction process involves considering clusters of cores. In the provided example, the cluster in question consists of two cores, π_0 and π_3, which are marked in red. The global analysis takes into account the tasks that interact within this cluster, which are τ_0, τ_1, and τ_2.

The sub-problems arising from this global analysis are used to determine the schedulability of τ_2. If at least one of these sub-problems is schedulable, then τ_2 is also schedulable for the overall system. This approach effectively derives schedulability guarantees for APA schedulers by breaking down the problem into manageable sub-problems, thus reducing the complexity of the analysis.

Chapter 4

State-of-the-Art on Task Allocation

The problem of allocating tasks on computing resources is much older than using the multi-core platforms in embedded system. It is the problem of assigning tasks to a distributed system consists of independent computing resources. Hence, it has been covered by many research papers in the last decades. However, a multi-core platform has a special feature that more than one task can run in parallel sharing more resources, e.g. caches, than in distributed systems.

In this chapter, an overview on the problem of task allocation on a multi-core platform is presented. Mainly, this chapter compares and describes the Global first APA (gAPA) method, and the Partitioned first APA (pAPA) algorithm, which was developed by the DLR-OSS group, as well as information on its potential uses and limits. Focusing on the goal of improving schedulability through better task assignment, it suggests that the accuracy of both techniques could be enhanced by allocating tasks using an Integer Linear Programming (ILP) approach. Additionally, a summary of the technique is explained in this chapter.

As discussed in chapter 3, the idea of processor affinity enables tasks to be linked or tied to a particular core, which in some circumstances can enhance performance. On the other hand, if affinity is configured to be arbitrary, the task can run on any core that the scheduler selects, without any restrictions from the operating system or scheduler.

4.1 Tasks Allocation Methods for Multi-core Architectures

For real-time tasks on multi-core systems, deadline-aware scheduling requires careful consideration of task deadlines to guarantee timely execution. One popular strategy in this area is Earliest Deadline First (EDF). In EDF scheduling, tasks are prioritized depending on their deadlines, with the work with the earliest deadline given the most priority [1]. This strategy seeks to reduce the chance of missing crucial deadlines by prioritizing the completion of time-sensitive tasks.

Another popular strategy to assign tasks to cores statically is bin-packing algorithms. In the context of partitioned scheduling, bin packing algorithms are used to efficiently allocate tasks to cores. In this thesis, partitioned scheduling approach with various bin packing assignment strategies alongside other algorithms and introduced algorithm were used to do the schedulability test. In task assignment strategies, the "bins" refer to the "partitions" that available for the allocation of tasks whereas the "items" in this context correspond to the "tasks" that need to be assigned to the available partitions. The next sub-section describes what a bin packing algorithm is, the various kinds of bin packing algorithms, and how their code is implemented for usage with partitioned scheduling. The code of bin-packing algorithms are developed by DLR-OSS group.

4.1.1 Bin-packing Heuristics

Bin packing is a classic combinatorial optimization problem that involves packing a set of items of varying sizes into a minimum number of containers, typically called

bins, with each bin having a fixed capacity. The goal is to minimize the number of bins used while ensuring that no bin exceeds its capacity.

The formal definition of the bin packing problem is as follows [15]:

Given a set of items, each with a size or weight, a fixed bin capacity, and the objective of minimizing the number of bins required to pack all items.

The packing problem-solving is allocating n objects with varying weights and bins, each with a capacity of c, to a bin so as to minimize the total number of bins needed. It is reasonable to suppose that every item weighs less than the capacity of the bin.

An overview of approximate algorithms in brief

To address the bin packing problem, numerous bin packing algorithms and heuristics have been developed. Every algorithm has a different strategy. Figure 4.1 is a example of some of the popular bin packing algorithms along with a quick explanation of each: Height of items 5, 2, 11, 15, 7, 6, 17 and the height of bin is 20.

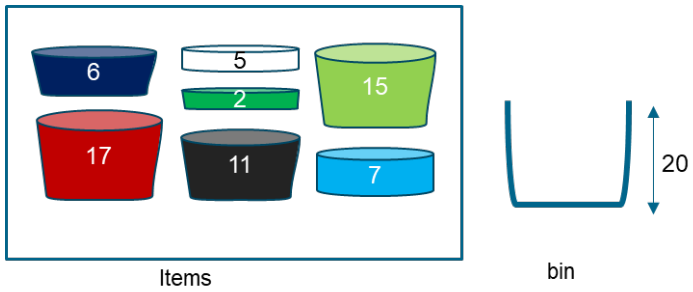


FIGURE 4.1: Example of the bin packing algorithm where items have different sizes but the bin has fix size.

1. **First-Fit (FF):**

   To solve the bin packing problem, the FF algorithm is a straightforward method. It functions like this:

   (a) Initialize a list of bins and an empty list of items to be packed.

   (b) For every item in the inventory:

   - Find the first bin in the list that can hold the item without going over its capacity by iterating over the list.
   - Once such a bin has been located, add the item to it and adjust the bin's remaining capacity.
   - Make a new bin and put the item inside if none of the existing ones can hold it.

   (c) For every item, repeat step b again.

   Allocate item j to the first bin where it fits:

$$\text{Let k} = \min\{i \mid \text{Capacity of bin } i \geq \text{Size of item } j\} \tag{4.1}$$

Allocate item j to bin k.

FIGURE 4.2: Illustration of the first-fit bin packing algorithm.

By using this algorithm total 4 numbers of bin are needed (shown in the Figure 4.2). The detailed algorithm to pack the items using first-fit method is shown in Algorithm 1.

Algorithm 1: First Fit Bin Packing Algorithm

```
// Allocate weights into bins based on the availability of space
Function first_fit(weights, bins, fit_function):
    everything_fit ← True                          // Assume everything fits initially
    foreach w in weights do
        fit ← False                                // Initialize fit flag for current weight
        foreach b in bins do
            if fit_function(w, b) is True then
                fit ← True
                b.append(w)                        // Add weight to the bin
                break
            end if
        end foreach
        if fit is False then
            everything_fit ← False
        end if
    end foreach
return everything_fit

// Fit function to check if a weight can fit into a bin
Function fit_function(weight, bin_):
    fit ← False
    if bin_.capacity ≥ weight then
        fit ← True
    end if
return fit
```

The algorithm 1 takes a list of weights, bins, and a fit function as input. It iterates through each weight. For each weight, it iterates through each bin and tries to fit the weight into the first bin where it can fit according to the fit function. If a weight fits into a bin, it is added to that bin, and the loop moves to the next weight. If a weight cannot fit into any bin, the algorithm sets a flag indicating that not everything fit. Finally, it returns True if every weight fits into a bin, otherwise False.

2. **Best-Fit (BF):**

The objective of the BF bin packing algorithm is to fit a collection of items(tasks) in different sizes into a minimum number of bins, or containers,

each having a specific capacity. This technique is one of the most effective bin packing heuristics since its goal is to reduce the amount of wasted space in the bins(partitions).

Regarding each item in the sorted list:

- Proceed through the list of bins one by one.
- While making sure the bin doesn't overflow, attempt to put the item into the "BF" bin-one that minimizes the amount of space left behind.
- Create a new bin and put the item in it if no existing bin can hold the item without going over capacity.

Allocate item j to the bin with the smallest capacity that can accommodate it:

$$\text{Let k} = \arg\min\{i \mid \text{Capacity of bin } i \geq \text{Size of item } j\} \tag{4.2}$$

Allocate item j to bin k.

By using this algorithm total 4 numbers of bin are needed (Shown in the Figure 4.3).

FIGURE 4.3: Illustration of the best-fit bin packing algorithm.

The detailed algorithm to pack the items using best-fit method is shown in Algorithm 2. The algorithm takes weights, a list of bins, and a fitting function as input. The given code provides 'best_fit', but the main bin-packing logic should be done in the 'remaining_Space_Fit function'. This function evaluates the fit function iteratively, trying to fit each weight into a bin. It determines the best-fitting bin for each weight by examining for the amount of space in each bin and applying a set of criteria. The weight is added into the appropriate bin if one is found; if not, the algorithm sets 'everything_fit' to False, indicating that the packing attempt was failed.

Algorithm 2: Best Fit Bin Packing Algorithm

```
1  Function best_fit(weights, bins, fit_function):
2  return _remaining_space_fit(weights, bins, fit_function, λa, b : a < b,
     bins[0].capacity)
3  /* space_cond (space condition) is the function to verify how the remaining space in the
     bin should be and default_space is starting value for the variable space            */
4  Function _remaining_space_fit(weights, bins, fit_function, space_cond,
     default_space):
5      everything_fit ← True                          // Assume everything fits initially
6      for each weight w in weights do
7          space ← default_space                      // Initialize space with default value
8          selected_bin ← None                        // Initialize selected bin to None
9          /* Iterate through each bin                                                    */
10         for each index i in range(length(bins)) do
11             // Check if weight fits into the bin
12             if fit_function(w, bins[i]) is True then
13                 actual_space ← bins[i].test_remaining_space(w)
14                 if space_cond(actual_space, space) is True then
15                     space ← actual_space
16                     selected_bin ← i
17                 end if
18             end if
19         end for
20         if selected_bin is not None then
21             bins[selected_bin].append(w)           // Pack the weight into the selected bin
22         else
23             /* If no bin was selected, set flag to indicate not everything fit        */
24             everything_fit ← False
25         end if
26     end for
27 return everything_fit
```

3. **Next-Fit (NF):**

   It is comparable to the FF algorithm, however it works especially well when items(tasks) come in a continuous manner or in sequential order. Items are put in one at a time using NF, which reduces the number of bins (partitions) needed.

   Follow these steps for every item on the list:

   - Verify that the item can fit in the given bin (that is, that its size does not exceed the capacity).
   - If it fits, put it in the bin that is currently in use and adjust the capacity that is available.
   - Create a new bin, put the item inside, then set the available capacity to the bin capacity if it doesn't fit.

Allocate item j to the next bin where it fits, starting from the last allocated bin:

$$\text{Let } k = \min \left\{ i \;\middle|\; \begin{array}{l} \text{Capacity of bin } i \geq \text{Size of item } j \text{ and } i > \text{Last} \\ \text{allocated bin} \end{array} \right\} \tag{4.3}$$

Allocate item j to bin k.

FIGURE 4.4: Illustration of the next-fit bin packing algorithm.

By using this algorithm total 5 numbers of bin are needed. That can be seen in the Figure 4.4. Algorithm 3 shows the full algorithm for packing items using the next-fit approach. The algorithm takes a list of weights, bins, and a fit function as input. It iterates through each weight. For each weight, it iterates through bins starting from the current index. It tries to fit the weight into the current bin using the fit function. If the weight fits into the current bin, it is packed into the bin, and the algorithm moves to the next weight. If the weight doesn't fit into the current bin, the algorithm moves to the next bin and repeats the process. After processing all weights, it checks if all weights were packed into bins. It returns True if every weight fits into a bin, otherwise False.

Algorithm 3: Next Fit Bin Packing Algorithm

```
Function next_fit(weights, bins, fit_function):
    i ← 0                                   // Initialize the index of the current bin
    packed ← 0                              // Initialize counter for packed weights
    for each weight w in weights do
        // Iterate through bins starting from the current index
        while i ⩽ length of bins do
            if fit_function(w, bins[i]) is True then
                bins[i].append(w)
                packed ← packed + 1
                break
            else
                i ← i + 1                   // Move to the next bin if weight doesn't fit
            end if
        end while
    end for
    // Check if all weights were packed
    if length of weights ⩾ packed then
        everything_fit ← False
    else
        everything_fit ← True
    end if
return everything_fit
```

4. **Worst-Fit (WF):**

WF bin packing is fitting a collection of items of varied sizes into a small number of containers (bins), each having a defined capacity. The algorithm finds the bin with the greatest unused space for each item, therefore referred to as "WF," with the goal of maximizing the amount of leftover space in each bin.

About each item in the sorted list:

- Proceed through the list of containers one by one.
- As long as the bin's capacity isn't exceeded, try to fit the item into the one with the greatest unused space.
- Create a new bin and put the item in it if no existing bin can hold the item without exceeding above capacity.

Allocate item j to bin with the largest capacity:

$$\text{Let k} = \arg\max\{i \mid \text{Capacity of bin } i \geq \text{Size of item } j\} \tag{4.4}$$

Allocate item j to bin k.

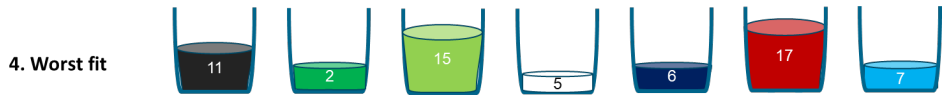


FIGURE 4.5: Illustration of the worst-fit bin packing algorithm.

By using this algorithm total 7 numbers of bin are needed (As shown in the Figure 4.5). Algorithm 4 provides a complete algorithm for packing the objects using the worst-fit approach. This algorithm works exactly same as BF algorithm works. Only the difference is value of argument *default_space* passes in function *_remaining_space_fit*. For the WF, *default_space* is set to 0 which means at every iteration, it will consider the bin is empty.

Algorithm 4: Worst Fit Bin Packing Algorithm

```
Function worst_fit(weights, bins, fit_function):
return _remaining_space_fit(weights, bins, fit_function, λa, b : a > b, 0)
Function _remaining_space_fit(weights, bins, fit_function, space_cond,
 default_space):
    everything_fit ← True                          // Assume everything fits initially
    for each weight w in weights do
        space ← default_space                      // Initialize space with default value
        selected_bin ← None                        // Initialize selected bin to None
        for each index i in range(length(bins)) do
            if fit_function(w, bins[i]) is True then
                actual_space ← bins[i].test_remaining_space(w)
                if space_cond(actual_space, space) is True then
                    space ← actual_space
                    selected_bin ← i
                end if
            end if
        end for
        if selected_bin is not None then
            bins[selected_bin].append(w)           // Pack the weight into the selected bin
        else
            // If no bin was selected, set flag to indicate not everything fit
            everything_fit ← False
        end if
    end for
return everything_fit
```

Bin packing is an optimization problem that involves properly fitting items into bins to reduce wasted space or increase resource efficiency.

There are some strategies such as task merging and task replication have been suggested to reallocate tasks when performance issues occurred. Task merging requires additional local memory, while task replication demands more processors to execute the same task [50]. A multi-core architecture which does not feature sufficient memory and processors will severely limit the available mapping options using the existing methodology [50]. Their study presents a framework for assigning tasks in soft real-time multi-core embedded systems, based on a prototype model. The framework provides a basis for effectively allocating tasks within the context of a multi-core architecture [23]. However, with the goal of enhancing schedulability through improved task assignment, an ILP-based technique is presented for APA-based hard real-time scheduling. The effectiveness of the strategy is proven in Chapter 5.

4.2 Task Allocation Approaches for APA

Although there are multiple variants of APA schedulers deployed in current real-time operating systems. However, no task allocation method applicable to tasksets for APA has been proposed to the date. The DLR-OSS group address this issue by applying the ideas which relate APA scheduling to the well-studied other

tasks allocation methods, and proposing simple and efficient techniques to allocate tasksets for APA scheduling. They uses shrinking and merging based task allocation for APA scheduling to analyze tasksets with APA and argue their correctness.

The present section introduces two specific approaches.

- **Global First Arbitrary Processor Affinity (gAPA)**
- **Partitioned First Arbitary Processor Affinity (pAPA)**

The gAPA strategy considering all cores as single affinity set before making specific affinity assignments, offering a global perspective first and then reduced the core set π_i. In contrast, the pAPA approach adopts a partitioned strategy, focusing on localized affinity assignments within specific partitions first and then merging the core set. Both approaches aim to mitigate issues in the current scheduler, leading to a reduction in the deadline miss ratio and an overall improvement in schedulability.

4.2.1 Global First APA (gAPA) Approach

The gAPA approach presents a strategy for enhancing the performance of real-time systems in a multi-core environment. This approach leverages the concept of APA to optimize task allocation. In multi-core real-time scheduling, the gAPA approach is designed to improve the schedulability of sporadic tasks. It builds on the idea of APA, where tasks are given affinities or preferences for specific cores.

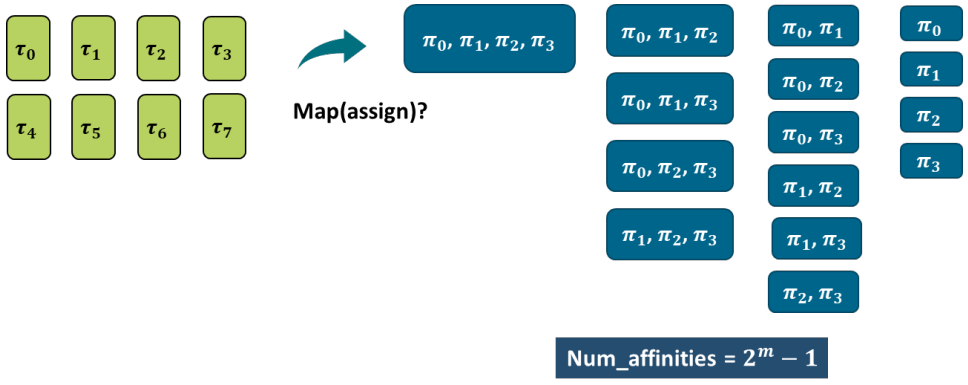


FIGURE 4.6: gAPA, shrinking based task assignment technique.

4.2.1.1 Shrinking Based Analysis

For a given task τ_i, global scheduling can be considered a special case of APA scheduling when the processor affinity $\alpha_i = \pi$, which means the processor affinity is equal to the entire processor set. Similarly, APA scheduling reduces to global scheduling for sub-problems with a restricted processor set α_i and a reduced taskset. This affinity-based approach aims to reduce interference and enhance system performance by strategically allocating tasks to cores that are most compatible with their affinities. In simple terms, if a task τ_i can be successfully schedule on a smaller subset of its affinity, it can also be scheduled successfully on the complete affinity set. This full scenario is presented in the Figure 4.6 and pseudo-code to develop this logic is presented in Algorithm 5 and 6.

Algorithm 5: Global First APA Scheduling Algorithm

Input : taskset, num_cores, priority_algo
Output: Scheduling success indicator

```
// Pass to the internal function _do_globalFirstAPA
Function globalFirstAPA(taskset, num_cores, priority_algo):
    success ← _do_globalFirstAPA(taskset, num_cores, priority_algo)
return success

// Function returns "True" if the taskset can be scheduled, "False" otherwise.
Function _do_globalFirstAPA(taskset, num_cores, priority_algo):
    // Assign priorities to tasks using the provided priority algorithm
    priority_algo(taskset, num_cores)
    cores ← [core.Core(i) for i in range(num_cores)]      // Initialize the CPU cores
    /* Iterate over tasks to shrink the CPU set based on global schedulability          */
    foreach current_task in sorted(taskset, key=lambda t: t.priority) do
        current_task.cpu_set ← cores              // Assign all cores initially to tasks
        foreach core_ in cores do
            core_.tasks.append(current_task)      // Add the current task to all cores
        end foreach
        // Calculate interference
        interference ← _return_tasks_in_cpu_set(current_task.cpu_set) -
          {current_task}
        affinity ← ShrinkingBasedAnalysis.do(cores, current_task,
          current_task.cpu_set, interference)
        if not affinity then
            return False                    // If task cannot be scheduled, return False
        end if
        current_task.cpu_set ← affinity      // Update the CPU set for the current task
    end foreach
return True
```

As described in Algorithm 5, the globalFirstAPA function acts as an interface that calls the _do_globalFirstAPA function, which performs the actual scheduling. _do_globalFirstAPA initializes the CPU cores, assigns priorities to tasks using a provided priority algorithm, and then iterates over tasks to shrink the CPU set based on global schedulability. ShrinkingBasedAnalysis class (described in Algorithm 6) implements the heuristic presented in paper [27]. It defines methods for performing shrinking-based analysis, checking schedulability, and removing candidate CPU cores based on certain criteria. The is_schedulable method checks whether a task is schedulable in given certain core affinity. The remove_candidate method removes a candidate (CPU core) from the core affinity based on the shrinking-based analysis. The 'do' method iteratively performs shrinking-based analysis until a schedulable CPU affinity is found or the algorithm terminates.

Algorithm 6: Shrinking Based Analysis

```
1  Class ShrinkingBasedAnalysis:
2      /* Performs the shrinking-based analysis to determine the core affinity of a task.
          This method returns list of core affinity after the shrinking-based analysis.  */
3      Method do(cores, task, affinity, interference)
4          backup_affinity ← [ ]                                  // Create an empty list
5          while True do
6              if task is schedulable then
7                  backup_affinity ← affinity
8              end if
9              (new_affinity, candidate) ← new core affinity and set of removed task
10             if new_affinity == affinity or not new_affinity then
11                 return backup_affinity
12             end if
13             interference ← interference - candidate
14         end while
15         return None
16     /* Checks if a task is schedulable with the given CPU affinity.  This method returns
          true if the task is schedulable, false otherwise.                              */
17     Method is_schedulable(task, affinity, interference)
18         cores_in_cpu_set ← len(affinity)
19         if cores_in_cpu_set > 1 then
20             gSchedulability ← do "GlobalSchedulabilityTest"
21             schedulable ← "compute response time and check if task schedulable"
22         else
23             schedulable ← "compute busy window and check if task schedulable"
24         end if
25         return schedulable
26     /* Removes a candidate CPU core from the CPU affinity based on the shrinking-based
          analysis.  This method returns tuple containing the new core affinity and the set of
          removed tasks.                                                                 */
27     Method remove_candidate(task, affinity)
28         return (new_affinity, t[RC.index(c_prime)])
29 return
```

In conclusion, the gAPA approach try to assign task by shrinking the processor affinity set. This methodology provides an arrangement for analyzing the global schedulability of tasks, which ultimately leads to better system performance and predictability.

However, there is an opportunity for improvement in defining the best core affinity and priority assignments. Hence, an alternative approach, offers a new strategy for task allocation that differs from gAPA approach. This approach, referred to as the pAPA, follows a bottom-up approach.

4.2.2 Partitioned First APA (pAPA) Approach

The tasks assignment procedure is fundamentally reversed than the gAPA or using bottom-up methodology in the pAPA approach as shown in Figure 4.7. It begins by allocating tasks to specific core first, similar to partitioned scheduling. This initial

allocation step prioritizes creating smaller, more manageable groups of tasks, each associated with a single core. The pAPA method then uses a merging process, combining two of these smaller groups and reassessing the tasks assigned to the remaining set. This process of iteratively merging and reassigning groups continues until no further group settling is feasible.

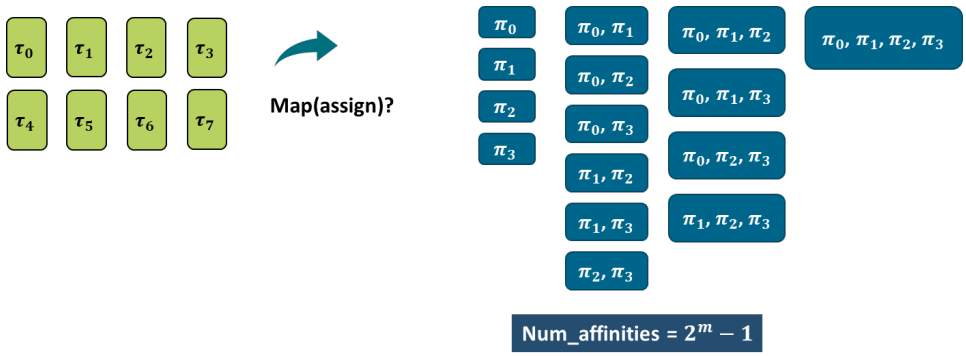


FIGURE 4.7: pAPA, merging based task assignment technique.

The aforementioned assessment methodology gives information about the approach's effectiveness as well as an established standard by which to compare it to other approaches.

Algorithm 7: Algorithm for Computing Combinations

Input: Parameters n, k
Output: Result of the combination

```
// Compute the number of combinations
Function compute_num_comb(n,k):
    n_fac ← factorial(n);
    k_fac ← factorial(k);
    nmink_fac ← factorial(n − k);
return n_fac/(k_fac × nmink_fac)
```

Algorithm 7 is used to compute number of combinations merging cores using factorials where n is number of cores and k is combination size. this algorithm compute the factorial of (n-k) and return the number of combinations which is useful in main algorithm later in Algorithm 8. Algorithm 8 performs partitioning using the First APA algorithm, which assigns tasks to CPU cores using FF task allocation method. It assigns priorities to tasks, initializes unallocated tasks, and creates core affinity. The function iteratively partitions cores into containers with increasing core combinations until all tasks are allocated or no more cores are available for merging. Algorithm 9 merges CPU cores into containers based on the specified combination size using the "itertools.combinations" function.

Algorithm 8: Partitioned First APA Algorithm

```
1  /* Perform partitioning using the First fit scheduling algorithm.  This function return
      true if all tasks are allocated, otherwise return false                          */
2  Function partitioningFirstAPA(taskset, num_cores, priority_algo, first_fit):
3      // Assign priorities to tasks using the specified priority assignment algorithm
4      priority_algo(taskset, num_cores) // Set an empty list to task's cpu_set
5      foreach task in taskset do
6          task.cpu_set ← [ ]
7      end foreach
8      comb_size ← 1
9      last_id ← num_cores - 1
10     /* Repeat partitioning until all tasks are allocated or no more cores to merge   */
11     while unallocated_tasks and comb_size ≤ num_cores do
12         containers, invol_cores ← merge_cores(cores, comb_size, last_id)
13         last_id ← last_id + (compute_num_comb(num_cores, comb_size)-1)
14         comb_size ← comb_size + 1
15     end while
16     // Return True if all tasks are allocated, otherwise return False
17     if unallocated_tasks then
18         return False
19     else
20         return True
21     end if
22 return
```

Algorithm 9: Algorithm for Merging Cores

Input: List of cores *cores*, *comb_size*, *last_id*
Output: List of containers, Dictionary of involved cores

```
1  // Merge cores into containers
2  containers ← [ ] ;                    // Initialize an empty list to store containers
3  // Initialize an empty dictionary to store involved cores for each container
4  invol_cores ← {};
5  Function merge_cores:
6      // Iterate over combinations of cores
7      foreach comb in combinations(cores, comb_size) do
8          last_id ← last_id + 1 ;              // Increment the ID for each new container
9          new_container ← create_new_container;
10         // Iterate over cores in the combination
11         for i ← 0 to comb_size − 1 do
12             // Add the task to the container and update the capacity of the container
13             add_tasks_and_capacity(new_container, comb[i]);
14             // Add the core to the list of involved cores
15             invol_cores[new_container.id].append(comb[i]);
16         end for
17         containers.append(new_container);
18     end foreach
19 return containers, invol_cores
```

In order to validate the results, pAPA was carefully tested using the first-fit bin-packing heuristic. Even if the results support the effectiveness of pAPA, it is important to acknowledge that the algorithm's full performance evaluation is still lacking. In particular, it is necessary to check the effect of different bin-packing techniques, e.g."best-fit","next-fit" and "first fit" on system performance.

Therefore, the analysis of pAPA approach with different bin packing algorithms and different priority assignments is carried out in this thesis which is explained in detail in Chapter 4, specifically in Sections 6.2 and 6.3.

Chapter 5

Optimized Affinity Sets for APA

This chapter investigates the field of ILP - based algorithms in an effort to expand the boundaries of optimization. The objective is to develop an ILP-based algorithm capable of optimizing affinity sets. This exploration not only contributes to the ongoing advancements in task assignment techniques but also signifies a broader commitment to enhancing the efficiency and precision of real-time systems.

5.1 Integer Linear Programming (ILP)

Integer Linear Programming (ILP) is a mathematical method that aims to optimize a linear objective function while considering linear constraints. It allows for discrete decision variables in addition to continuous ones. This means that ILP enables the solution to problems where decisions must be made from a limited set of choices rather than as a continuous range. ILP involves finding the optimal solution by selecting integer values for decision variables within specified constraints, adding an additional layer of complexity and precision to problem-solving compared to standard linear programming. The linear objective and constraints must consist of linear expressions.

The foundation of problem design leverages the principles of ILP to achieve optimization. In this scenario, the aim is to maximize the objective function by strategically assigning the highest priority tasks or a task which has higher utilization, initially to first row of core affinities. The key component is to create constraints that precisely manage the work allocation process.

In this pursuit, two primary constraints come to the forefront: the first constraint ensures that each task is allocated to precisely at least one affinity. This condition guarantees a well-defined assignment scheme, avoiding ambiguities or overlaps in task distribution across core affinities.

The second constraint is the more critical aspect of meeting task deadlines. The design here focuses on conforming to task deadlines strictly by implementing restrictions that limit the response time of each task within the set deadline. This constraint is pivotal in ensuring that the system's real-time requirements are met, avoiding any potential delays that could compromise task completion within the prescribed time frame.

The ILP is carefully built to handle the unique complexity of the tasks allocation problem by organizing the problem with these limitations and an objective function intended at increasing task allocation efficiency while sticking to fundamental affinities and achieving deadlines. This use of ILP is a sophisticated strategy that adds layers of accuracy and complexity to the problem-solving process, increasing the possibility for optimum task allocation inside a multi-core real-time scheduling context.

5.2 An Efficient ILP Solution

As shown in Figure 5.1, consider a scenario where set of tasks $\tau = \tau_1, \tau_2, \tau_3 \ldots \tau_7$ needs to assigned to the core affinities then, using ILP based approach for optimized task allocation could be valuable to solve the problem. It emerges as a significant tool in task scheduling, providing a systematic way to accurately specify and optimize task allocations to processor cores. This is especially important in multi-core systems to ensure effective resource use.

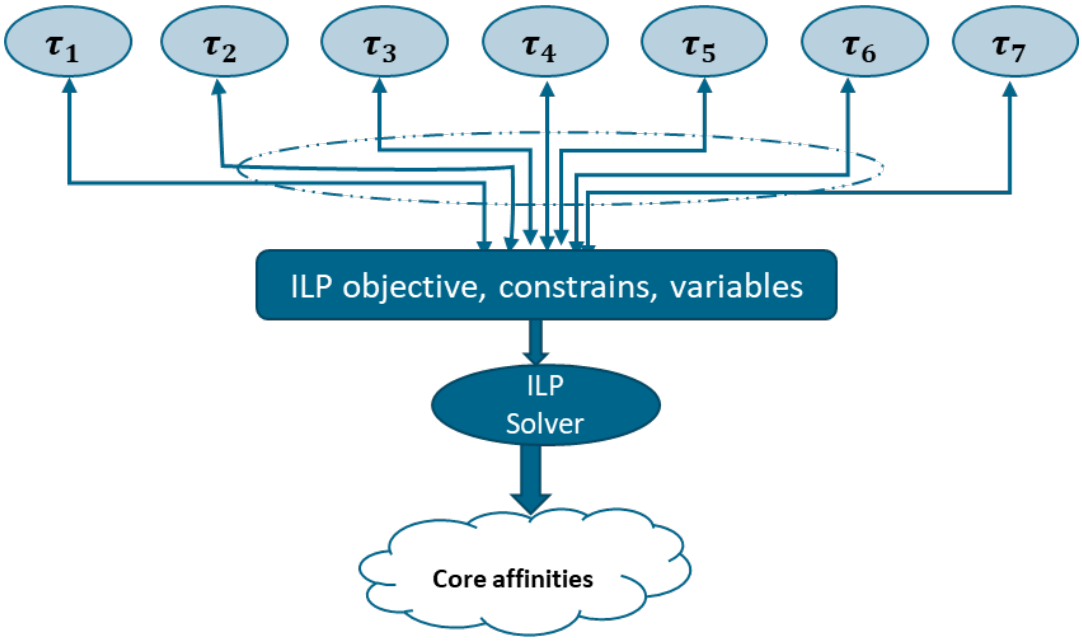


FIGURE 5.1: Tasks τ from 0 to 7 needs to assigned to the core affinities using ILP based approach by defining objective function, constrains and variables.

ILP Formulation for Task Assignment:

In the realm of multi-core systems, ILP can be applied to task assignment problems where each task must be allocated to a specific processor core. The decision variables in this scenario are binary, representing whether a task is assigned to a particular core or not. The objective function typically aims to minimize or maximize a certain criterion, such as total execution time or total number of schedulable tasks.

The formulation of the objective function is important for optimizing task allocation for multi-core real-time scheduling using ILP. The objective function can be divided into two assumptions: one that works to maximize the function by assigning highest utilization task first, or another that aims to maximize the objective function by assigning the highest priority tasks first to the core affinities.

The general form of ILP for task assignment is represented as follows:

$$\text{Maximize} \quad \sum_{\alpha \in A} \sum_{i \in T} u_i x_i^{\alpha} \tag{5.1}$$

OR

$$\text{Maximize} \quad \sum_{\alpha \in A} \sum_{i \in T} prio_i x_i^{\alpha} \tag{5.2}$$

The objective function aims to maximize the sum of the utilization values (u_i) of tasks (As per Equation 5.1) or by assigning highest priority tasks first across different affinity sets (α) (As per Equation 5.2). Each x_i^{α} is a binary decision variable, indicating whether task i is assigned to the affinity set α or not. $prio_i$ is the assigned priority of each task and u_i is the utilization of each task.

Through these different viewpoints, this approach aims to maximize a possible advantage in maximizing the number of schedulable test cases. The formulation includes constraints that specifically enforce to an affinity set, in which the total response time (weight) of tasks assigned up to a given task i within this set of tasks does not exceed the deadline (capacity) D_i. This constrain is derived based on the RTA schedulability test (See Equation 3.4 in Section 3.2.2).

Capacity Constraint:

$$C_i \cdot x_i^{\alpha} + \left\lfloor \frac{1}{m} \sum_{j<i} [x_j^{\alpha} \cdot \widehat{w}(\theta, \beta)] \right\rfloor \leq D_i, \quad \forall i < N \tag{5.3}$$

where:

$$\begin{aligned} \mathrm{D}_i &= \text{Capacity or deadline of task}i \\ \mathrm{C}_i &= \text{Execution time of task } i \\ \widehat{w}(\theta, \beta) &= \text{Response time between tasks } i \text{ and } j \\ \theta &= \left\lfloor \left(\widehat{W}_j(\mathrm{WCRT}_i) \right) \right\rfloor \\ \beta &= \mathrm{WCRT}_i - C_i + 1 \end{aligned}$$

Equation 5.3 is the response time of task j according to RTA. This constraint ensures that the response time of tasks assigned to affinity set α up to task i does not exceed the deadline (capacity) D_i. C_i represents the execution time of task i, and $\widehat{w}(\theta, \beta)$ is a function that considers the interaction weight or response time between tasks i and j.

Binary Decision Constraint:

$$x_i^{\alpha_1} + x_i^{\alpha_2} + x_i^{\alpha_3} + \ldots + x_i^{\alpha_n} \leq 1 \tag{5.4}$$

This constraint ensures that each task i is assigned to at most one affinity set. The sum of binary decision variables (x_i^{α}) across different affinity sets for task i is limited to 1.

Binary Decision Variable:

$$x_i^{\alpha} \in \{0, 1\}, \quad \forall i < N \tag{5.5}$$

This variables specifies that each binary decision variable x_i^{α} can only take values 0

or 1, representing the absence or presence of task i in affinity set α. If the value is 0, it indicates that the task is not assigned to any core affinities; if it is 1, then the task is assigned to a core affinity.

5.2.1 Tailored Task Scheduling Solutions: Versatility of ILP in Adapting to Varied Constraints

The use of ILP method gives an appealing approach to dealing with various constraints in task scheduling issues. Its adaptability derives from its capacity to quickly adopt extra limitations as required by individual circumstances. For example, adding a new constraint requiring that tasks i and j not work together inside the same processor affinity can be effortlessly included into the ILP formulation. ILP is a good solution for dealing with complicated task scheduling scenarios because of its ability to easily adjust and include such delicate limitations.

Safety and Security Constraint:

$$x_i^\alpha + x_j^\alpha \leq 1, \quad \forall i, j < N \tag{5.6}$$

This constraint enforces safety and security considerations by ensuring that tasks i and j are not assigned to the same affinity set. This prevents certain tasks from coexisting in the same set for safety or security reasons.

The fundamental flexibility of ILP enables for the introduction of many limitations that may come out in actual circumstances, such as task dependencies, resource limits, or task affinity constraints. This is especially useful when task scheduling requires complex interactions between tasks, processors, and associated constraints. By adding constraints to the ILP formulation, it is possible to clearly specify and enforce rules that govern the allocation and execution of tasks among processor cores, assuring that they satisfy stated criteria.

Furthermore, the flexibility and accuracy of ILP formulations contribute to their ease of use in dealing with changing constraints or changes to the issue statement. The ILP concept provides a systematic and organized way to accommodating changes when needs develop or new restrictions are imposed. Its essential flexibility supports a strong solution approach, allowing for the smooth incorporation of developing restrictions without the need for significant reworking or redesign of the underlying scheduling model.

Thus, the use of ILP approach is beneficial in solving complicated task scheduling difficulties, as it provides a dynamic and efficient methodology capable of accommodating developing restrictions and assuring the optimization of scheduling results in a variety of computing situations.

5.3 ILP Objective: Utilization vs Priority as Weight

The implementation of ILP in this experiment aims to optimize the allocation of tasks by prioritizing tasks based on their utilization levels or by maximizing highest priority tasks first to the core affinities. The ILP formulation focused on maximizing the utilization of computational resources by assigning tasks with the highest resource requirements first. Python was employed to implement this approach,

encoding the ILP model to prioritize tasks according to their utilization metrics or by priority.

5.3.1 System Model

In this section, task dependencies are added to the system model introduced in Chapter 2. Analyzing multi-core systems with a finite number of n different tasks that are scheduled using the fixed priority preemptive scheduling which is commonly used in real-time systems. With fixed priority preemptive scheduling, the scheduler ensures that at any given time, the processor executes the highest priority task of all those tasks that are currently ready to execute. A task set is a collection of separate tasks with varying utilization and worst-case execution times. A task in a system must belong to exactly one chain.

Core Affinity Assignment Strategy

Three distinct assignment strategies are observed:

1. Solving the problem as one ILP for all possible core affinities by a defining objective for the entire problem (Figure 5.2 (1)).

2. Solving the problem iteratively as one ILP per affinity set, where each affinity has its own objective (Figure 5.2 (2)).

3. Solving the problem iteratively as one ILP per row, considering each row as a separate ILP problem and defining objective functions and constraints for each row independently(Figure 5.2 (3)).

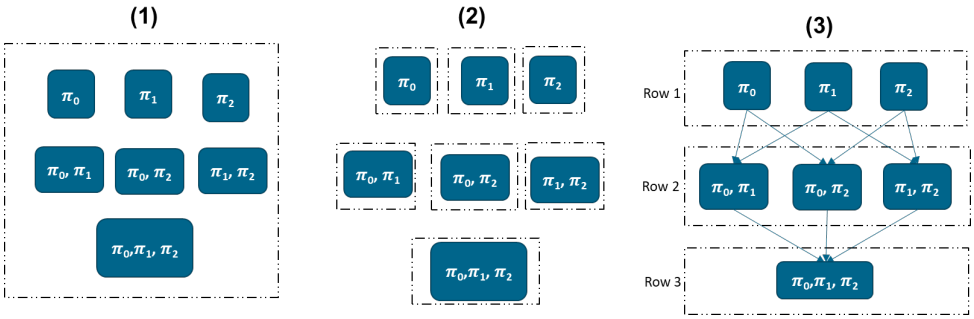


FIGURE 5.2: Core affinity assignment strategy.

However, examining options 1 and 2 becomes more complicated since the problem size in these two approaches is huge. As a result, choosing the third alternative is preferable when considering complexity and quality of the results. This strategy divides the problem into manageable sub-problems, progressively improving the ILP model. Core affinities are organized into rows, with each row reflecting a different stage in the ILP formulation. The row-by-row approach has two variations:

1. Top-down approach

2. Bottom-up approach

The top-down approach as shown in Figure 5.3, in which tasks are initially assigned to the group of all available cores in the first row, and the unassigned task list is

passed to the next row with a modified core affinity group. Consider this row to be an individual ILP problem. Define the objective function, specify the constraints, and then solve the issue with an ILP solver. After solving the ILP for the first row, a list of unassigned tasks will be generated, which are the tasks from the lists that were not allocated to any core affinities in the first allocation.

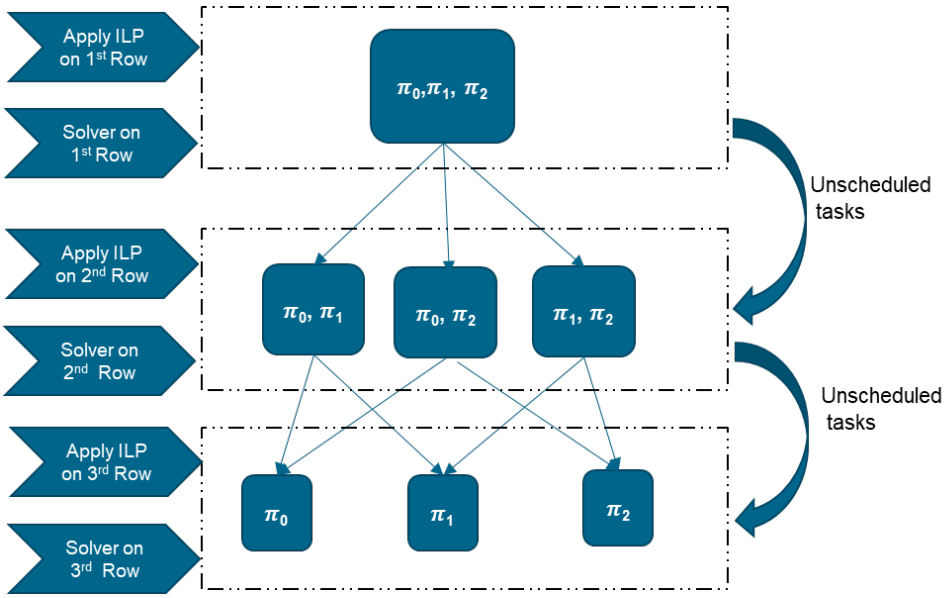


FIGURE 5.3: Divide and conquer in the top down order.

And the bottom-up approach as shown in Figure 5.4, in which tasks are assigned to single-core affinities first, and cores are added iteratively in consecutive rows. The selected technique of using a bottom-up approach for core affinity assignment requires a systematic methodology in the context of task scheduling and affinity allocation. After careful consideration and analysis of results, it is found that the bottom-up approach proves more beneficial in balancing complexity and result quality. As a result, bottom-up strategy is selected for the ILP approach.

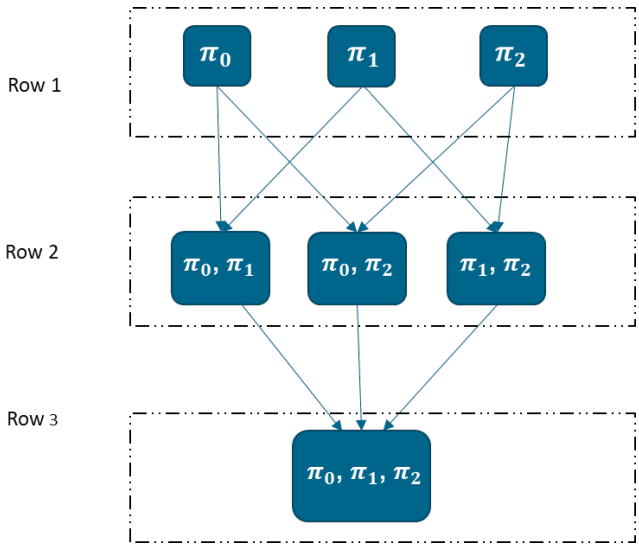


FIGURE 5.4: Divide and conquer in the bottom up order.

Tasks are allocated to core affinities based on Figure 5.4. This involves traversing through the affinity lists in a reversed order, assigning tasks to cores iteratively from the highest to the lowest hierarchical levels. Each iteration allocates tasks to cores affinity within the reversed affinity list structure, ensuring optimal task-to-core mappings based on predefined criteria such as utilization, priority, or other specific constraints.

By employing the bottom-up approach, the experiment aims to systematically assign tasks to core affinities facilitating an optimized task allocation strategy tailored to the specific requirements and constraints of the scheduling problem.

5.4 Evaluation of ILP Strategies: Utilization-Centric vs. Priority-Driven Task Allocation

This section describes the approach used to perform a thorough comparison of two unique ILP methodologies. The experiment specifically examines the efficacy of two different objective functions, one stressing utilization-centric task allocation and the other assigning tasks based on highest priority.

The experimental procedure includes the use of an ILP technique, as shown in the preceding schematic model (As shown in Figure 5.4). The experiment sets out to evaluate the performance of the ILP model in a variety of computing situations that have different core counts, utilization ranges, and task sets.

The following parameters were taken into account when designing the experiments:

- **Core Configuration:** The experiment was carried out using several core configurations, specifically core counts of 4, 5, 6, and 8.
- **Utilization Range:** The utilization ranges were chosen from number of cores/2 to number of cores with steps of 0.5.

$$u \in \left[\frac{\pi}{2}, \pi\right] \tag{5.7}$$

  and mathematically, it can be represented as follows:

$$u(\pi) = \{\frac{\pi}{2}, \frac{\pi}{2} + 0.5, \frac{\pi}{2} + 1, \ldots, \pi\}$$

  For an example, let's consider when $\pi = 4$. In this case, the utilization range $u(4)$ is given by:

$$u(4) = \{2, 2.5, 3, 3.5, 4\}$$

- **Number of Task:** The tasks were randomly generated starting from the range of number of cores*2. Let the list of taskset denoted by L and the number of cores be denoted by 'core'. In a mathematical expression, the list elements can be defined recursively as: L[n] = L[n-1] + core for n > 0

For example, when core = 4:

$$L[0] = 4 \times 2 = 8$$
$$L[1] = L[0] + 4 = 8 + 4 = 12$$
$$L[2] = L[1] + 4 = 12 + 4 = 16$$

Therefore, when number of core = 4, number of taskset is [8, 12, 16]

- **Total Number of Test Cases:** For each experimental run, a total of 1000 test cases were generated.

The ILP-based experiments were carried out in a setting with the aforementioned core configurations, number of tasks sets and utilization ranges. The ILP model was tested using two different objective functions to see how they affected the assignment of tasks within the current computing environment.

Task Characteristics: Each individual test case within the experiment represents a unique set of task parameters. These parameters include task periods, randomly chosen from a predefined range determined by a period scaling factor (1, 2, 3, 4, 5, 6, 7, 8) multiplied by predefined period ranges (280.0, 340.0, 450.0, 500.0).

T_i = random.choice(periodsScalingFactor)*random.choice(periodsRange)

Additionally, the deadlines for each task (D_i) were set equal to the minimum inter-arrival time, contributing to diverse and dynamic task compositions across the experiment.

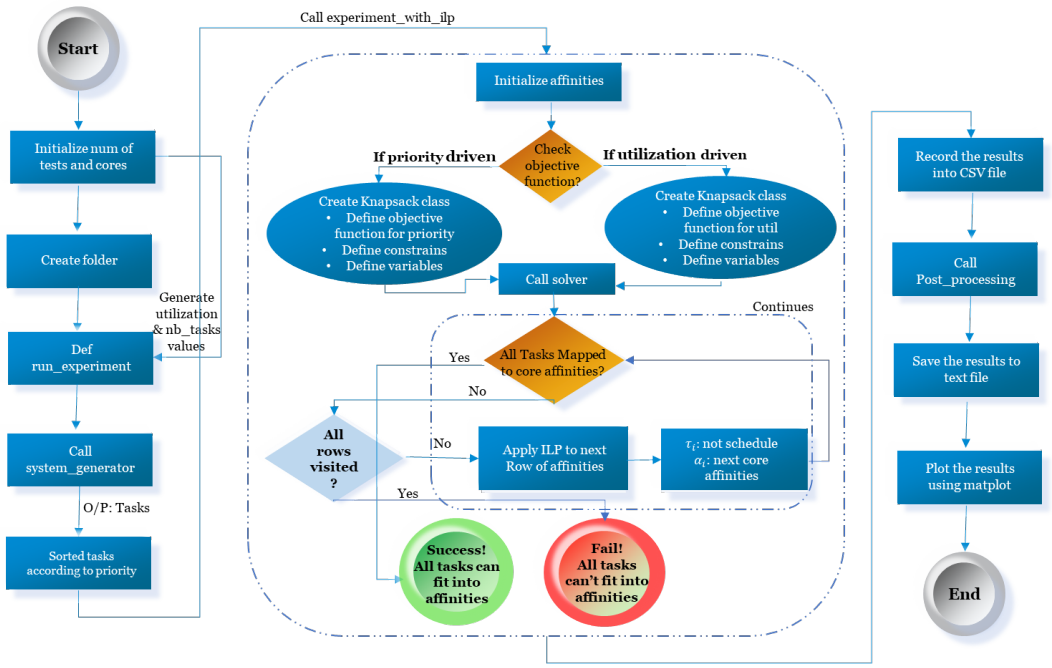


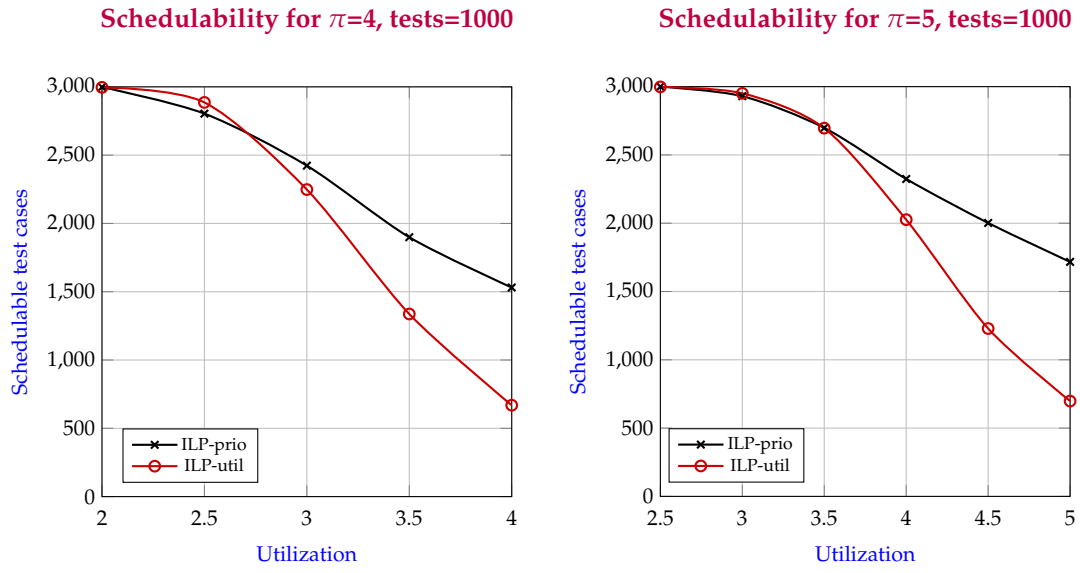
FIGURE 5.5: Flowchart showing the setup for an ILP experiment.

As shown in Figure 5.5, the experiment begins by initializing the number of tests and cores. Using given number of cores, generate utilization and number of tasks

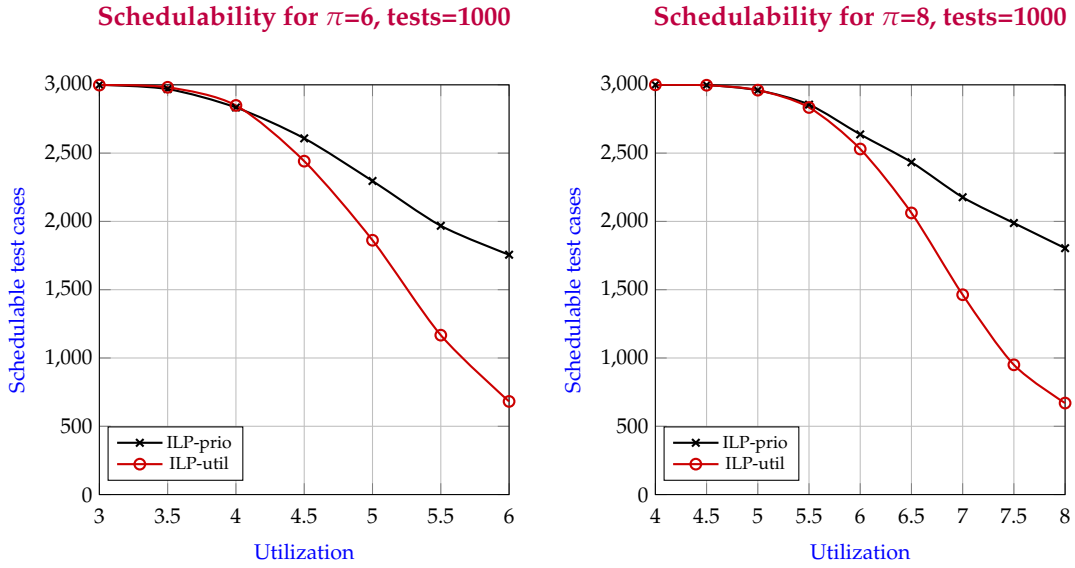
values which are explained in Equation 5.7 on page 39. This value is passed as an input to "run_experiment" method. And this method runs experiments with different system configurations. The "system_generator" function is called, and as an output, it produces random sets of tasks. After getting tasks, all tasks are sorted according to priorities. ILP is used to prioritize tasks based on either priority or utilization objectives. After this, call “experiment_with_ilp” method. According to this method, it will first initialize affinities with help of given number of cores. A Knapsack class is created, which defines the objective function and constraints based on whether the experiment is priority-driven or utilization-driven. After defining objective function, constrains and variables, call “ilp_solver” from the ‘Knapsack’ class. After solving for first iteration, it will checked that are all tasks mapped to core affinities of first row? If all tasks are mapped, then it will print “Success! All tasks can fit into affinities” into the console. If not, then it will check for other rows. The objective function, constraints, and variables will only be defined for the tasks that were left unassigned in the previous iteration, and the following row will treat it as a separate ILP problem. And again check the condition. If all rows visited and there are still some unassigned tasks available, it will print “Fail! All tasks can’t fit into affinities”. This process continues until all tasks are mapped or it is determined that all tasks cannot fit into the affinities. The results are recorded in a CSV file and a text file. Next call “post_processing” method. This method updates result curves based on experiment outcomes and “plot_results” method generates and saves plots based on experiment results. The experiment ends, reporting success if all tasks fit into the affinities, or fail if they don’t.

5.4.1 Results and Interpretation of Outcomes

The obtained results, depicted in the Figure 5.6 , present a comparative analysis between the two ILP objective functions across 1000 test cases. The x-axis indicates individual utilization value of task, while the y-axis reflects schedulable test cases. These results help to clarify the performance significance and variations found between utilization-focused and priority-driven ILP techniques within the given computational environment.



(A) The comparison of utils vs priority for 4 cores (B) The comparison of utils vs priority for 5 cores



(C) The comparison of utils vs priority for 6 cores (D) The comparison of utils vs priority for 8 cores

FIGURE 5.6: Experimental graphs to compare utils vs priority with various cores.

The results derived from this experiment provide valuable insights into the effectiveness and suitability of ILP objective functions concerning task assignments in varied computational scenarios. The results show that assigning tasks based on their priority allows for a much enhanced scheduling of a higher number of test cases inside the computing environment analyzed. When the priority-driven objective function in the ILP model is used, a better success rate in task allocation can be achieve than the utilization-driven objective function.

This conclusion emphasizes the importance of task prioritizing techniques in maximizing task scheduling, particularly when the primary goal is to effectively handle a higher number of test test cases. As a result, a priority-based objective function has been taken into account for the further ILP experiments, improving the overall approach for task scheduling efficiency.

Chapter 6

Experiments

This chapter highlights various kinds of algorithms ideal for real-time experiments, building upon the insights presented in preceding chapters. It elaborates on the real-time exploration process using diverse approaches. Furthermore, it combines the latest developments in assigning analysis, presenting significant in general implications obtained from each experiment, followed by the results.

This section explains more details on how the algorithms described in this work. Python 3.8 and its libraries, as well as other packages, were used for development, testing, and experimentation. To start with, Matplotlib is used for interactive visualizations, Math library is used for mathematical function calculation, and NumPy is used to produce random Minimum Inter Arrival Time (MIAT) and uniform distribution, which is necessary to achieve utilization.

The following sub-sections explain the implementation of each module separately and end with a section that combines all the modules.

To comprehensively assess the proposed analysis, a series of synthetic test scenarios were developed in order to evaluate diverse system configurations. These experiments studies include evaluations of pAPA, gAPA, partitioned, global scheduling policies, alongside an examination of their performance under varied conditions such as system load or utilization, scalability, and task assignments. The study focuses on several key aspects:

- A comparison of the pAPA and gAPA algorithms using different bin packing approaches to determine superior performance.
- Examination of how task overload influences the assurances provided by the scheduling policies.
- Analysis of the computational run-time across different algorithms, including their comparative efficiency.
- Assessment of the impact of priority assignment on the resultant assurances.
- Evaluation of the superiority of unique ILP-based methodology over conventional approaches.

6.1 Synthetic Test Case Generation

The synthetic test cases developed consist of tasks worst-case execution times, periods or minimum inter arrival time, CPU sets, and relative deadlines. The complete pseudo-code to generate synthetic test cases is shown in Algorithm 12 on page 46. These tasks, chosen for their periodic behavior in this experiments, were generated following a structured methodology:

- Determining the number of tasks and the intended system utilization shared among them.

- Allocating the system's utilization to each task using the UUnifast method [16] to ensure fair distribution.

- Assigning periods to each task randomly from a predefined set of harmonic values. Subsequently, computing the worst-case execution time for each task (C_i) as follows:

$$C_i = u_i \cdot T_i \tag{6.1}$$

  where T_i representing the period of task τ_i. To ensure a diverse spectrum of system models, we varied the following parameters:

- The tasksets were randomly generated starting from the range of number of cores*2. It is generated same as a way explained in Section 5.4. The Algorithm 10 is used to create new task object.

Algorithm 10: Task Class

```
Class Task:
    // Constructor method to initialize a Task object
    Method __init__(taskId)
        // Set task attributes
        self.id ← taskId
        self.priority ← None
        self.miat ← None                    // Set minimum inter arrival time
        self.wcet ← None                    // Set worst case execution time
        self.deadline ← None
        self.util ← None
        self.cpu_set ← []
return
```

- Total utilization (U) starting from number of cores/2 to number of cores. It is also generated same as explained in Section 5.4 in Equation 5.7.

- Allocating relative deadlines (D_i) to tasks based on $D_i = T_i$. This defined that a task possessed an implicit deadline ($D_i = T_i$).

In order to simplify the analysis of parameter impact, we selected to have a range of values rather than discrete intervals. These settings apply to all experiments unless otherwise mentioned.

Algorithm 11: UUniFast

Input : num_cores, utilization, NbTasks
Output: vectU

```
// Generates a list of task utilizations using the UUniFast algorithm introduced in [16].
Function UUniFast(n, U, M):
    if U > M or U ≤ 0.0 then
        // Check if U is within valid range
        PrintError("U is the upper bound utilization and it should be
         0 < U < M, U =", U, "M =", M)
        ExitProgram
    end if
    if n ≤ 0 then
        // Check if n is valid
        PrintError("N is the number of tasks and it should be N > 0, N =", n)
        ExitProgram
    end if
    if Type(n) == float then
        // Check if n is an integer
        PrintError("N is the number of tasks and it should be an integer")
        ExitProgram
    end if
    vectU ← [ ]                                  // Initialize list for task utilizations
    sumU ← U                                     // Initialize sum of utilizations
    for i in range(0, n - 1) do
        // Generate utilization for each task except the last one
        nextSumU ← sumU * (RandomUniform(0, 1) ** (1 / (n - i)))
        if sumU - nextSumU < 1.0 then
            vectU.append(sumU - nextSumU)
            sumU ← nextSumU
        else
            break
        end if
    end for
    if sumU ≥ 1.0 then
        // If the last task would have utilization greater than or equal to 1, return an
           empty list
        return [ ]
    else
        vectU.append(sumU)                       // Add utilization of the last task
        return vectU
    end if
return
```

The Algorithm 11 implements the UUniFast algorithm, as introduced by Bini and Buttazzo in 2005 [16]. UUniFast is a method for generating utilization values for a set of tasks using a uniform distribution. This algorithm ensures that the total utilization of tasks equals the specified upper bound U while distributing the utilizations uniformly. It iteratively generates utilization values for each task, ensuring that the sum of utilizations equals upper bound utilization. Utilization for each task

is generated using a random value between 0 and 1, scaled appropriately to ensure the sum remains within the upper bound U. The generated utilizations are added to a list vectU, and if the utilization of the last task would exceed 1, an empty list is returned. Various error checks are performed to ensure that the input parameters (U, M, n) are within valid ranges and of the correct types where M is maximum utilization per task, n is number of tasks and U is upper bound utilization. This particular implementation is provided by the DLR_OSS group.

Algorithm 12: Generate Synthetic Testcases

Input : num_cores, utilization, NbTasks
Output: tasks

```
1  Function generate_system(num_cores, utilization, NbTasks):
2      tasks ← dict( )                          // Initialize dictionary to store tasks
3      while not t_utilization do
4          // Generate task utilizations using UUniFast algorithm
5          t_utilization ← UUniFast(NbTasks, utilization, num_cores)
6      end while
7      for idx in range(0, NbTasks) do
8          tasks[idx] ← model.Task(idx)                    // Create a task object
9          util ← t_utilization[-1]                        // Create a task object
10         // Assign period using period scaling factor as explained above
11         miat ← (periodsScalingFactor ∗ periodsRange) deadline ← miat // Assign
            deadline
12         wcet ← (util ∗ miat, 2)                                  // Assign wcet
13         // Adjust deadline if wcet exceeds it
14         if wcet ≥ deadline then
15             deadline ← miat
16             wcet ← wcet - 0.01
17         end if
18         // Handle case where wcet is zero
19         if wcet == 0 then
20             wcet ← 0.001
21         end if
22     end for
23 return tasks
```

Algorithm 12 is used to generates a system with tasks for scheduling. It takes three parameters: num_cores (number of CPU cores), utilization, and NbTasks (total number of tasks) and returns the dictionary containing the generated tasks.

6.2 Priority Assignment Algorithms: Their Influence on Various Approaches

When talking about various priority assignment, it means analyzing the effectiveness of heuristic priority assignment policies like deadline monotonic and DkC, an extension of TkC priority assignment policy that has been used to any schedulability test [21]. Two distinct priority assignment policies can be utilized based on that statement.

- Deadline monotonic priority assignment (DM)
- DkC Priority assignment

Following the assignment of priorities to tasks, the experiment performs tasks assignments based on the applied algorithms. These tests aim to assess whether the assigned priorities enable feasible task scheduling within the system constraints.

6.2.1 Deadline Monotonic Policy

Deadline Monotonic Scheduling (DMS) is a scheduling method that is used in real-time systems to ensure that tasks are completed by the deadline. It is a sort of fixed-priority preemptive scheduling in which higher-priority tasks always take preference over lower-priority task. DMS was developed on the concept that tasks with shorter deadlines should be prioritized higher [7]. This guarantees that the most significant tasks are always completed first, limiting the possibility of missed deadlines.

Consider the Table 6.1 to better understand how DMS works:

ID	Period (ms)	Deadline (ms)	Execution time (ms)
Task 1	20	7	3
Task 2	5	4	2
Task 3	10	9	2

TABLE 6.1: Task allocation to understand DMS works.

Task 2 in this case has the lowest deadline (4 ms) and hence has the greatest priority. Task 1 has a slightly longer deadline (7 ms) and should be prioritized second. Finally, Task 3 has the longest deadline (9 ms) and should be prioritized the least.

$$T_3 \leqslant T_1 \leqslant T_2$$

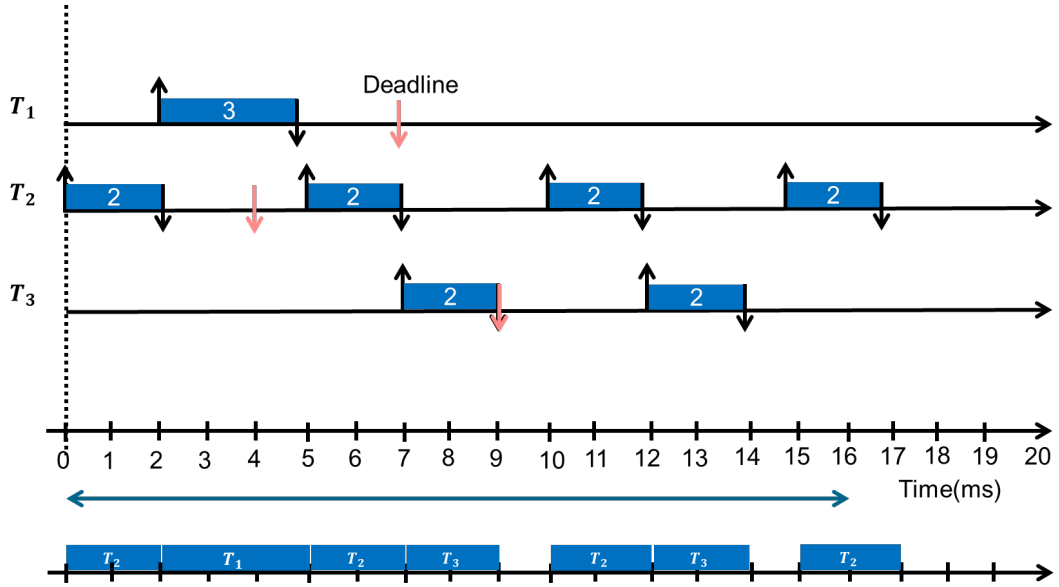


FIGURE 6.1: DM scheduling method, where shows 3 different task represent as T, and their execution time and deadline.

Figure 6.1 depicts how all three tasks could be carried out using DM priority policy. Because task 1 has the highest priority, it is always completed first. Task 2 is then carried out, followed by task 3. This ensures that all three tasks meet their deadlines.

DMS is a very efficient scheduling method used in real-time systems. It is simple to set up and can be used to plan a variety of tasks with varying deadlines and execution times.

Here in this section a detailed explanation of the provided python algorithm designed to assign priorities to tasks based on deadlines.

The algorithm `assign_priority_DM` commences by initializing an empty dictionary named priority meant to store task priorities. It sequentially traverses through the tasks list and allocates priorities to each task based on their respective deadlines. In this process, the dictionary priority is populated, mapping task objects as keys to their corresponding deadline values. Subsequently, to facilitate efficient task organization based on deadlines, the algorithm directs the creation of a sorted list called `sorted_priorities`.

This list is formed by sorting the priority dictionary items in descending order, thereby arranging tasks from those with the highest deadlines to those with the lowest. Then, the algorithm initiates the priority assignment process by setting the variable `priority_val` to represent the total number of tasks, signifying the highest priority value. For each task within the sorted list of priorities, the algorithm assigns the current `priority_val` to the priority attribute of the task. To systematically reduce subsequent task priorities, the `priority_val` is decremented after each assignment, ensuring a progressive decrease in priority values for the following tasks.

This detailed explanation provides insights into the algorithm's workflow, illustrating its role in determining task priorities and enhancing scheduling

strategies for multitasking environments in Algorithm 13.

Algorithm 13: Deadline Monotonic Priority Assignment

Input: A list of tasks $[t_i]$, where $i = 1, 2, \cdots, n$, and each task has a `deadline` attribute.
Output: Assigned priorities for each task.

```
Function assign_priority_DM(tasks, num_cores):
    Create an empty dictionary priorities // Initialize dictionary to store priorities
    // Calculate priority for each task based on its deadline
    for each task t in tasks do
        priorities[t] ← t.deadline
    end for
    Sort priorities in descending order by values
    priority ← length of tasks
    // Assign priorities such that tasks with longer deadlines have lower priorities
    for each task, deadline in sorted priorities do
        task.priority ← priority
        priority ← priority − 1
    end for
return
```

This algorithm demonstrates a fundamental approach to task prioritization based on deadlines, a critical aspect in optimizing task execution and resource allocation within multi-core systems.

6.2.2 DkC Assignment Policy

Andersson and Jonsson conducted a study in 2000 that investigated a priority assignment known as TkC, which was particularly intended for implicit deadline task sets [6]. TkC prioritizes tasks based on the calculated value of $(T_i - kC_i)$, where $'k'$ is a real number determined by the number of cores. The value of $'k'$ is calculated using a particular formula that takes into account the core setup. *TkC* essentially provides a new technique to give priorities to tasks within implicit deadline task groups by using a computed parameter to decide their order in the scheduling process.

To begin, the algorithm initializes a constant m with the value of `num_cores`, thereby representing the number of processor cores available for task execution. A subsequent step involves the calculation of another constant, k, derived from a mathematical formula integrating m, specifically formulated as k,

$$k = \frac{m - 1 + \sqrt{5m^2 - 6m + 1}}{2m} \quad [21] \tag{6.2}$$

This calculated value of k plays a pivotal role in the subsequent priority computation for the tasks. Robert Davis and Alan Burns extend this technique to support tasksets with constrained deadlines by establishing the DkC priority assignment strategy. This strategy determines the order of activities based on the computed value of $(D_i - kC_i)$, where $'k'$ is obtained using Equation 6.2 [21].

The function `assign_priority_DkC` serves the purpose of allocating priorities to tasks within a scheduling context, aiming to optimize task execution based on their respective deadlines and Worst-Case Execution Times (WCET) while considering the available processor cores. It takes two input parameters: tasks, denoting a list of tasks awaiting priority assignment, and `num_cores`, indicating the count of available processor cores.

`assign_priority_DkC` effectively employs mathematical computations and sorting techniques to assign priorities to tasks, a crucial aspect in scheduling for optimizing task execution in systems with multiple processor cores, which fully descriptive pseudo-code is described in Algorithm 14.

Algorithm 14: DkC Priority Assignment

Input: A list of tasks $[t_i]$, $i = 1, 2, \cdots, n$, where each task has `deadline` and `wcet` attributes.

Output: Assigned priorities for each task.

Function `assign_priority_DkC`(*tasks*, *num_cores*)**:**
- $m \leftarrow num\_cores$
- $k \leftarrow \frac{m-1+\sqrt{5m^2-6m+1}}{2m}$ `// Calculate the DkC factor`
- Create an empty dictionary *priorities* `// Initialize dictionary to store priorities`
- **for** *each task t in tasks* **do**
  - $priorities[t] \leftarrow t.deadline - k * t.wcet$
- **end for**
- `// Assign priorities such that tasks with longer DkC have lower priority`
- Sort *priorities* in descending order by values
- $priority \leftarrow$ length (*tasks*)
- **for** *each task in sorted priorities* **do**
  - $task.priority \leftarrow priority$
  - $priority \leftarrow priority - 1$
- **end for**

return

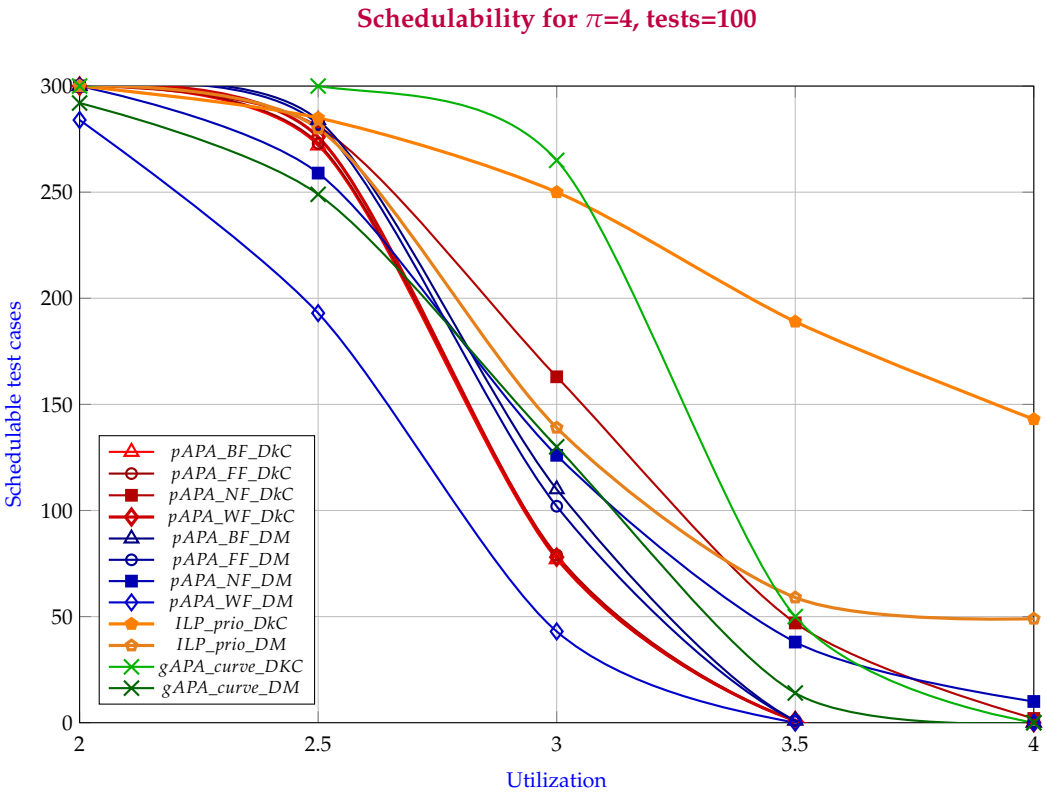
The algorithm proceeds to calculate priorities for each task by iterating through the provided tasks list. For every task encountered, the algorithm computes its priority, involving a formula that considers the task's deadline (t.deadline) and its worst-case execution time (t.wcet). These calculated priorities are then stored in an empty dictionary, priorities, for further processing.

Following the priority calculation, the algorithm sorts the priorities dictionary items in descending order based on the calculated priority values, resulting in a collection called `sort_priorities` containing the tasks sorted by their respective calculated priorities.

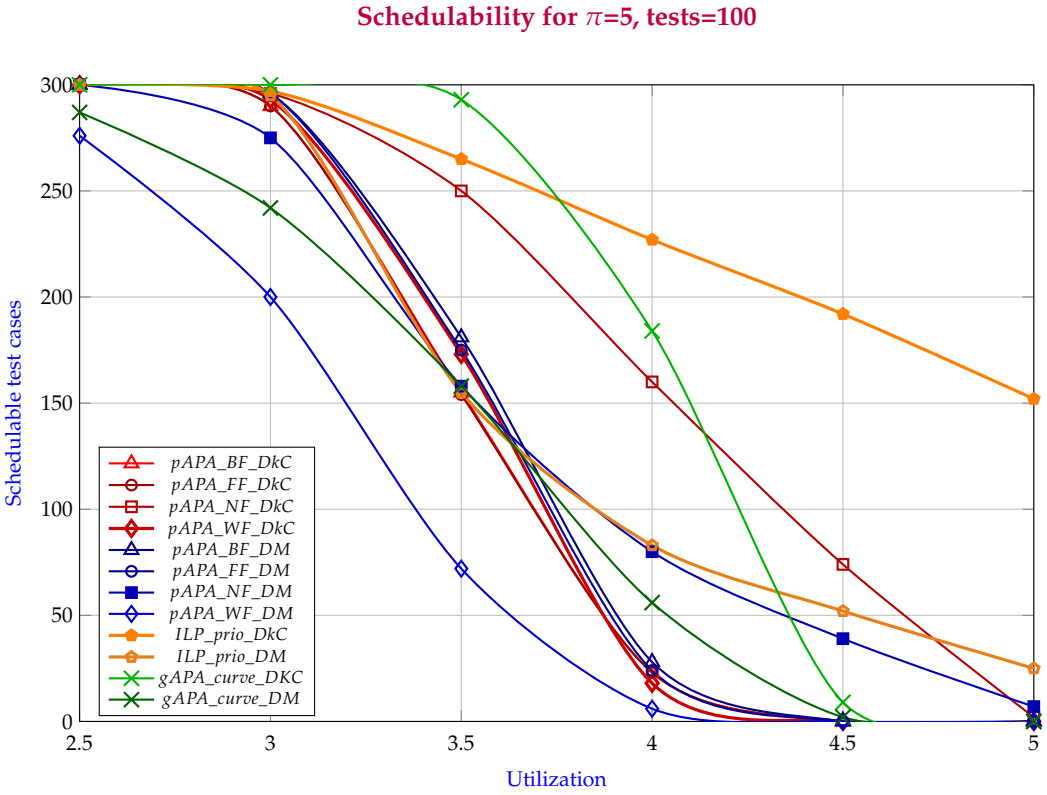
Lastly, the algorithm assigns the calculated priorities to the tasks, starting from the highest priority and decrementing successively for subsequent tasks. This priority assignment process iterates through the sorted tasks, setting the task[0].priorities attribute for each task with the respective priority value and subsequently reducing the priority value for the subsequent tasks.

6.2.3 Results and Interpretation of The Outcome

The graphical representation of the results shows unique patterns in task scheduling performance across different priority assignment techniques and bin-packing algorithms. Notably, *DkC* priority assignment outperforms the *DM* priority assignment for *ILP*, *gAPA*, *pAPA_NF*, *pAPA_WF* approach. The superiority of *DkC* over *DM* is highlighted in the gAPA and ILP approach, where *gAPA_DkC* and *ILP_DkC* performing nearly double to the *gAPA_DM* and *ILP_DM* respectively. Similarly, the pAPA approach that uses next-fit and worst-fit bin packing assignment strategies with DkC priority exceeds to the DM priority. While the pAPA approach with the best-fit and first-fit algorithms, together with DM priority assignment, has a minor advantage over its DkC priority, the difference is not significant.



(A) The comparison of priority assignment algorithms for all scheduling algorithm with 4 cores 100 tests



(B) The comparison of priority assignment algorithms for all scheduling algorithm with 5 cores 100 tests

FIGURE 6.2: Experimental graphs to compare priority assignment algorithms for all scheduling algorithm with different number of cores and 100 tests.

As a result shown in Figure 6.2, it is concluded that the choice of priority assignment has a significant impact on the schedulability of test cases, indicating that the DkC priority assignment method should be used in subsequent experiments to increase the number of schedulable test cases.

6.3 Evaluating Different Bin Packing Algorithms within pAPA and Partitioned Scheduling

This section provide experimental setup overview using different bin packing algorithms, which detailed explanation provided in Chapter 3. The conducted python code is designed to execute an extensive experiment evaluating diverse task scheduling algorithms within a given system configuration. The aim of this experimentation is to comprehensively assess and compare the performance of pAPA and Partitioned scheduling approaches under varying task loads and bin packing algorithms.

The primary objective of this experiment is to analyze the effectiveness and efficiency of different task scheduling algorithms in handling tasks with various utilizations and quantities. The algorithms under investigation include both pAPA and partitioned schedulability tests employing diverse strategies such as FF, NF, BF, and WF.

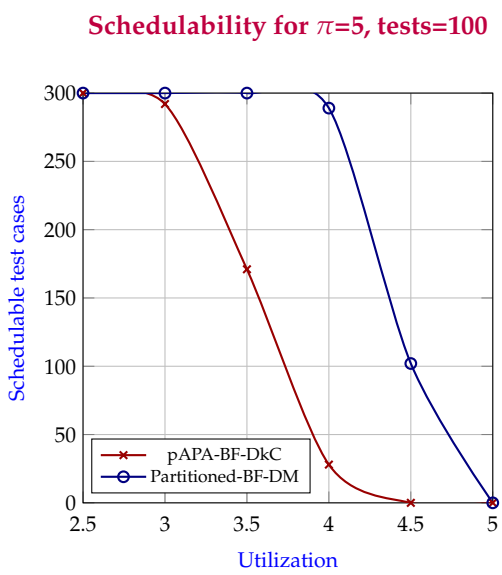
The experiment initiates by generating multiple tasksets characterized by different levels of utilization and varying numbers of tasks. This diversity in taskset creation is vital for assessing algorithmic behavior across a spectrum of system scenarios.

The results of the schedulability tests are properly recorded in the experiment, documenting the success or failure of each algorithm in meeting the scheduling requirements. It builds up the results to provide a comprehensive overview of the performance of the different algorithms under varied system conditions.

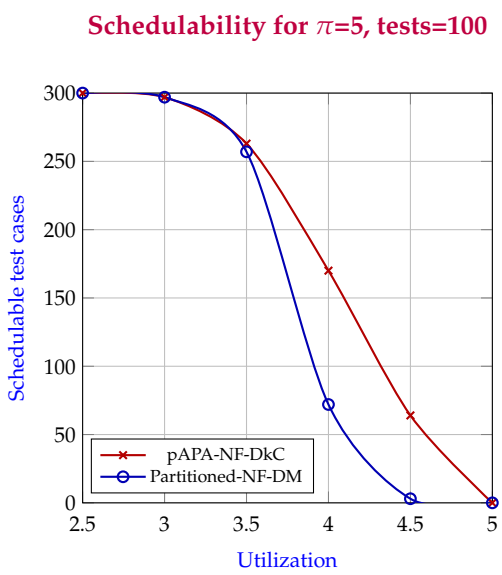
Lastly, the experiment concludes by saving the recorded results and generating visual representations in the form of plots. These visualizations serve as valuable tools for further analysis, enabling the comparison of algorithmic performance across diverse system configurations and highlighting the relative strengths and weaknesses of each scheduling approach.

6.3.1 Results and Interpretation of the Outcome

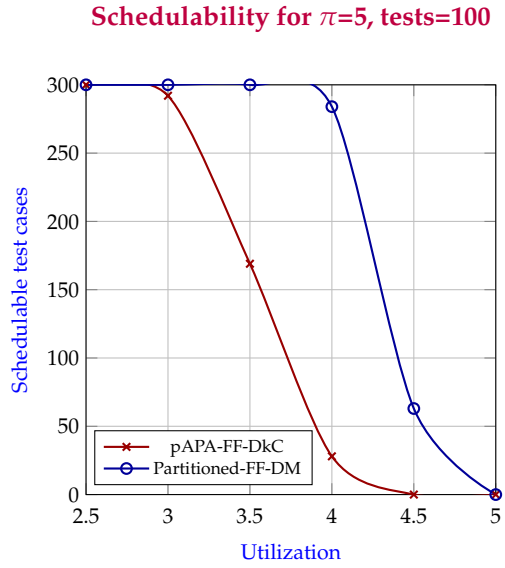
A comparison of the bin packing algorithms best-fit, first-fit, next-fit, and worst-fit for both partitioned and pAPA methods was conducted in an experiment with five cores. In this comparison analysis, exploring the complexities of these two approaches involves examining their various benefits, drawbacks, and effects on system performance.



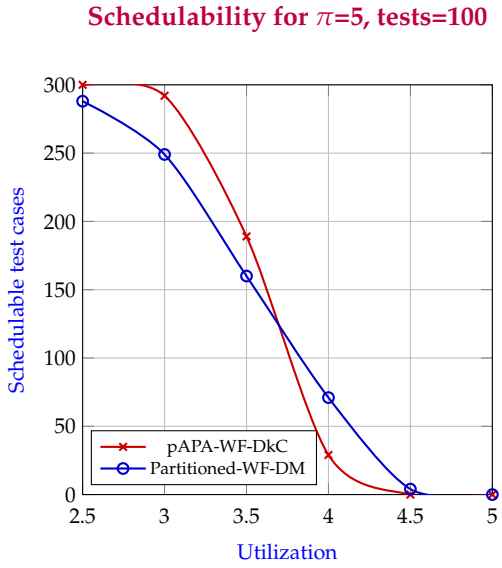
(A) The comparison of best fit algorithm for 5 cores



(B) The comparison of next fit algorithm for 5 cores



(C) The comparison of first fit algorithm for 5 cores



(D) The comparison of worst fit algorithm for 5 cores

FIGURE 6.3: Experimental graphs to comparison between best-fit, first-fit, next-fit and worst-fit bin packing algorithms with 5 cores for pAPA and partitioned.

The partitioned algorithm outperformed the pAPA algorithm when using the best-fit and first-fit task assignment methods as shown in Figures 6.3. However, there was a significant reversal in the next-fit and worst-fit algorithms, with the pAPA outperforming the partitioned approach. The observed occurrence can be related to the distinct features of each bin packing technique. Best-fit and first-fit manage to solve problem with partitioned without merging cores. In contrast, the pAPA algorithm perform better in the next-fit and worst-fit algorithms due to its ability to manage schedule tasks through strategic core merging. Also, the observed weak performance of the *pAPA* algorithm as compared to the typical partitioned approach can be due to some other important factors. Primarily, the beginning of the first iteration in pAPA approach applying the response time analysis as a schedulability test and that methodology stands out as a major contributor to the noted pessimism in pAPA algorithm.

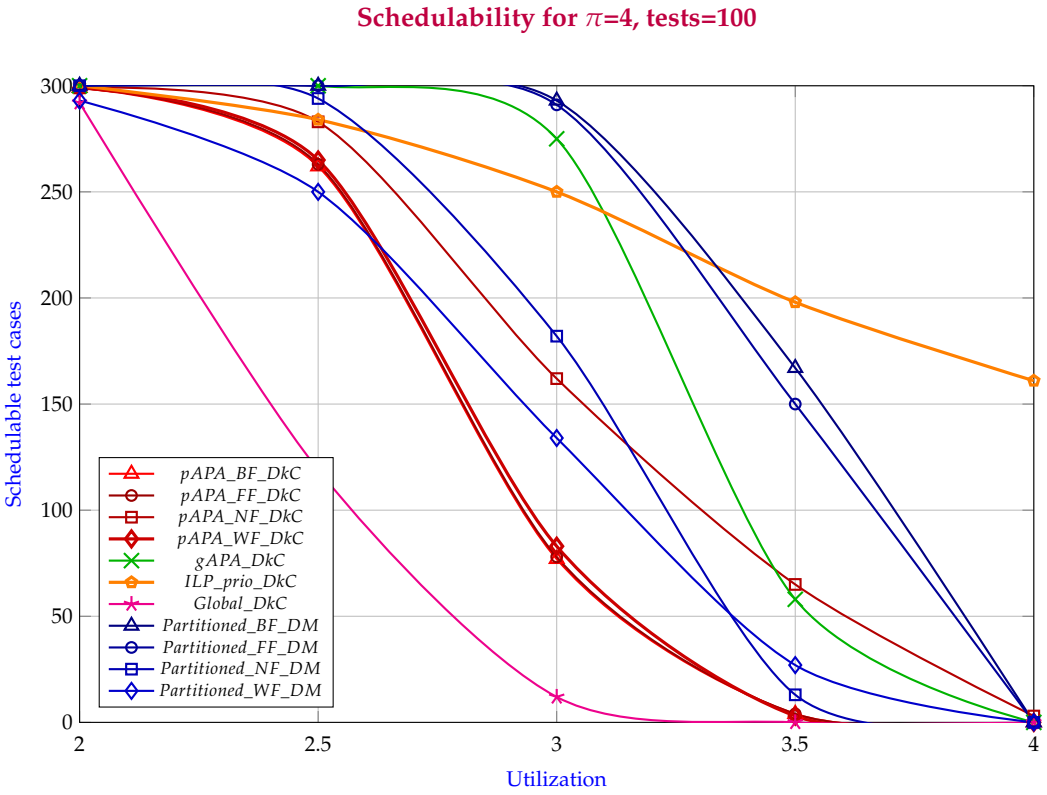
Examining the first row of pAPA, which closely resembles the partitioned technique, provides an informative viewpoint. However, the variation in results is due to the use of different schedulability analysis methods. pAPA uses RTA whereas partitioned approach uses busy window analysis for schedulability test. And the differences in the schedulability test could be directly related to the performance of system, which significantly add to the notably less successful outcomes of the pAPA approach than the partitioned.

In summary, while the theoretical expectations suggest similarity between pAPA and the partitioned approach, the practical results shows more complex reality created by the particular schedulability analysis tests used. Addressing these complexities may allow for a more accurate examination and explanation of the observed performance differences.

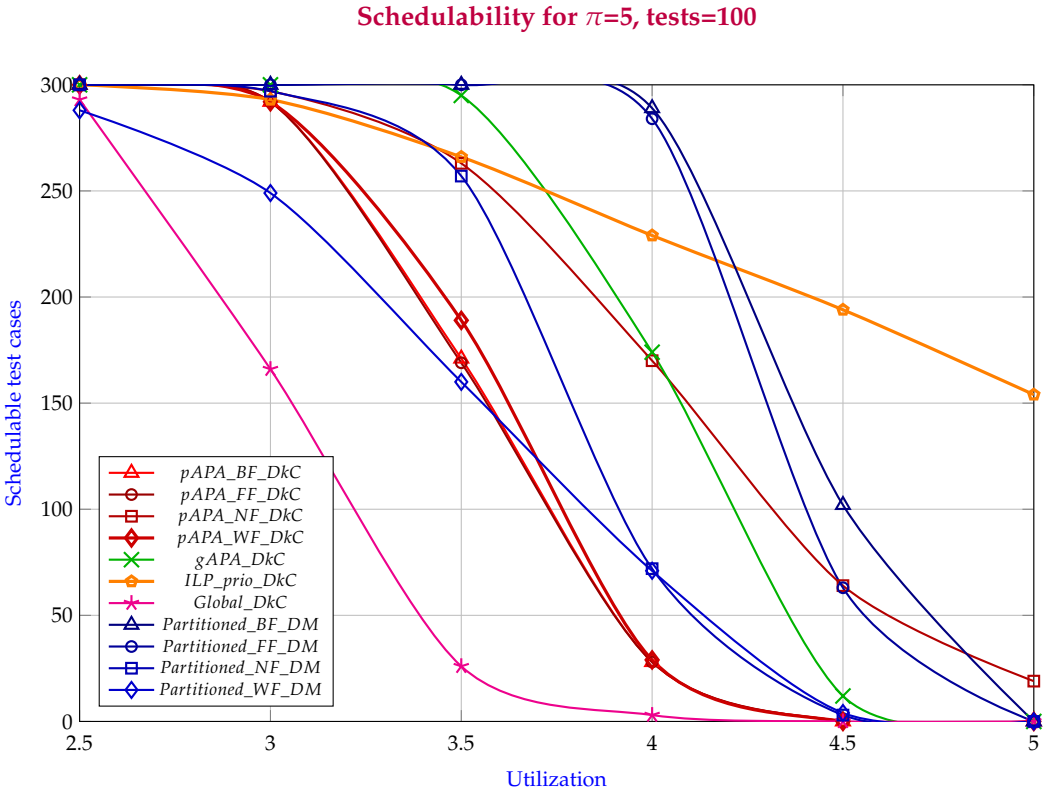
6.4 A Comparative Analysis of the Effectiveness of Different Approaches

Building upon the insights gained from previous experiments, it is clear that using the *DkC* priority assignment strategy consistently results in a greater number of schedulable tasksets for $pAPA, gAPA, ILP$, and *global* scheduling, even with high utilization levels. Notably, partitioned scheduling is an exception, with the DM priority assignment showing greater performance. Drawing on these findings, this experiment conducts a comparative study of the most beneficial methods observed in the previous experiments. This extensive analysis aims to determine the most effective approach for pAPA, gAPA, ILP, Partitioned, and global scheduling, taking into account their performance across different core counts (4, 5, and 6 cores). The consistency of tasksets and test case generation, as described in Section 6.1.

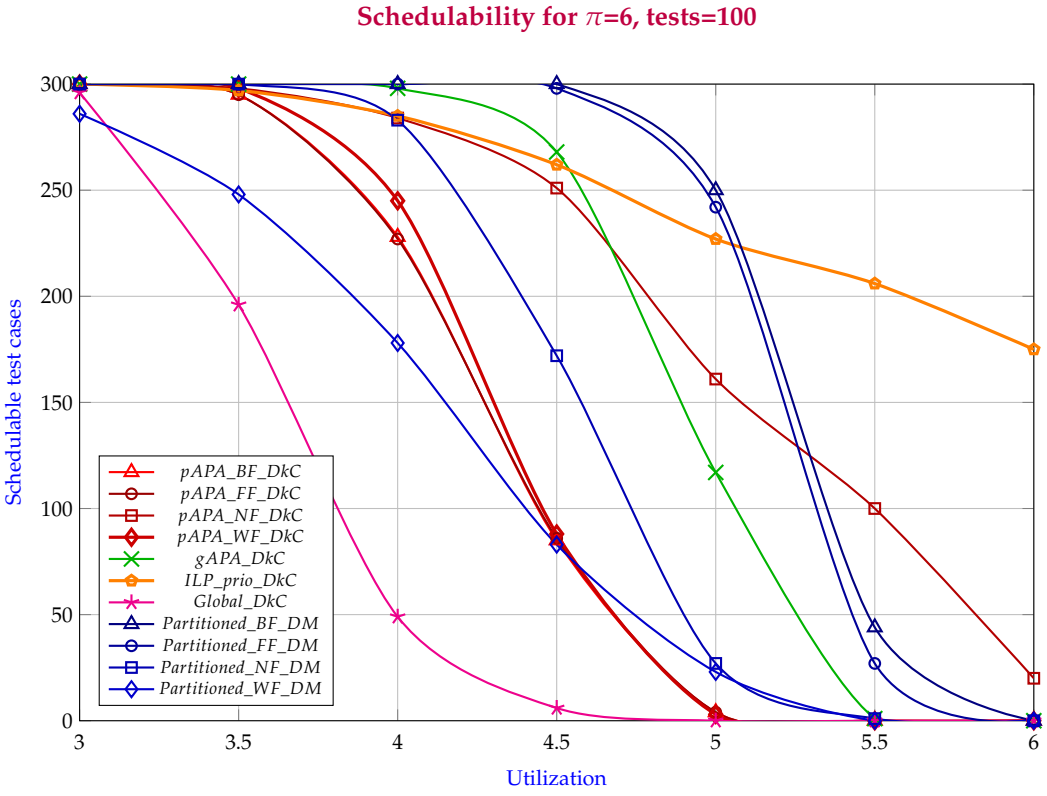
6.4.1 Results and Interpretation of the Outcome



(A) Analysis of the ideal methods with the ILP algorithm for 4 cores and 100 tests



(B) Analysis of the ideal methods with the ILP algorithm for 5 cores and 100 tests



(C) Analysis of the ideal methods with the ILP algorithm for 6 cores and 100 tests.

FIGURE 6.4: Comparison of all best algorithms with ILP algorithm for different cores and 100 tests.

As shown in the Figure 6.4, the ILP_DkC has an exceptional capacity to schedule a greater amount of tasks, even in cases when its utilization value is rather high. This observation highlights how well ILP assigns tasks to cores, demonstrating its ability to maximize task scheduling and get better performance even at higher utilization values. Within the context of partitioned scheduling, which has typically been seen as a standard solution, we intentionally included this technique to highlight possible pessimism within specified constraints and schedulability analysis. According to the theory, APA should be superior, but in fact, partitioning is better. This mismatch highlights an area of pessimism in the analytical approach. Furthermore, APA shows the ability to efficiently integrate global, partitioned, and cluster approaches, highlighting its flexibility to a variety of scheduling contexts.

In summary, the overall goal of this research is to improve the quality of schedulability analysis which is done by developing of novel tasks assignment strategy for the APA. The graphical representations highlight the prevalence of pessimism in the schedulability study, emphasizing the need for modification. Since the results obtained from the experiments highlight the weaknesses in current schedulability assessment, the need for more research is obvious in the attempt of enhanced approaches and strategies that contribute to the ongoing development of schedulability assessment, particularly within the APA framework.

6.5 A Comparative Analysis of the Execution Time of Various Approaches

Expanding on what has been learned from previous strategies, it turns out clear that using ILP for task allocation greatly improves schedulability. However, the practical consequences of online reconfiguration require a more detailed examination of scalability. In the context of online reconfiguration systems, immediate decision-making for schedulability is essential.

The next experiment explores into a comparative evaluation of execution times across different approaches, which is critical for determining their applicability in scenarios demanding rapid online decision-making. This study includes an in-depth analysis of execution time patterns utilizing box plot representations while keeping consistent with the configurations used in the previous experiment.

6.5.1 Results and Interpretation of the Outcome

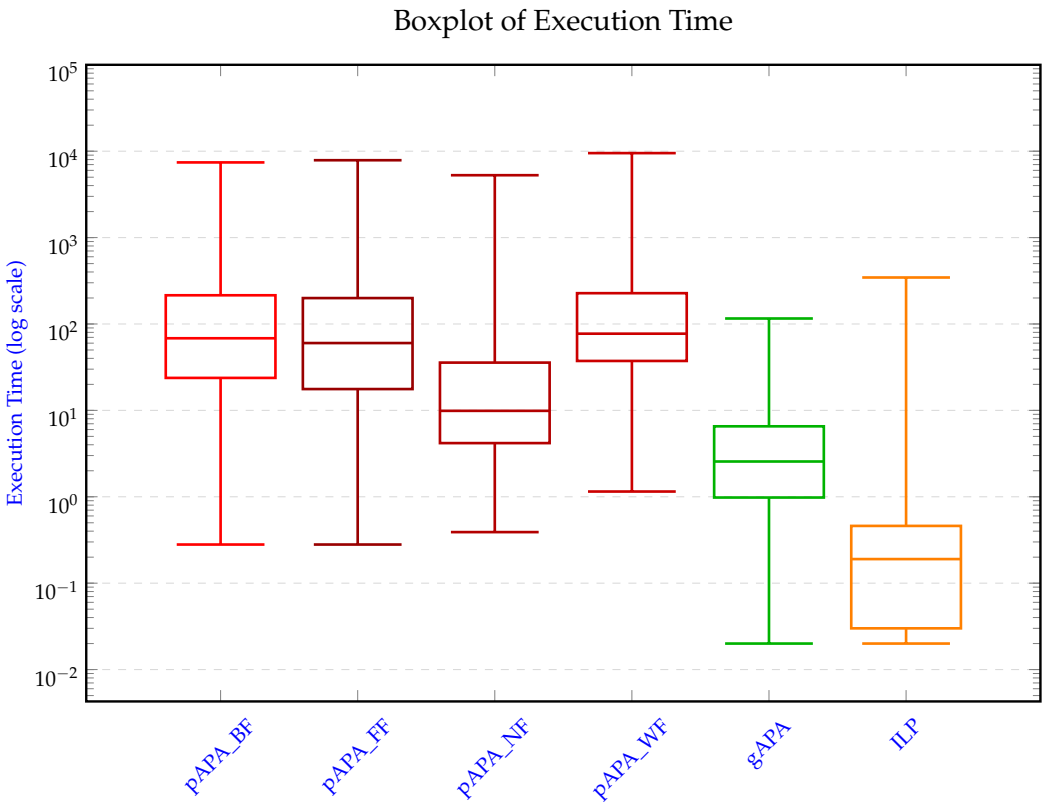


FIGURE 6.5: Execution time for all algorithms considering 4 cores.

The illustrated box plots in Figure 6.5 show important observations about the execution times of various methodologies. Specifically, $pAPA\_BF$, $pAPA\_FF$ and $pAPA\_WF$ have comparable medians, with both indicating longer median duration in comparison to gAPA and ILP. ILP stands out for its clear stability and efficiency, which are demonstrated by the fact that its median execution time is far shorter than that of the other methods. In this context, ILP appears as the dominant algorithm, highlighting its ability to provide fast and consistent performance. The reason it is fast is it uses optimized C code (CPLEX solver) as an its core whereas other approaches are implemented in python without optimization. Therefore, establishing it as a preferable option among the variety of analyzed methodologies.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Finally, this thesis delves into the complex world of real-time task scheduling, specifically the assignment of tasks inside tasksets using ILP for Arbitary Processor Affinity (APA). The analysis begins with an examination of processor affinities, which reveals their critical role in providing isolation and average-case improvements as used by application developers.

The impact of priority assignment algorithms on system performance is demonstrated. The choice of priority assignment has a significant impact on the schedulability of test cases, indicating that the DkC priority assignment method should be used to increase the number of schedulable test cases. The study expands its scope to consider the effects of various bin-packing techniques on partitioned and pAPA scheduling approaches. The variation in both the results indicate that the difference in schedulability analysis can directly impacted to performance of system. And, it has been seen that *pAPA_NF_DkC* can schedule more number of test cases even with higher utilization.

The primary contribution of this thesis, however, is the task allocation for APA scheduling using ILP to enhance multi-core real-time scheduling. The suggested bottom-up-based analysis demonstrates a powerful approach for scheduling a larger number of tasksets. In this regard, the ILP based task assignment reduces the computation time significantly and, based on evaluation results, its accuracy is also great and at the same time achieves low runtime complexity, i.e., it easily scales for problem sizes of up to eight processors.

Additionally, a comparative analysis of the most effective methods observed in preceding experiments has been undertaken. This extensive analysis aims to determine the most effective approach for pAPA, gAPA, ILP, partitioned, and global scheduling, considering their performance across different core counts. Among all effective approaches, it is clear that *ILP_DkC* has a more capacity to schedule a greater number of tasks, and also explores into a comparative evaluation of execution times, to examine the scalability of different approaches. The experiment findings demonstrate that the median execution time of ILP is much shorter than that of the other approaches, demonstrating its scalability and efficiency. As a consequence, ILP stand out as a leading algorithm to deliver quick and reliable results.

In conclusion, this thesis undoubtedly proves APA scheduling usefulness. The unique task assignment technique based on ILP not only improves the field of real-time scheduling, but it also demonstrates the effectiveness of creative ways in tackling complicated issues. The study establishes the way for future advances in the field of multi-core real-time scheduling, offering an excellent foundation for further investigation and application in practical situations.

7.2 Future Work

In regards to future research initiatives in this domain, several interesting alternatives might be explored further. For example, optimizing priority assignment methods is a possible option for improving the assignment of priorities for tasks across various scheduling methods. In the context of ILP, exploring various objective functions seems to be an appealing idea. By changing the optimization criteria, researchers can investigate the possibility for increased performance and efficiency in task assignment methods. More study is being done on the development of improved schedulability analysis for APA scheduling.

Bibliography

[1] Luca Abeni and Tommaso Cucinotta. "EDF Scheduling of Real-Time Tasks on Multiple Cores: Adaptive Partitioning vs. Global Scheduling". In: *SIGAPP Appl. Comput. Rev.* 20.2 (July 2020), pp. 5–18. ISSN: 1559-6915. DOI: 10.1145/3412816.3412817. URL: https://doi.org/10.1145/3412816.3412817.

[2] *About CPU affinity*. Visited on 04.01. 2024. URL: https://enterprise-support.nvidia.com/s/article/what-is-cpu-affinity-x#:~:text=CPU%20affinity%20enables%20binding%20a,each%20one%20on%20different%20core..

[3] Alexander Krutwig AK and Sebastian Huber SH. *RTEMS SMP Status Report*. 2015. URL: http://microelectronics.esa.int/gr740/RTEMS-SMP-StatusReportEmbBrains-2015-10.pdf.

[4] J. H. Anderson and A. Srinivasan. "Pfair scheduling: beyond periodic task systems". In: *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*. 2000, pp. 297–306.

[5] J.H. Anderson, V. Bud, and U.C. Devi. "An EDF-based scheduling algorithm for multiprocessor soft real-time systems". In: *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. 2005, pp. 199–208. DOI: 10.1109/ECRTS.2005.6.

[6] Björn Andersson and Jan Jonsson. "Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition". In: *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications* (2000), pp. 337–346. URL: https://api.semanticscholar.org/CorpusID:1397137.

[7] N.C. Audsley et al. "Hard Real-Time Scheduling: The Deadline-Monotonic Approach". In: *IFAC Proceedings Volumes* 24.2 (1991). IFAC/IFIP Workshop on Real Time Programming, Atlanta, GA, USA, 15-17 May 1991, pp. 127–132. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)51283-5. URL: https://www.sciencedirect.com/science/article/pii/S1474667017512835.

[8] Neil C. Audsley et al. "Applying new scheduling theory to static priority pre-emptive scheduling". In: *Softw. Eng. J.* 8 (1993), pp. 284–292. URL: https://api.semanticscholar.org/CorpusID:310239.

[9] Theodore P. Baker. "A Comparison of Global and Partitioned EDF Schedulability Tests for Multiprocessors TR-051101". In: 2005. URL: https://api.semanticscholar.org/CorpusID:2610522.

[10] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer Publishing Company, Incorporated, 2015. ISBN: 3319086952.

[11] Sanjoy K. Baruah et al. "Proportionate progress: A notion of fairness in resource allocation". In: *Algorithmica* 15 (2005), pp. 600–625.

[12] Waldemar Bauer and et al. "DLR Reusability Flight Experiment ReFEx". In: *Acta Astronautica* (Nov. 2019). URL: https://elib.dlr.de/132256/.

[13] G. Bernat, A. Burns, and A. Liamosi. "Weakly hard real-time systems". In: *IEEE Transactions on Computers* 50.4 (2001), pp. 308–321. DOI: 10.1109/12.919277.

[14] Marko Bertogna and Michele Cirinei. "Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 2007, pp. 149–160. DOI: 10.1109/RTSS.2007.31.

[15] *Bin Packing problem*. Visited on 23.01. 2024. URL: https://iq.opengenus.org/bin-packing-problem/.

[16] Enrico Bini. "Measuring the Performance of Schedulability Tests". In: *Real-Time Systems* 30 (May 2005), pp. 129–154. DOI: 10.1007/s11241-005-0507-9.

[17] Giorgio C. Buttazzo. "Rate Monotonic vs. EDF: Judgment Day". In: *Embedded Software*. Ed. by Rajeev Alur and Insup Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 67–83. ISBN: 978-3-540-45212-6.

[18] John M. Calandrino, James H. Anderson, and Dan P. Baumberger. "A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms". In: *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*. 2007, pp. 247–258. DOI: 10.1109/ECRTS.2007.81.

[19] John Carpenter et al. "A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms". In: (June 2003).

[20] Robert Davis. "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems". In: *ACM Comput. Surv.* 43 (Oct. 2011). DOI: 10.1145/1978802.1978814.

[21] Robert I. Davis and Alan Burns. "Improved Priority Assignment for Global Fixed Priority Pre-emptive Scheduling in Multiprocessor Real-Time Systems". In: *IEEE International Conference on Real-Time Systems (RTS)*. Real-Time Systems Research Group, Department of Computer Science, University of York. York, UK.

[22] Michael L. Dertouzos. "Control Robotics: The Procedural Control of Physical Processes". In: *IFIP Congress*. 1974. URL: https://api.semanticscholar.org/CorpusID:38135638.

[23] Juraj Feljan, Jan Carlson, and Tiberiu Seceleanu. "Towards a Model-Based Approach for Allocating Tasks to Multicore Processors". In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. 2012, pp. 117–124. DOI: 10.1109/SEAA.2012.56.

[24] N. Fisher, S. Baruah, and T. P. Baker. "The partitioned scheduling of sporadic tasks according to static-priorities". In: *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*. July 2006, 10 pp.–127. DOI: 10.1109/ECRTS.2006.30.

[25] A. Foong, J. Fung, and D. Newell. "An in-depth analysis of the impact of processor affinity on network performance". In: *Proceedings. 2004 12th IEEE International Conference on Networks (ICON 2004) (IEEE Cat. No.04EX955)*. Vol. 1. 2004, 244–250 vol.1. DOI: 10.1109/ICON.2004.1409136.

[26] M. Gonzales Harbour and J.C. Palencia. "Response time analysis for tasks scheduled under EDF within fixed priorities". In: *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*. 2003, pp. 200–209. DOI: 10.1109/REAL.2003.1253267.

[27] Arpan Gujarati, Felipe Cerqueira, and Björn B. Brandenburg. "Outstanding Paper Award: Schedulability Analysis of the Linux Push and Pull Scheduler with Arbitrary Processor Affinities". In: *2013 25th Euromicro Conference on Real-Time Systems*. 2013, pp. 69–79. DOI: 10.1109/ECRTS.2013.18.

[28] Zain A. H. Hammadeh. "Deadline Miss Models for Temporarily Overloaded Systems". PhD thesis. Sept. 2019. DOI: 10.24355/dbbs.084-201909020857-0. URL: https://leopard.tu-braunschweig.de/receive/dbbs_mods_00066886.

[29] Zain A. H. Hammadeh and Rolf Ernst. "Weakly-Hard Real-Time Guarantees for Weighted Round-Robin Scheduling of Real-Time Messages". In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. Sept. 2018, pp. 384–391. DOI: 10.1109/ETFA.2018.8502621.

[30] Zain A. H. Hammadeh, Sophie Quinton, and Rolf Ernst. "Weakly-Hard Real-Time Guarantees for Earliest Deadline First Scheduling of Independent Tasks". In: *ACM Trans. Embed. Comput. Syst.* 18.6 (Dec. 2019). ISSN: 1539-9087. DOI: 10.1145/3356865. URL: https://doi.org/10.1145/3356865.

[31] Jian-Jun Han et al. "Blocking-Aware Partitioned Real-Time Scheduling for Uniform Heterogeneous Multicore Platforms". In: *ACM Trans. Embed. Comput. Syst.* 19.1 (Feb. 2020). ISSN: 1539-9087. DOI: 10.1145/3366683. URL: https://doi.org/10.1145/3366683.

[32] *Importance of Worst-Case Execution Time*. Visited on 04.01. 2024. URL: https://www.rapitasystems.com/worst-case-execution-time.

[33] Hye-Churn Jang and Hyun-Wook Jin. "MiAMI: Multi-core Aware Processor Affinity for TCP/IP over Multiple Network Interfaces". In: *2009 17th IEEE Symposium on High Performance Interconnects*. 2009, pp. 73–82. DOI: 10.1109/HOTI.2009.19.

[34] J. Kang and D. G. Waddington. "Load Balancing Aware Real-Time Task Partitioning in Multicore Systems". In: *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2012, pp. 404–407. DOI: 10.1109/RTCSA.2012.71.

[35] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. "Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors". In: *2009 21st Euromicro Conference on Real-Time Systems*. July 2009, pp. 239–248. DOI: 10.1109/ECRTS.2009.33.

[36] J.P. Lehoczky. "Fixed priority scheduling of periodic task sets with arbitrary deadlines". In: *[1990] Proceedings 11th Real-Time Systems Symposium*. 1990, pp. 201–209. DOI: 10.1109/REAL.1990.128748.

[37] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743. URL: https://doi.org/10.1145/321738.321743.

[38] E.P. Markatos and T.J. LeBlanc. "Using processor affinity in loop scheduling on shared-memory multiprocessors". In: *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*. 1992, pp. 104–113. DOI: 10.1109/SUPERC.1992.236705.

[39] E.P. Markatos and T.J. LeBlanc. "Using processor affinity in loop scheduling on shared-memory multiprocessors". In: *IEEE Transactions on Parallel and Distributed Systems* 5.4 (1994), pp. 379–400. DOI: 10.1109/71.273046.

[40] G. Nelissen et al. "U-EDF: An Unfair But Optimal Multiprocessor Scheduling Algorithm for Sporadic Tasks". In: *2012 24th Euromicro Conference on Real-Time Systems*. 2012, pp. 13–23.

[41] Farhang Nemati. "Partitioned Scheduling of Real-Time Tasks on Multi-core Platforms". PhD thesis. Mälardalen University, 2010.

[42] Dionisio de Niz and Raj Rajkumar. "Partitioning bin-packing algorithms for distributed real-time systems". In: *IJES* 2 (Jan. 2006), pp. 196–208. DOI: 10.1504/IJES.2006.014855.

[43] J. C. Palencia and M. González Harbour. "Response Time Analysis of EDF Distributed Real-Time Systems". In: *J. Embedded Comput.* 1.2 (Apr. 2005), pp. 225–237. ISSN: 1740-4460.

[44] Dheeraj Reddy et al. "Bridging functional heterogeneity in multicore architectures". In: *Operating Systems Review* 45 (Feb. 2011), pp. 21–33. DOI: 10.1145/1945023.1945028.

[45] RTEMS. *RTEMS Real Time Operating System (RTOS)*. URL: https://www.rtems.org/.

[46] James D. Salehi, James F. Kurose, and Donald F. Towsley. "Further results in affinity-based scheduling of parallel networking". In: 1995. URL: https://api.semanticscholar.org/CorpusID:15459500.

[47] René Schwarz and et al. "Overview of Flight Guidance, Navigation, and Control for the DLR Reusability Flight Experiment (ReFEx)". In: *8th European Conference for Aeronautics and Space Sciences (EUCASS)*. July 2019. URL: https://elib.dlr.de/129086/.

[48] Insik Shin, Arvind Easwaran, and Insup Lee. "Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors". In: *2008 Euromicro Conference on Real-Time Systems*. 2008, pp. 181–190. DOI: 10.1109/ECRTS.2008.28.

[49] Wenbo Xu et al. "Improved Deadline Miss Models for Real-Time Systems Using Typical Worst-Case Analysis". In: *2015 27th Euromicro Conference on Real-Time Systems*. July 2015, pp. 247–256. DOI: 10.1109/ECRTS.2015.29.

[50] Ying Yi et al. "An ILP formulation for task mapping and scheduling on multi-core architectures". In: *2009 Design, Automation & Test in Europe Conference & Exhibition*. 2009, pp. 33–38. DOI: 10.1109/DATE.2009.5090629.