Delft University of Technology
Master of Science Thesis in Embedded Systems

Low-Overhead Non-Preemptive Scheduling of Real-Time Tasks upon Multiprocessor Platforms

Eghonghon-aye Eigbe
Supervised by: Dr. Mitra Nasri

TU Delft Delft University of Technology

Low-Overhead Non-Preemptive Scheduling of Real-Time Tasks upon Multiprocessor Platforms

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Eghonghon-aye Eigbe
Supervised by Dr. Mitra Nasri

18th July, 2020

Author
Eghonghon-aye Eigbe
Title
Low-Overhead Non-Preemptive Scheduling of Real-Time Tasks upon Multiprocessor Platforms
MSc Presentation Date
27th July, 2020

Graduation Committee

Dr.ir. Fernando Kuipers	Delft University of Technology
Dr. Mitra Nasri	Delft University of Technology and Eindhoven University of Technology
Dr. Lydia Chen	Delft University of Technology

Parts of the work presented in this thesis was presented at the sCalable And PrecIse Timing AnaLysis for multicore platforms (CAPITAL) Workshop in February, 2020.

Abstract

While multiprocessor platforms have been widely adopted by the embedded systems industry in the past couple of years, there are still fundamental challenges about their timing predictability for applications with real-time timing constraints. The common-off-the-shelf (COTS) multiprocessor platforms typically use complex hardware components, interconnects and multi-level caches which are designed to deliver higher average-case performance. However, these features negatively impact the worst-case performance as they increase the interference of tasks on shared hardware resources. One effective software-based solution to counteract these issues is to use the non-preemptive execution model.

Despite its positive impact on timing predictability, non-preemptive execution causes potential *blocking problem* which can decrease the ability to guarantee all timing constraints of the system. It is also known that scheduling non-preemptive periodic tasks on multiprocessor platforms is an NP-hard problem.

In this thesis, we focus on non-preemptive execution of sequential as well as parallel real-time tasks upon multiprocessor platforms and *investigate, extend, and improve the state of the art on global, partitioned, and semi-partitioned scheduling approaches* for the problem.

We provide the first necessary test for partition-ability, i.e., a test that can determine whether a given task set cannot be partitioned on a given number of cores regardless of the partitioning policy. This test allows us to quantify the pessimism of the existing partitioning heuristics as well as obtain the limits of partitioned scheduling. We further introduce the first non-work-conserving global scheduling policy and show that despite the fact that it improves over the existing global scheduling policies, it is not as effective as the partitioned scheduling strategies. We extend a sustainable scheduling algorithm designed for uni-processor platforms to multiprocessor ones to improve the performance of partitioning heuristics. A sustainable scheduling algorithm does not have timing anomalies and hence it is easier to analyze and can have better scheduling results.

Furthermore, we introduce the first semi-partitioned non-preemptive scheduling solution for multiprocessor platforms. Our solution is able to schedule some of the task sets for which it is impossible to find a partitioning solution. Finally, we compare the overheads and memory consumption of various scheduling approaches (including ours) on a bare-metal multiprocessor hardware platform, i.e., a 4 processor Raspberry Pi board. We show that our sustainable scheduler has a very low overhead while it out-performs other solutions in terms of schedulability.

Preface

Achebe said that being an intellectual does not mean knowing about intellectual issues; it means taking pleasure in them. It is my hope that this work is intellectual enough to be pleasurable.

I am especially grateful to Mitra Nasri, for supervising this work. To Junaid Aslam, for answering all my questions in record time and providing a DAG generator. To Leon de Boer, Rob Davis and all the other members of the real-time systems community. Thank you for making not only your tools open source but also your minds and time.

I also want to give special thanks to *booking.com* for sponsoring this degree and making this possible for me at all in the first place. And finally, to the people I owe my existence - Asuelime, Beatie, Shosho, Koko, Ehi, Tikam, Rilwan, Ismail, Chika, Oyinkan, and Hassan. I am shielded always by your love. I no dey ever loseguard and na una make.

Eghonghon-aye Eigbe

Delft, The Netherlands
18th July 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multiprocessor platforms increase computation power by giving us opportunities for parallelism, in other words, allowing multiple functionalities or tasks to be carried out simultaneously. Such systems have been widely adopted in the embedded systems industry for a variety of applications ranging from digital signal processing [44] to industrial control [28]. A survey by VDC Research showed that 40% of industrial systems, including those with real-time timing constraints, are using multiprocessor platforms [3].

The common-off-the-shelf (COTS) multiprocessor platforms typically use complex hardware components, interconnects and multi-level caches. While these features improve the average-case performance, they negatively impact the worst-case performance as they increase the interference of tasks on shared hardware resources such as caches, busses, memory banks, etc. Such an impact, however, can be reduced drastically when a more time-predictable execution model such as non-preemptive execution is applied on the system.

1.1 Problem Definition and Research Questions

While non-preemptive execution increases the timing predictability of the system, it results in potential *blocking* times which can then increase the *response-time* of higher-priority tasks. Jeffay et al. [23] have shown that the problem of scheduling a set of real-time non-preemptive periodic tasks upon a uni-processor platform is NP-hard. This holds true also for multiprocessor platforms when tasks are partitioned between processors and run exclusively on the processor they are assigned to [10].

In this work, we focus on the problem of scheduling a set of non-preemptive periodic sequential or parallel tasks upon a multiprocessor platform. We *investigate, extend, and advance the state of the art* in each of the following major scheduling approaches: *global, partitioned, and semi-partitioned scheduling.* The difference between these scheduling approaches comes from how they consider task migration between processors [16]: in partitioned scheduling, no migration is allowed, hence, tasks run exclusively on the processor they are assigned to. In semi-partitioned scheduling, most tasks do not migrate but some do. And in global scheduling, there is no explicit assignment between tasks and processors. In the following, we describe some of the open problems w.r.t. the state of the art of each of these scheduling approaches.

Partitioned scheduling. Most existing heuristics for task partitioning rely on a fitness test that determines whether or not a new task can be added to an existing partition. This fitness test is usually a schedulability test for a given scheduling algorithm that will be used to schedule the tasks on the processor

they are assigned to. This test determines whether the scheduling algorithm can guarantee the timing constraints of the tasks during the lifetime of the system. As a result, *the success of a partitioning heuristic will depend on the success of the underlying scheduling algorithm as well as the accuracy (lack of pessimism) of the schedulability test used with it.*

The state of the art has extensively explored policies such as non-preemptive fixed-priority (NP-FP) or the earliest-deadline-first (NP-EDF) scheduling [20, 30, 19]. However, these policies are subject to *scheduling anomalies* and hence are not *sustainable* [6]. A scheduling algorithm is called sustainable if and only if a task set that can be successfully scheduled by the algorithm remains schedulable if the timing constrains under which it was tested become more relaxed [6], e.g., a task has a smaller execution time at runtime. Unfortunately, most well-studied scheduling policies such as NP-FP and NP-EDF are not sustainable w.r.t. execution time variation and those that are sustainable, such as the first-in-first-out (FIFO) policy are known to have a poor schedulability.

Hence, our first research goal is to design and implement sustainable scheduling policies and investigate their impact on the success of partitioning heuristics. For this aim, we try to find an answer for the following research question:

> **RQ1.** What is the impact of a sustainable scheduling policy on the success of partitioning heuristics when applied to non-preemptive sequential and/or parallel tasks?

The second research problem we address is about quantifying the pessimism of the existing partitioning heuristics. Currently, apart from brute-force exploration methods with exponential computational complexity, there is no known optimal partitioning policy for periodic tasks scheduled on multiprocessor platforms. This means that there is no method with a reasonable runtime that is guaranteed to find a successful partition for a non-preemptive task set if such a partition exists. This leaves the following question open:

> **RQ2.** How can the pessimism of the existing partitioning heuristics be quantified given the lack of an optimal partitioning algorithm?

Semi-partitioned scheduling. While the literature has extensively studied semi-partitioned scheduling for preemptive tasks [2, 11], there is currently no solution for non-preemptive tasks. This brings us to the third research question:

> **RQ3.** Can a semi-partitioned scheduling solution be designed to schedule task sets that cannot be partitioned?

Global scheduling. Non-preemptive execution may cause long blocking from lower-priority tasks on higher-priority ones, and hence has typically a lower success rate compared to preemptive scheduling. To combat this, recent work [37, 36, 34] have introduced *non-work-conserving* policies, i.e., policies that allow the resource to remain idle even if there are tasks in the ready queue. To leave the processor idle, these policies look at future workloads in the system to make scheduling decisions. Although these studies have shown promising improvements in schedulability, they have not yet been adopted for multiprocessor platforms. This leads us to the next research question:

> **RQ4.** How viable are non-work-conserving policies for global scheduling in terms of schedulability and overheads?

Runtime overheads. Scheduling theory sometimes assumes that decisions are made instantaneously but in practical implementations, schedulers incur

runtime overheads. When these overheads are too large, they can decrease the schedulability of a task set. Additionally, embedded systems are often memory constrained and hence there is a need for scheduling solutions with small memory footprints. This motivates our final research question:

> **RQ5.** What are the overheads (in terms of runtime and memory usage) of global, partitioned, and semi-partitioned scheduling on multiprocessor platforms?

1.2 Contributions

In order to address the first research question **(RQ1)**, we introduce the concept of sustainable partitioning to improve the success rate of partitioning heuristics, where instead of typical non-sustainable scheduling policies to schedule a task set in each partition, we use a sustainable scheduling policy (i.e., FIFO-OT [35]). In addition, we propose a new partitioning heuristic which maps the task set to a graph and partition them by finding the largest cliques in an iterative way. Moreover, we extend our solution to parallel tasks by designing a **task-decomposition strategy** to break down a given parallel task into a set of sequential ones and then apply our sustainable partitioning solution on the resulting tasks set.

We present **the first necessary test for partition-ability** of a task set on a given number of processors. Our test determines if a task set cannot be partitioned on a given number of processors with *any* partitioning policy. To answer **(RQ2)**, we measure the performance of the existing partitioning heuristics against our necessary test.

For the third research question (**RQ3**), we propose an offline **semi-partitioned scheduling solution** for sequential tasks called *liquid-path scheduling* and complement it with an online sustainable semi-partitioned scheduling policy. Our online policy extends the FIFO scheduling with *offset tuning* (FIFO-OT) [35] from the uni-processor to multiprocessor platforms.

To address the fourth research question **(RQ4)**, we introduce **the first non-work-conserving global scheduling policy** for non-preemptive sequential tasks on multiprocessor platforms. Our policy is an extension of one of the most efficient non-work-conserving scheduling policies on uni-processor platforms, called CW-EDF [36].

Finally, to answer **(RQ5)**, we implement various scheduling solutions including the ones we propose on a bare-metal Raspberry PI board with 4 processors and measure their runtime overheads and memory consumption.

1.3 Organization

Chapter 2 provides important concepts forming the background of this thesis, and presents the system model and notations used in this thesis. Chapter 3 introduces the state of the art. Chapters 4 and 5 present solutions for sequential and parallel workloads respectively. Experimental results are presented in chapter 7, after which conclusions and future recommendations are presented in chapter 8.

Chapter 2

System Model and Background

This chapter introduces concepts integral to the formulation of the problem and understanding of the solutions presented in later chapters. We begin in section 2.1 with definitions of the scope of this work and introduce our system model in section 2.2. Next, in section 2.3, we present necessary background information to aid understanding of the rest of the thesis.

2.1 Definitions

A real-time system is one whose correctness depends on both the timing and logical correctness of results. A common misconception is that a real-time system is one which responds as quickly as possible, however, the requirement is that it complete its operation within a stipulated time frame irrespective of the length of this time frame. In such systems, timing constrains are classified as hard, firm or soft in order of the consequences suffered from their violation. Hard real-time systems require that all deadlines be respected as violations are catastrophic. A common example is safety critical systems in avionic or automotive industries. In firm systems, the results lose their utility if they are produced outside of timing constraints. The consequence of timing failures in soft real-time systems is typically performance degradation. We concern ourselves with hard real-time systems.

Some important concepts in modelling a real-time system are as follows:

- **Task** - A task, often synonymous with a process, is a piece of computation or functionality executed by a processor. In a real time system, the relative deadline of a task is the time from its entry to the system, at which it must be completed to guarantee correctness. Tasks generate an infinite number of instances (called **jobs**) throughput the lifetime of the system and releases are characterised by their periodicity. Tasks can be periodic - with jobs arriving at a constant frequency, sporadic - with jobs arriving randomly with a minimum inter-arrival time or aperiodic - with jobs arriving randomly [12].
- **Feasibility** - A feasible schedule is one which respects all timing constrains. A task set is said to be feasible if and only if there exists a feasible schedule for it, i.e., an infeasible task set is one that can never be successfully scheduled.
- **Schedulability***(with a scheduling algorithm A)* - A task set is schedulable with a given algorithm A when the algorithm results in a feasible schedule for any execution scenario that can be generated by the task set. Execution scenarios can vary at runtime resulting in different schedules. An example is the variation in execution time. A task set that is schedulable with algorithm A must remain schedulable for all possible execution times of the task set.

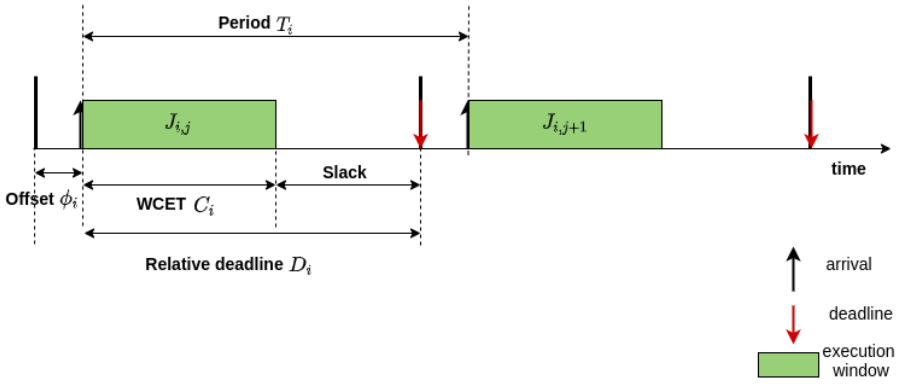


Figure 2.1: **Sequential task timing properties**

- **Precedence** - Precedence as a concept becomes important when tasks cannot be carried out in arbitrary order but instead follow a sequence imposed by data dependency or simply the nature of the applications to which they belong.
- **Slack** - This is the maximum delay that a job can experience after its release and still meet its timing requirements.

2.2 System Model

Generally, multiprocessor platforms give us the chance to exploit parallelism. This can either be *inter-task parallelism* where different tasks runs *sequentially* on a single processor or *intra-task parallelism* where a task can have components or threads that run simultaneously on multiple processors. The option available to us depends on the kind of tasks we have in the system. A task is either able to carry out its operations simultaneously, i.e., it is a parallel task or is constrained to perform computations in a sequential order.

Sequential Task Model

A task set is represented as $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ where n is the number of tasks in the set. $\tau_i = (C_i, T_i, D_i, \phi_i)$, where C_i is the worst-case execution time (WCET), T_i is the period, $D_i \leq T_i$ is the relative deadline of the task τ_i and ϕ_i is the task offset. Other task properties that can be derived from the model are task utilisation, U_i, computed as C_i/T_i. Unless otherwise stated, the offset of a periodic task is assumed to be zero, i.e., it arrives exactly at multiples of its period.

Given a set of tasks, the hyperperiod denoted by H, is the least common multiple (LCM) of all the periods in the task set. In a periodic task set, all events repeat in each hyperperiod. $J_i = \{J_{i,1}, J_{i,2}, ..., J_{i,n_i}\}$ represents the set of jobs in a hyperperiod H for τ_i where n_i is the number of jobs of τ_i in H. Each job $J_{i,j}$ has a release time $r_{i,j} = (j-1) * T_i + \phi_i$ and absolute deadline $d_{i,j}$. The analysis presented in this thesis assumes task deadlines are equal to their periods. Figure 2.1 shows the described timing properties graphically represented.

Parallel Task Model

There are multiple ways to represent a parallel task but in this thesis, each task, $\tau_i = (C_i, T_i, D_i, V_i, E_i, P_i)$, is represented as a directed acyclic graph (DAG). A DAG is a collection of nodes, V_i, also referred to as sub-tasks. Each sub-task

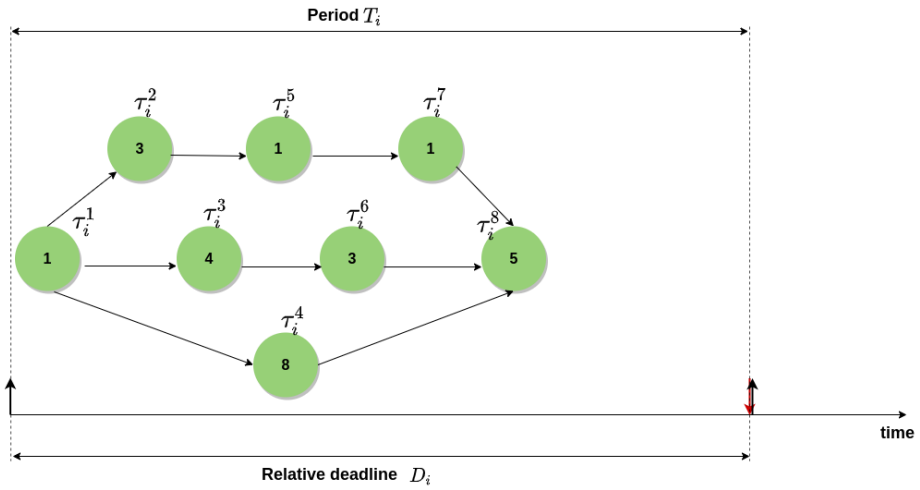


Figure 2.2: **An example of a parallel task modelled as a DAG**

$\tau_i^j | 1 \leq j \leq n_i \in V_i$ is characterised by a worst case execution time C_i^j where n_i is the number of nodes in the DAG. E_i represents a set of directed edges which determine the precedence constrains and subsequently the execution flow of the whole task.

The global timing properties of the DAG resemble those of a sequential task in that C_i is the WCET of the task computed as the sum of all the WCETs of the nodes. It represents the worst case execution time should the DAG be executed sequentially. T_i is the period and D_i is the relative deadline of the task. Another important timing property is P_i, which is the length of the critical path of the task where the critical path is the longest chain of sequential computation as necessitated by precedence constrains in the DAG. The definitions of utilisation and jobs hold the same as in a sequential task.

Figure 2.2 shows an example of a parallel task modelled as a DAG with 8 sub-tasks. Arrows represent precedence constraints, e.g. τ_i^8 can only be run after τ_i^4,τ_i^6 and τ_i^7 have all been completed. This task has a sequential computation time C_i of 26 units and a critical path τ_i^1,τ_i^4,τ_i^8 with length P_i of 14 units.

$J_i = \{J_{i,1}, J_{i,2}, ..., J_{i,k_i}\}$ represents the set of jobs in a hyperperiod H for τ_i where k_i is the number of jobs of τ_i in H. The qth sub-task of a job of the DAG is represented as $J_{i,j}^q$

Processor Model

The system is assumed to be a homogeneous multiprocessor platform composed of m identical unit-speed processors $p_1, ..., p_m$.

2.3 Background

2.3.1 Execution Models

Execution of processes or tasks can either be preemptive, non-preemptive or limited-preemptive depending on whether a process can be interrupted by another or not.

- **Preemptive execution** allows interruptions of running jobs by higher priority jobs. This is implemented by first saving the context of the running

task, switching to the new task, and then restoring the state of the system to continue the interrupted job after.

- **Non-preemptive execution** does not allow interruptions and as such a running task holds onto computing resources assigned to it until its completion. This means that a higher priority job can be delayed by low-priority jobs. This effect is known as *blocking*.
- **Limited-preemptive execution** is a scenario in which preemptions are allowed but only at specific points in the execution of the running task. These points are typically predetermined by application designers.

2.3.2 Schedulers

In computing systems, a scheduler is the entity responsible for allocation of computing resources to processes. They generally control when a process has access to a computing resource.

Scheduling Algorithms

Schedulers work with scheduling algorithms to make decisions and many classes and categories of these algorithms exist. Scheduling algorithms can be classified based on priority assignment is as follows:

- **Fixed-priority (FP) assignment**: Here, each task is assigned a fixed priority and all jobs of this task inherit the same priority which remains unchanged throughout the lifetime of the system. Such algorithms are sometimes referred to as fixed-priority (FP) algorithms and some common priority assignment techniques are rate monotonic (RM) and deadline monotonic (DM) where priorities are assigned in ascending order of period or relative deadlines respectively.
- **Job-level fixed-priority (JLFP) assignment**: In this case, jobs of the same task can have different priorities. The priorities are assigned based on the activation order of jobs in the system. One example is earliest deadline first (EDF) which assigns priorities to jobs based on their *absolute deadlines*. Figure 2.3 shows an example of scheduling three tasks with EDF. This schedule is the same produced by rate-monotonic fixed-priority (RM-FP) because RM-FP makes the same decisions as EDF when the task set is harmonic, i.e., all periods in the set divide each other.
- **Dynamic-priority assignment**: Here, priorities of jobs can change during their execution.

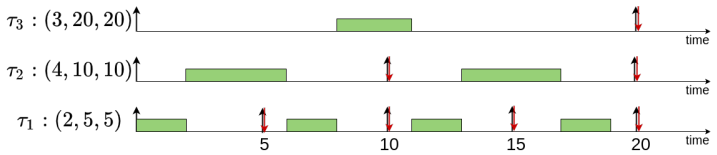
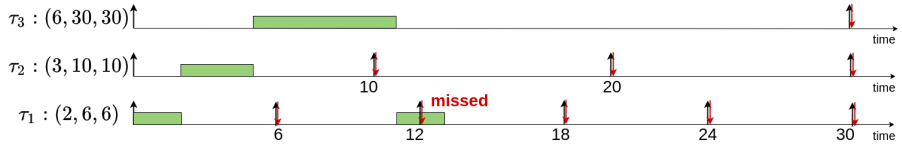


Figure 2.3: **EDF and RM-FP scheduling**

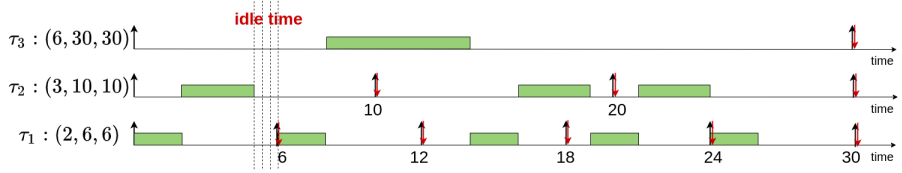
Another important classification of real-time scheduling algorithms is based on idle time insertion.

- **Work-conserving schedulers** are those that never leave a computing resource idle when there is work to be done, i.e., tasks are run as soon as possible.

- **Non-work conserving schedulers** on the other hand, purposefully insert idle times in schedules usually to guarantee schedulability of some future workload.



(a) **Work conserving scheduling**



(b) **Non-work conserving scheduling**

Figure 2.4: **Non-work conserving schedulers vs. work-conserving schedulers**

Additionally, an **optimal** scheduling algorithm is one that is able to schedule all feasible task sets, i.e., if this algorithm fails to schedule a given task set, no other algorithm can.

Schedulability Tests

The concepts of feasibility and schedulability as described in section 2.1 are often determined by schedulability tests. A schedulability test is a test that takes a set of tasks and a scheduling algorithm as input and returns an answer to whether or not the task set is schedulable by the given algorithm. In the event that the algorithm is known to be optimal, the result of this test represents the feasibility of the task set. Schedulability tests are classified as follows:

- **Necessary Tests**: A *false* result from a necessary test means that the task set is not schedulable..
- **Sufficient Tests**: A *true* result from a sufficient test means that the task set is schedulable.
- **Exact Tests**: A *true* result from an exact test means that the task set is schedulable and a *false* result means that means that the task set is not schedulable.

Multiprocessor Schedulers

Multiprocessor schedulers are classified as either global, partitioned or semi-partitioned.

- **The global approach** to scheduling involves one central scheduler and a central ready queue. The scheduler dispatches jobs to the processors according to a scheduling algorithm.
- **The partitioned approach** pins tasks in the system to processors in a way that all jobs of a certain task are only allowed to execute on one processor. This typically means there is a separate scheduler for each processor and the scheduling problem reduces to a series of uni-processor scheduling problems after task to processor assignment.

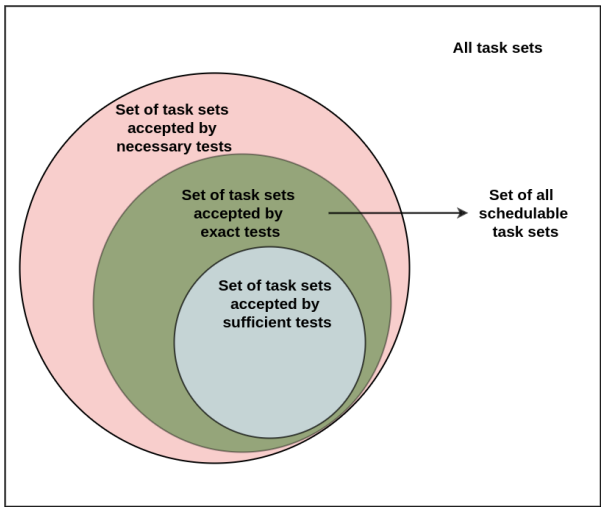


Figure 2.5: **Sufficient, exact, and necessary tests**

- **A semi-partitioned approach** to scheduling pins some tasks to processors while others are allowed to migrate.

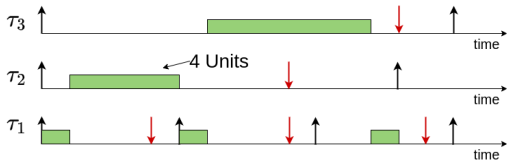
Bin Packing

Partitioned and semi-partitioned scheduling require us to first solve the partitioning problem which is known to be NP-hard for periodic hard real-time tasks [10]. However, if this problem is approached with the knowledge that each processor in a system has a limited processing capacity, it can be handled with bin packing heuristics. Bin packing has been extensively studied in algorithmics and is formulated as finding the minimum number of containers required to house a set of items of varying volume or weights [15].

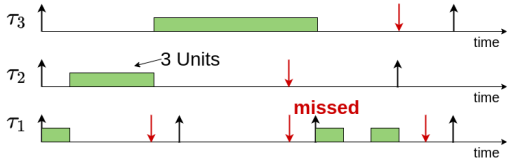
Volume is sometimes interpreted to mean task utilisation under real-time systems and a task is placed in a container *(processor)* in the event that its addition does not make utilisation of the container exceed 1. However, in non-preemptive scheduling, we cannot depend on utilisation because the utilisation bound for non-preemptive scheduling is 0 [33]. This is because tasks are never preempted and a task with low utilisation can have a very long period and an execution time that is long enough to completely block other tasks in the system. As a result of this very low utilisation bound, we must couple whatever fitting algorithm is used with a *schedulability test*. The test employed has a huge effect on the success of the solution and the extent is explored empirically in our evaluation.

Another important factor in determining the success of bin-packing is the order in which the items are sorted. This determines the state of the bins when an item is to be placed and since the algorithm does not employ backtracking, this can make or break a solution. Some important bin packing heuristics are:

- **First-fit**: Under first-fit bin-packing, we traverse the containers in the same order for each new item and place the item in the *first* bin that can accommodate it. We only open a new bin the case that the item cannot fit into any existing bin.
- **Next-fit**: Under next-fit bin-packing, we keep track of the last bin in which an item was placed and for any new item, traverse from this point onwards. Like first-fit, we place the item in the *next* bin that can accommodate it.



(a) **Safe schedule**



(b) **Unsafe schedule**

Figure 2.6: **Unsustainability of EDF [33, 36]**

- **Best-fit**: Like first-fit, we traverse the existing bins from the beginning for each new item and only open a new bin the case that the item cannot fit into any existing bin. The difference is that we place the item in the *fullest* bin that can still accommodate it.
- **Worst-fit**: Much like best-fit except that we place the item in the *least full* bin that can accommodate it.

2.3.3 Sustainable Scheduling

A scheduling algorithm is said to be sustainable when all task sets accepted by the algorithm as feasible remain so if the timing constrains under which it was tested become more relaxed [6]. These timing constrains include decreased execution time, increased deadlines or increased periods. The existence of non-sustainable systems speak to the anomalies present under non-preemptive execution model.

Figure 2.6 shows an example scenario. In Figure 2.6(a), at the time τ_2 completes, τ_1 has been released and the scheduler chooses the job of this task as it has the earliest deadline. In Figure 2.6(b), we see that a reduction of the execution time of τ_2 leads the scheduler to make a decision that causes a missed deadline. At the decision point, τ_1 is not yet released and as such cannot be a contender for the earliest deadline task [33].

2.4 Summary

In this chapter, we have presented a general introduction on the main focus of our work. Important definitions and notations of our system model have been introduced. Subsequent chapters refer to these models to explain the different solutions proposed by this thesis.

Chapter 3

Related Work

In this chapter, a brief survey of the existing techniques with similar aims or methods to this work is presented. We treat the related work under two broad categories namely sequential tasks in 3.1 and parallel tasks in 3.2 showing proposed solutions from previous publications.

3.1 Sequential Tasks

Popular schedulers such as earliest deadline first (EDF), show poor performance when applied on non-preemptive tasks [36]. Here, we consider schedulability as a performance metric for a scheduler where schedulability refers to the ability of a scheduling algorithm to successfully schedule its input task sets, namely, build feasible schedules for the input task sets. An important tactic that has been employed in literature to improve schedulability of non-preemptive task sets is the concept of non-work conserving schedulers. In this case, processors are allowed to be kept idle by the scheduler even when there is ready workload. Novel non-work conserving schedulers have been presented by Nasri et al. [37, 36].

Precautious RM [37] works by looking at the next job of the highest-priority task at every decision point. It schedules the highest-priority ready job only if it does not cause a deadline miss for the next (non-ready) job of the task with the smallest period. Critical window EDF (CW-EDF) [36, 34] on the other hand, defines a *critical window* at each decision point and schedules the current job that has the earliest deadline in the ready queue only if it will not cause a deadline miss for any of the jobs in the critical window. If scheduling the current job with the earliest deadline is going to cause a timing violation for any of the jobs in the window of interest, then the processor is kept idle until the next release event. CW-EDF only looks at one future job of each task that is absent from the ready queue at the current decision point.

Figure 3.1 was culled from [34] and shows the improvement of a non-work conserving scheduler like CW-EDF over work-conserving schedulers like non-preemptive fixed-priority (NP-FP) and non-preemptive earliest deadline first (NP-EDF). The figure presents a plot of schedulability ratio against utilisation (U) on a uni-processor platform. While non-work conserving schedulers have been able to schedule many cases that work-conserving schedulers could not, both papers [37, 36] only consider uni-processor platforms. In the event that a system is completely partitioned, these can be used for each processor.

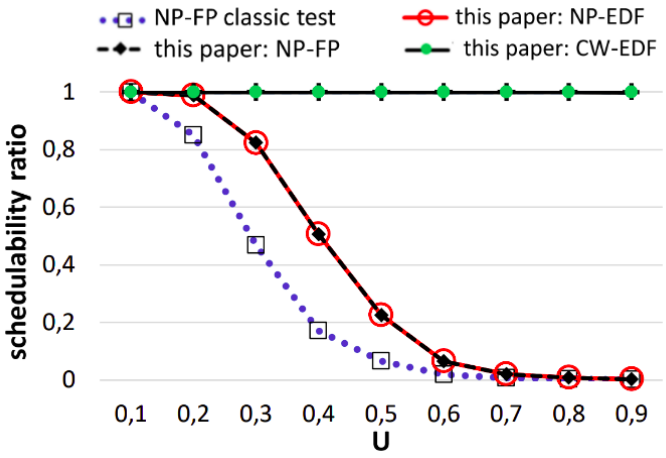


Figure 3.1: **Performance of CW-EDF [34]**

3.1.1 Partitioned Scheduling

The question of how to partition a given task set still remains. Classical partitioning algorithms as discussed in section 2.3.2 have been employed in combination with schedulability tests to extract partitions for a given task set.

Mayank et al. [30] performed an analysis for partitioned non-preemptive EDF scheduling and results presented showed that *best-fit* and *first-fit* produce the best results, where the tasks were sorted according to their utilisation in a non-increasing order. The method used to test if a new task fits on a processor was according to Jeffay's Test [23] which is formulated as $\forall i, 1 < i < n;\ \forall L, T_1 < L < T_i$:

$$L \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{T_j} \right\rfloor C_j \tag{3.1}$$

This equation checks that the computation demand in every time interval L is never greater than the size of the interval. Otherwise, such a task set will require more computation time than any unit-speed processor can offer in that interval and is thus unschedulable. This condition was proven to be only sufficient for periodic tasks in [36].

An earlier work in a similar scenario was presented in [20] where partitioning heuristics and an approximation of the demand bound function (DBF) [7] are combined to form a partitioning algorithm for limited-preemptive tasks. This test ensures the resulting partitions are EDF schedulable.

The approximation used [1] is computed as:

$$DBF(\tau_i, L) = \begin{cases} 0, & \text{if } L < D_i \\ WCET_i + u_i(L - D_i), & \text{otherwise.} \end{cases} \tag{3.2}$$

A task is added to an existing partition π if:

$$\left(D_i - \sum_{\tau_j \in \pi} DBF(\tau_j, D_i) \right) \geq C_i + q_{max}(\tau), \tag{3.3}$$

where $q_{max}(\tau)$ is the maximum length of non-preemptive execution present in the task set. In a completely non-preemptive scenario, this value is simply the longest execution time of all tasks in the considered task set.

Other works like Fan et al. [18] consider partitioning under fixed-priority scheduling. An existing schedulability test that can be applied to partitioning under fixed-priority scheduling is Davis' Test [17]. It works by first computing a busy period which is the duration of time for which the task undergoing the test experiences the worst scheduling conditions. This is a fixed-point iteration computed for a task with priority i as:

$$t_i^{n+1} = B_i + \sum_{\forall k \in hep(i)} \left\lceil \frac{t_i^n}{T_k} \right\rceil C_k, \tag{3.4}$$

where B_i is the worst *blocking* experienced by a task with priority i and $hep(i)$ refers to set of tasks with priority higher than or equal to i.

The worst-case response time is then computed for every job that falls in this time window. The number of jobs for which this is computed is:

$$q_i = \left\lceil \frac{t_i}{T_i} \right\rceil \tag{3.5}$$

We compute the worst-case response time (WCRT) for every instance q in q_i as:

$$WCRT_i^{n+1}(q) = B_i + qC_i + \sum_{\forall k \in hp(i)} \left\lceil \frac{WCRT_i^n}{T_k} \right\rceil C_k \tag{3.6}$$

To pass this test, no instance must have a WCRT greater than the relative deadline of the task.

3.1.2 Global Scheduling

Apart from partitioning, there are other possibilities such as globally scheduling task sets. The following attempts have been made to extend EDF scheduling to the global case.

Baruah [4] presents a schedulability condition for non-preemptive task sets under global EDF scheduling. This condition arose from a modification of the preemptive scheduling condition to account for *blocking*.

In the preemptive case, a task set is schedulable by global EDF upon m processors if:

$$U_{sum} \leq m - (m-1)U_{max} \tag{3.7}$$

where U_{sum} and U_{max} represent the total and maximum utilisation in the task set respectively. However, due to blocking, utilisation is not a dependable metric for non-preemptive global EDF.

The adjustment made is to compute a metric

$$V_i = \frac{C_i}{T_i - e} \quad , \tag{3.8}$$

which is a utilisation measure for a task subject to a maximum blocking of

$$e = \max\{C_i \mid \forall \tau_i \in \tau\}. \tag{3.9}$$

Thus, the resulting schedulability condition for non-preemptive global EDF is:

$$V_{sum} \leq m - (m-1)V_{max}, \tag{3.10}$$

where V_{sum} and V_{max} represent the total and maximum adjusted utilisation in the task set respectively.

Response-time based analysis techniques also exist for global non-preemptive scheduling. Nasri et al. [38] presented a sufficient test based on a *schedule abstraction graph*. The *schedule abstraction graph* of a task set, shows all possible

job orderings in every possible schedule that can be generated from a given JLFP scheduling policy for a given job set. Each node in the graph represents a state of the system and is characterised by the finish time interval, i.e, earliest and latest finish time of any job ordering that leads to that state. State transitions are represented by edges and are made by scheduling jobs. From the graph, we can extract the worst-case response times of all jobs in the system. This test is exact on a uni-processor platform [34] but only sufficient for global non-preemptive scheduling.

Yalcinkaya et al. [45] provide an exact test for global non-preemptive scheduling. The test employs timed automata models of the task and schedule. The task automata contains states such as *ready, running, completed* and *miss* which represent different conditions a task can be in. State transitions are made according to decisions of the scheduler automata. To pass the test, no task should reach the state where a deadline is missed, i.e, *miss* state.

3.1.3 Semi-Partitioned Scheduling

In semi-partitioned schedules, the majority of tasks are pinned to processors and some are left to migrate. Under preemptive scheduling, an explored method is to split the utilisation of tasks determined to be migratory and then use partitioning techniques to determine where each portion of a migratory task is executed as in [11]. There are two main strategies, [2] and [11], that split the utilisation of migratory tasks. They allow the execution of any job of such a task to be shared between a set of processors and both strategies maintain EDF schedulability of the resulting semi-partitioned task set.

A key difference between [2] and [11] is in the way tasks are split. Andersson et al. [2] group tasks into heavy and light groups with both groups executed on disjoint sets of processors. No such separation is used by Burns et al. [11], instead processors are filled with un-split tasks until it is no longer possible. After which, the remaining tasks are split into parts with the first part having its computation time equal to its deadline (a C=D splitting scheme).

These are possible strategies for preemptive scheduling but fail in the non-preemptive case because tasks can only migrate at job boundaries. To the best of our knowledge, no scheme that takes this into consideration has been demonstrated in the literature.

3.2 Parallel Tasks

Parallel tasks can also be scheduled using global or partitioned scheduling algorithms but intra-task parallelism blurs the definition of semi-partitioning. Additionally, DAG scheduling can be classified in two other categories *direct scheduling*, i.e., systems in which the scheduler is aware of the precedence constraints of the DAG and *decomposed-based scheduling*. Decomposition involves expressing DAG precedence as timing requirements. In a decomposed DAG, each sub-task is assigned a safe offset and deadline in a way that ensures that precedence is maintained. The resulting deadline constrained sub-tasks are then scheduled using the sequential task concepts which are typically less difficult to analyse.

3.2.1 Direct Scheduling

Global scheduling techniques have also been applied to parallel tasks as in [27] which proved a capacity augmentation bound of $4-\frac{2}{m}$ for global EDF scheduling of DAG tasks on m processors under preemptive scheduling. This bound means

that any DAG task set with total utilisation $\frac{m}{4-\frac{2}{m}}$ is schedulable by global EDF on m unit speed processors provided that no DAG in the task set has a critical path longer than $\frac{1}{4-\frac{2}{m}}$. Nasri et al. [39] provide a sufficient test for globally scheduled DAG Tasks using a *schedule abstraction graph* for any global job-level fixed-priority scheduler.

There is another technique called federated scheduling which exclusively assigns n of m processors to a single parallel task. Li et al. [26] presents a scheme that assigns processors to DAG tasks based on the relationship between the utilisation and the critical path length. The number of processors assigned exclusively to a single task is computed as:

$$n = \left\lceil \frac{C_i - P_i}{D_i - P_i} \right\rceil \tag{3.11}$$

Tasks with utilisation less or equal to unity do not need more than one processor. All such tasks in the system are left to be globally scheduled on whatever processors are left after the assignment of processors to the heavy tasks, i.e., tasks whose utilisation is larger than 1. The tasks considered are all implicit-deadline tasks while [5] shows analysis for arbitrary-deadline tasks.

Although some literature regards federated scheduling to be the partitioned counterpart of parallel tasks [5], we separate these concepts by defining the partitioned DAG as one whose sub-tasks are pinned to processors. Partitioned scheduling of DAG tasks has been studied by Casini et al. [14] who presented a partitioning algorithm for non-preemptive scheduling of DAG tasks. The main idea of the solution is to successively fit sub-tasks of a DAG on processors while ensuring that the worst-case response time satisfies the deadline requirements on each processor. These worst-case response times were computed by a path-based response-time analysis also developed in the paper. It builds on the response-time analysis of preemptive parallel tasks developed in [21]. The scheme was demonstrated to outperform known global schedulers under fixed-priority scheduling and is to the best of our knowledge, the first of such a partitioning strategy.

3.2.2 DAG-decomposition-based Scheduling

In this approach, a DAG is decomposed into a series of sequential tasks where each task is formed from a node of the DAG. The decomposition is done in such a way that precedence constraints are maintained by the timing parameters of the decomposed tasks, namely, offsets and deadlines. Decomposed scheduling has been explored in [41, 40, 24]. Each work shows analysis for schedulability of the resulting decomposed task set under preemptive global EDF. The general method is to first build the *ideal schedule* assuming infinite processors are present. As shown in Figure 3.3, such an *ideal schedule* completes its execution in a time equal to the critical path length. From this schedule, information on the earliest possible release and latest safe start times of each sub-task in the DAG is extracted. This information is then employed for decomposition in different ways.

The method in [41] divides the ideal schedule into segments and then assigns relative deadlines to each segment based on the amount of parallelism present in the segment. This method was found to have a resource augmentation bound of 4 under preemptive scheduling with global EDF. This means that if there exists any way to schedule the DAG on m identical unit-speed processors, then the decomposed task set is schedulable by global EDF on m identical processors if each processor is 4 times as fast as the original. The bound was found to be $4 + 2p$ under non-preemptive scheduling where p is a measure of the effect of

Work	Authors	Problem Focus
Sequential Tasks	Andersson and Tovar [2]	Semi-partitioned scheduling of preemptive tasks.
	Fischer and Baruah [20]	Limited-preemptive partitioned scheduling of sequential tasks.
	Baruah and Burns [4]	Schedulability analysis of non-preemptive task sets under global EDF scheduling.
	Nasri and Kargahi [37]	Non-work conserving scheduling of non-preemptive tasks upon uni-processor platforms.
	Nasri and Fohler [36]	Non-work conserving scheduling of non-preemptive tasks upon uni-processor platforms.
	Mayank and Mondal [30]	Non-preemptive partitioned scheduling of sequential tasks.
	Guan, Yi, Deng, Gu and Yu [22]	Schedulability analysis of non-preemptive task sets under global FP scheduling.
	Nasri, Nelissen and Brandenburg [38]	Response-time analysis for non-preemptive job sets under global scheduling.
	Yalcinkaya, Nasri and Brandenburg [45]	An exact schedulability test for non-preemptive self-suspending real-time tasks.
	Burns, David, Wang and Zhang [11]	Semi-partitioned scheduling of preemptive tasks.
Parallel Tasks	Li, Chen, Agrawal, Lu, Gill and Saifullah [26]	Federated scheduling of implicit deadline parallel tasks upon multiprocessor platforms.
	Saifullah, Ferry, Li and Agrawal [41]	Decomposition based scheduling of parallel tasks upon multiprocessor platforms.
	Qamhieh, George and Midonnet [40]	Decomposition (Stretching) based scheduling of parallel tasks upon multiprocessor platforms.
	Li, Luo, Ferry, Agrawal, Gill and Lu [27]	Global EDF scheduling of parallel tasks upon multiprocessor platforms.
	Baruah [5]	Federated scheduling of arbitrary deadline parallel tasks upon multiprocessor platforms.
	Jiang, Long, Guan and Wan [24]	Decomposition based scheduling of parallel tasks upon multiprocessor platforms.
	Casini, Biondi, Nelissen and Buttazzo [14]	Non-preemptive partitioned scheduling of parallel tasks upon multiprocessor platforms.
	Nasri, Nelissen and Brandenburg [39]	Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling.

Table 3.1: **Related work summary**

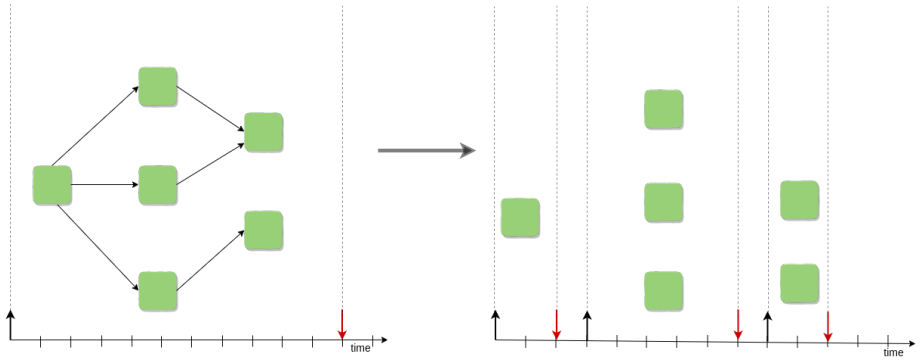
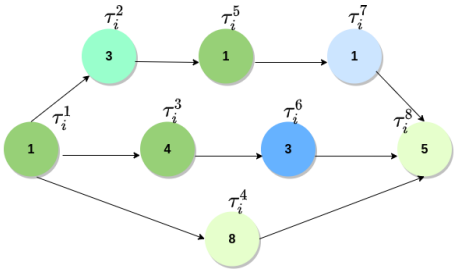
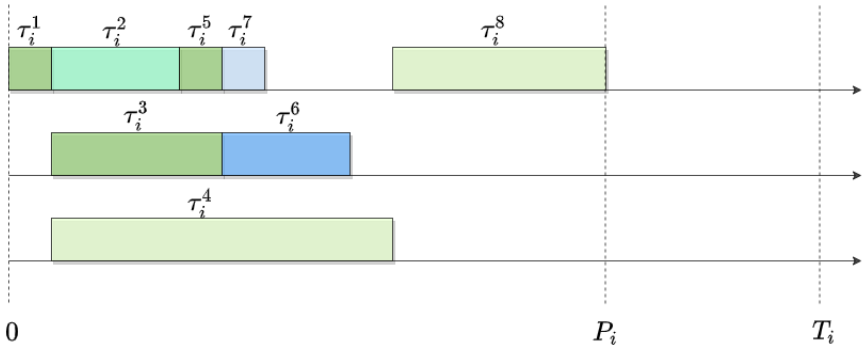


Figure 3.2: **DAG decomposition**

blocking computed as $\frac{E_{max}}{E_{min}}$. E_{max} and E_{min} are the maximum and minimum WCET among all nodes of the DAG respectively.



(a) **Sample DAG**



(b) **Ideal schedule of sample DAG**

Figure 3.3: **Ideal schedule of a DAG**

Qamhieh et al. [40] employ the concept of stretching. Stretching entails progressively adding sub-tasks to the critical path until it has a utilisation of 1. After this, it executes on a dedicated processor. Offsets and deadlines are assigned to the remaining sub-tasks or portions of sub-tasks that did not make it to this stretched path following the precedence constrains of the DAG. Each sub-task or portion must only be released after all its predecessors have completed execution and must have a deadline earlier than the start time assigned its earliest successor. These offsets and deadlines are based on both the structure of the DAG and the positions of the sub-tasks in the stretched path. We deal with portions of sub-tasks because the publication considers preemptive scheduling. Under non-preemptive scheduling, no sub-task is allowed to be split.

Jiang et al. in [24], segment the ideal schedule and differentiate light and

heavy segments. The assignment of relative deadlines is based on a measure defined as the *structure characteristic value* of the DAG. This measure is computed as the sum of the contribution of heavy segments to the sequential computation time of the DAG and the contribution of light segments to the critical path. It is handled this way to minimise the density of both threads and segments which directly affect schedulability of the resulting decomposed DAG.

Of these methods, [41] and [24] are comparable to our work because they can be made to handle non-preemptive DAG tasks. The method in [40] allows the execution of a node of a DAG to be split between two processors and this cannot hold under non-preemptive execution.

3.3 Summary

A summary of the related work presented in this chapter is as shown in Table 3.1. The table shows closely related works and their focus areas ranging from designing scheduling policies to building schedulability tests.

Chapter 4

Sequential Task Scheduling

In this chapter, we present our solutions for scheduling sequential tasks on multiprocessor platforms. In section 4.1, we present our partitioned scheduling solutions. In section 4.2, we present the first semi-partitioned scheduler for non-preemptive task sets and finally in section 4.3, we introduce a global non-work conserving policy. These solutions provide a means to answer research questions **RQ1**, **RQ2** and **RQ3**.

4.1 Partitioned Scheduling

In section 4.1, we present three partitioning solutions. The first solution is sustainable partitioning, which is an idea for partitioning when systems are unaffected by execution time variations. We then discuss clique partitioning in section 4.1.2 which can be used along with our sustainable partitioning idea to divide task sets into partitions. Thirdly, we apply the idea of cliques to build a necessary test for partitioned scheduling in section 4.1.3.

4.1.1 Sustainable Partitioning

Under partitioned scheduling, tasks are assigned to processors at design time and are executed only on the assigned processors through out the lifetime of the system. Bin-packing heuristics like first-fit and next-fit have been used to accomplish task to processor assignment. Under preemptive partitioning, utilisation is often used to determine if a task can fit on a processor. Under non-preemptive scheduling, we cannot depend on utilisation because the utilisation bound is 0 [33]. As a result of this very low utilisation bound, we must couple whatever fitting algorithm is used with a *schedulability test*.

Schedulability tests for non-preemptive scheduling typically guarantee sustainability by accounting for possible anomalies in the schedules and as such are quite pessimistic [17, 23]. One way to improve the schedulability when we have "non-deterministic" execution times is to use a sustainable scheduling algorithm such as FIFO scheduling to begin with. In that case, if the scheduler is able to successfully schedule the task set when tasks have their worst-case execution times, it will be able to schedule the task set for any other smaller execution time values as well. While FIFO scheduling in its pure form is known to have a poor schedulability ratio in contrast to other scheduling algorithms, there is a recent extension of it called FIFO with offset tuning (FIFO-OT) [35] that uses the FIFO policy in its core but adapts job offsets such that the underlying FIFO scheduler is able to regenerate a certain schedule at runtime. FIFO-OT is not able to create schedules by itself but it is rather an online technique to allow a given offline schedule be recreated at runtime. For this aim, FIFO-OT requires

a feasible input schedule that is built for jobs in one hyperperiod in which each job of a task has its worst-case execution time (WCET). There is no limit to how such a schedule has been generated. One can use a brute-force algorithm or one of the existing efficient online scheduling policies such as CW-EDF [36].

Given that among the existing uni-processor scheduling policies, CW-EDF has the highest schedulability ratio, we build the schedule of one hyperperiod using CW-EDF assuming that each job of each task has the worst-case execution time. If the resulting schedule does not have a deadline miss, then it is guaranteed that the that task set can be successfully scheduled on one processor using FIFO-OT. It is worth noting that CW-EDF itself is not a sustainable scheduling policy and hence can not replace the FIFO-OT technique. However, with the combination of a schedule generated by CW-EDF and applied online via FIFO-OT, one can ensure that the task set can be successfully scheduled on one processor at runtime. Moreover, such a combination of techniques will have the highest schedulability compared to online scheduling policies including CW-EDF itself.

4.1.2 Clique Based Partitioning

In this section, we revisit the partitioning problem and try to deviate from the classical bin-packing-based solutions and move on to graph-based solutions. A graph, in this context, is denoted by $G = (V, E)$, where V is the set of vertices and E is the set of edges.

The key idea of our solution is to find *cliques* in the graph where a clique represents a set of tasks that can be partitioned together. A *clique* is a complete sub-graph, i.e., a subset of vertices in which each vertex is connected to every other member of the subset. A *maximal clique* is a clique that cannot be extended by the addition of any other vertex in the graph.

Our starting point is to model a given task set as a graph where a vertex $v \in V$ represents a periodic task $\tau_i = (C_i, T_i, D_i)$ and edges are placed between tasks based on a predefined relationship between tasks. For our partitioning scheme, this relationship is defined as follows:

An edge exists between two tasks τ_i and τ_j if both tasks fulfil the necessary scheduling conditions for non-preemptive execution:

$$U_i + U_j \leq 1 \quad \wedge \quad C_j \leq 2(T_i - C_i) \quad \wedge \quad C_i \leq 2(T_j - C_j). \tag{4.1}$$

These are based on the necessary conditions for non-preemptive scheduling presented in [13] and ensure that

1. the feasible utilisation bound is not exceeded,
2. non-preemptive scheduling of two consecutive jobs of each task is feasible.

Theorem 4.1.1 *(From Cai et al. [13]) For any two non-preemptive tasks τ_i and τ_j that satisfy the equation $C_j > 2(T_i - C_i)$, it is impossible to find a feasible schedule for both tasks on one processor regardless of the scheduling policy used.*

Upon completion of the graph model, partitions are made by continuously removing the largest schedulable clique in the graph until the graph is empty, i.e., all tasks have been assigned a partition. We judge the size of a clique by the number of tasks it contains and break ties in favour of the clique with the highest total utilisation. The partitioning scheme also requires a schedulability test to determine whether a given clique is schedulable as shown in line 6 of algorithm 4.2. We use the sustainable scheduling algorithm introduced in section 4.1.1 and hence after selecting a partition, we check if the tasks can be successfully scheduled by CW-EDF for one hyperperiod where tasks run for their worst-case execution times. This process is illustrated by the following example.

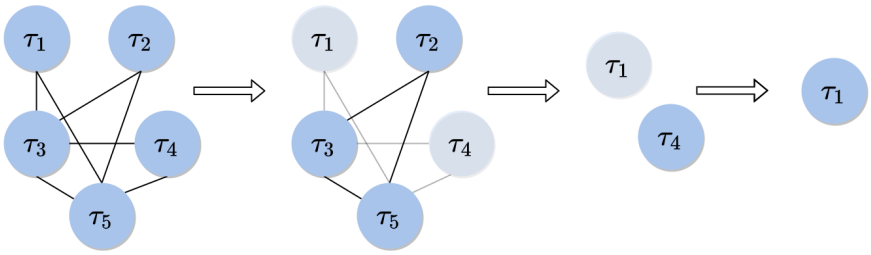


Figure 4.1: **Clique partitioning steps for tasks in Table 4.1**

Illustrated Example

Given a task set, $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$ with task parameters as shown in Table 4.1, Figure 4.1 shows the resulting graph model with cliques highlighted. A edge exists between τ_1 and τ_3 because the following relationships hold:

$$U_1 + U_3 = 0.7 < 1 \quad \wedge \quad C_1 < 2(T_3 - C_3) \quad \wedge \quad C_3 < 2(T_1 - C_1). \tag{4.2}$$

	T_i	C_i	U_i
τ_1	10	5	0.5
τ_2	20	15	0.75
τ_3	50	10	0.2
τ_4	50	30	0.6
τ_5	100	2	0.02

Table 4.1: **Sample task set to illustrate clique partitioning**

The set of maximal cliques in order of their size (judged by number of vertices and total utilisation) are $\{(\tau_1, \tau_3, \tau_5), (\tau_2, \tau_3, \tau_5), (\tau_3, \tau_4, \tau_5)\}$. Of these, the largest clique, (τ_1, τ_3, τ_5), has a utilisation sum 1.3 and is thus unschedulable. The next largest, (τ_2, τ_3, τ_5), is schedulable with a utilisation sum of 0.97. Thus, this is removed from the graph as a partition leaving only vertices τ_1 and τ_4. As there is no edge between them, they are assigned to two different partitions and our algorithm is complete as the graph is now empty. The resulting partitions are (τ_2, τ_3, τ_5), (τ_4) and (τ_1).

It should be noted that subsets of maximal cliques are also cliques and the algorithm begins to visit these in the case that no maximal clique is schedulable. Additionally, listing all cliques may theoretically take exponential time as there exist graphs with exponentially many cliques. However, considering the characteristics of typical industry task sets (laid out in chapter 7), the problem is expected to be solvable.

Algorithm 4.1: Graph Building Algorithm (τ)

```
1 V ← {v_i ∀ τ_i ∈ τ} ;
2 E ← ∅;
3 for τ_i ∈ τ do
4     for τ_j ∈ τ ∧ τ_j ≠ τ_i do
5         if Equation 4.1 then
6             E ← E + (τ_i, τ_j) ;
7 return G ;
```

Algorithm 4.2: Clique Based Partitioning (G, τ)

```
   Input: G, a graph model of τ according to algorithm 4.1
 1 P ← ∅ ;
 2 while G ≠ ∅ do
 3 |   G_o ← ∅ ;
 4 |   Find and order cliques by size;
 5 |   for clique ∈ G do
 6 |   |   if isSchedulable(clique) and |clique| > |G_o| then
 7 |   |   |   G_o ← clique ;
 8 |   Add G_o to P;
 9 |   G ← G − G_o;
10 return P;
```

4.1.3 A Necessary Test for Partition-ability

Due to the inherent complexity of the partitioning problem for periodic tasks, there is currently no exact solution that can determine whether a given task set can be successfully partitioned on a given number of processors. As a result, there is no way to know what the true limit of the partition-ability of periodic task sets is. Moreover, due to the lack of such an exact method for partitioning, there is no method (with a reasonable runtime) that allows us to quantify the pessimism of the existing partitioning heuristics.

The goal of this section is to introduce the first necessary test to evaluate the partition-ability of a given task set. In other words, to build an efficient test that is able to determine task sets that are impossible to be partitioned on a given number of processors. Our test can then be used to quantify the pessimism of the existing heuristics and quantify the limits of partitioning based solutions.

Our necessary test derives a lower bound on the minimum number of processors required to partition a given task set under non-preemptive scheduling. The test works by building the $conflict$ graph of the task set, denoted by $G = (V, E)$, where V is the set of vertices (representing tasks) and E is the set of edges (representing conflicts). A vertex $v \in V$ represents a periodic task τ_i. An edge $e \in E$ represents a *conflict* between the two tasks that are assigned to the vertices connected by that edge. Two tasks τ_i and τ_j ($T_i \leq T_j$) are in conflict if and only if they cannot be scheduled on one processor under any scheduling policy. In other words, tasks τ_i and τ_j violate the necessary schedulability condition [37] of non-preemptive tasks on a uni-processor platform:

$$U_i + U_j > 1 \quad \vee \quad C_j > 2(T_i - C_i) \quad \vee \quad C_i > 2(T_j - C_j). \tag{4.3}$$

To build the conflict graph, every pair of tasks in the task set are considered; if they have a conflict, then an edge is added between their vertices. The size of the largest clique in the conflict graph represents the minimum number of processors required to schedule the task set since none of the tasks in that clique can be partitioned together with any other tasks in the clique. Hence, the size of the largest clique, denoted by M, is a lower bound on the number of processors required to partition this task set. If the available number of processors, denoted by m, is smaller than M, the task set is not partition-able on m processors. This can also be formulated as a graph colouring problem [29] with no two vertices who share an edge coloured alike.

Algorithm 4.3: Conflict Graph Building Algorithm (τ)

```
1 V ← {v_i ∀ τ_i ∈ τ} ;
2 E ← ∅;
3 for τ_i ∈ τ do
4     for τ_j ∈ τ ∧ τ_j ≠ τ_i do
5         if Equation 4.3 then
6             E ← E + (τ_i, τ_j) ;
7 return G ;
```

Algorithm 4.4: Necessary Test For Partition-ability (τ)

Input: G: a conflict graph model of τ according to algorithm 4.3
Output: M: the lower bound of the number of processors on which the task set can be successfully partitioned.

```
1 C ← {cliques ∈ G} ;
2 M ← 0 ;
3 for clique in C do
4     if |C| > M then
5         M ← |C| ;
6 return M ;
```

Illustrated Example

Using the same task set as presented in Table 4.1, we demonstrate the workings of the necessary test. The conflict graph for this task set is as shown in Figure 4.2. We see that this graph is the anti-graph of that presented in Figure 4.1. This is because the conditions for forming an edge are the reverse of those used in the clique partitioning algorithm.

From Figure 4.2, we see that there is only one clique in the resulting conflict graph and thus, this is also our largest clique. The size of a clique in this context is judged purely on the number of vertices contained therein, i.e., M for this graph is 3. This is interpreted to mean that the smallest number of partitions possible for any algorithm is **3**.

This test is only necessary because the conflict conditions themselves are necessary. Additionally, since we build it based only on pair wise conditions, the absence of a clique among any subset of more than 2 tasks does not necessarily mean that they together satisfy the necessary condition for scheduling.

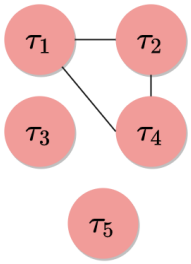


Figure 4.2: **Conflict graph model of tasks in Table 4.1**

4.1.4 Clique Finding Problem

Both algorithms in sections 4.1.2 and 4.1.3 depend on finding cliques in a graph. The class of graphs we build are undirected, i.e., a conflict or the absence thereof between two tasks exists both ways. Our versions of the clique problem have two formulations:

1. finding the maximum clique, i.e., the clique with the largest number of vertices of all cliques in the graph,

2. finding all maximal cliques, i.e., all cliques that cannot be extended by the addition of a vertex.

Both problems are related in that the maximum clique can be found from listing all maximal cliques and then selecting the largest one. Moon et al. [32] found that the largest possible number of maximal cliques in a graph with n vertices is $3^{n/3}$ and as such, the worst case time complexity of the maximum clique problem is similarly $O(3^{n/3})$. In our implementations, we use the clique finding algorithm proposed in [9] which while not being the fastest available in the literature, provides good enough results for the cases we handle in this thesis. On average, it took less than 30 minutes to find the maximum clique.

4.2 Liquid-Path Scheduling

Partition-ability vs Feasibility: Our Motivation

In section 4.1.3, we have presented a necessary test for partition-ability of task sets and this forms a bound on what is possible for any partitioning scheme. However, failing such a test does not imply that the task set is infeasible. In this section, we present a novel job-based fitting technique that results in a semi-partitioned schedule of a task set. The intuition behind semi-partitioning is to enable efficient use of processing time in cases where processors do not have room for every job of a task but can likely accept some of its jobs. Under non-preemptive scheduling, such task migration is only possible at job boundaries in the case of sequential tasks.

Unlike partitioning heuristics that work on the tasks, we try to partition the jobs of the tasks in a hyperperiod. Namely, we start by obtaining all jobs in one hyperperiod and sort them according to some criteria and then assign each job to the available processors. In order to see which processor can host the job, we keep track of the currently assigned jobs to each processor in an ordered list which contains the order of execution of the currently assigned jobs and the leeway of each job in the list. We call this list a liquid path; it is liquid because new jobs can use the available slack between existing items and fit themselves in the path. We allow such insertions only if adding a new job to the path does not result in a deadline miss for any other already admitted job.

Next, we introduce the pseudo-code, job-ordering criteria and job-admission policy of our *liquid-path scheduling* algorithm.

Liquid-path scheduling algorithm

Liquid-path scheduling is a job-based fitting technique. Algorithm 4.5 shows the steps for liquid-path scheduling given a set of tasks τ and m processors. In line 1, we sort the given task set in descending order of WCET and declare m empty paths $P = \{P_1, ..., P_m\}$. We then loop through every job in the hyperperiod in line 4 and try to fit it on one of the paths in P. If no suitable position is found for the job, i.e., all paths are too solid to accommodate the job, the algorithm

fails. Otherwise, we continually place jobs on suitable paths until all jobs have been handled and we return P which can then be interpreted as a schedule.

A path is called *liquid* for as long as there is enough slack to insert jobs in between and gains *solidity* when any job insertion between any two path members would result in a deadline miss along the path. A path is feasible when every member satisfies its timing requirements.

Algorithm 4.5: Liquid-Path Semi-Partitioning (m, τ)

Input: A task set τ, a processor model with m processors

```
1  sort τ in descending order of WCET ;
2  P ← {P_1, P_2, ..., P_m} where P_i is an empty path;
3  for τ_i ∈ τ do
4      for J_{i,j} ∈ J_i do
5          fitFound ← False ;
6          core ← 1 ;
7          while ¬fitFound and core ≤ m do
8              find spot for job according to Equations 4.4, 4.5 ;
9              if Equation 4.6 then
10                 fitFound ← True ;
11                 update P_core according to 4.7 - 4.10 ;
12             core ← core + 1 ;
13         if ¬fitFound then
14             return false ;
15 return P ;
```

In one extreme case, we end up with a completely partitioned schedule and in the other we end up with a global schedule in which every task has migratory jobs. The details of fitting a job on a path are discussed below.

A path $L = \{l_0, l_1, ..., l_k\}$ is defined as a sequence of jobs executed in a strict order on a processor, where $l_i = (c_i, est_i, lst_i, S_i, F_i, d_i)$ represents the job at the i-th position on the path with a computation time c_i, an earliest start time est_i, a latest start time lst_i, a starting time S_i, a finishing time F_i and an absolute deadline d_i. l_0 and l_k are padding elements whose purpose is to enable the algorithm place elements at the head and tail of the path thus, the length of a path is $k - 1$.

Job Location

To insert a job $J_{x,y}$ in a path, we have to decide what position exactly to put the job. It is possible to decide on such a position for a job by checking every item in the path but the search can be reduced by only considering positions in which it is feasible to insert $J_{x,y}$. We do this by checking only items that fall within the execution window of $J_{x,y}$ defined by the interval

$$w_{x,y} = [r_{x,y}, d_{x,y}], \tag{4.4}$$

where $r_{x,y}$, is the arrival time of the job in consideration and $d_{x,y}$ is its deadline. This is because it is not feasible to place the job outside of this interval. A position i is considered for inserting $J_{x,y}$ if

$$w_{x,y} \cap [F_{i-1}, lst_i] \neq \emptyset \tag{4.5}$$

In the event that multiple positions satisfy the conditions for successfully scheduling $J_{x,y}$, one of them is selected based on a heuristic. We do not use

backtracking and do not check multiple solutions to ensure the time complexity of the search is linear with respect to the number of jobs in the hyperperiod. Otherwise, it explodes to an exhaustive search of a feasible non-preemptive schedule which is a known NP-Hard problem.

Path Insertion Policy

Jobs are executed non-preemptively hence, insertion of a job $J_{x,y}$ in a position i on a path can only be between the finishing time F_{i-1} of a predecessor element l_{i-1} and latest start time lst_i of a successor element l_i. As jobs are allowed to alter the starting time of l_i, a bound is placed to ensure the path remains feasible. The estimated finishing time of $J_{x,y}$, i.e., $f_{x,y}$ is derived from $s_{x,y} + C_x$ where $s_{x,y}$, i.e, its start time upon insertion is $s_{x,y} = max(r_{x,y}, F_{i-1})$. The following conditions must hold for $J_{x,y}$ to be inserted in a given position i

$$f_{x,y} \leq d_{x,y} \wedge f_{x,y} \leq lst_i \tag{4.6}$$

namely, the finishing time of the job is smaller than its deadline and smaller than the latest start time of its successor. Figure 4.3 shows the process of inserting a job $J_{x,y}$ in position 3 of the liquid path. Upon insertion, $J_{x,y}$ becomes l_3. The job that previously occupied l_3 is moved to position 4 as l_4 with its starting and finishing times updated as shown in the diagram.

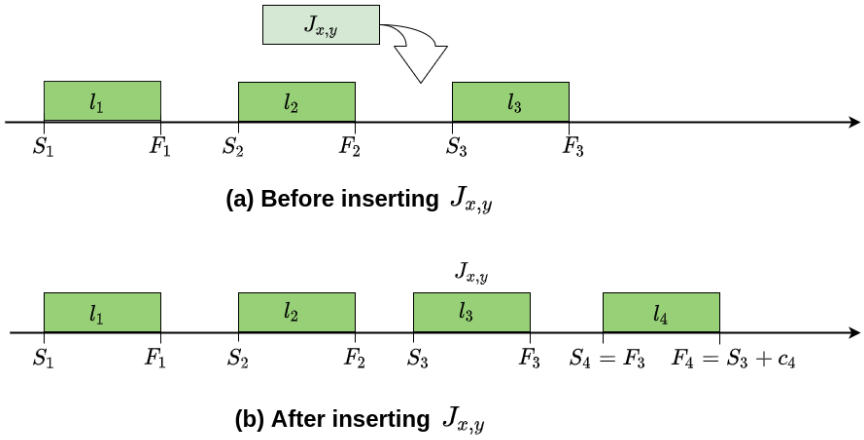


Figure 4.3: **Sample liquid-path job insertion**

After any job insertion, the rest of the path is updated using

$$\forall l_z \in L | z \geq i, S_z \leftarrow max(F_{z-1}, est_z) \tag{4.7}$$

until we either reach the tail of the path or a pair of elements in the path l_a and l_b for which the following condition holds:

$$b = a + 1 \ \wedge \ F_a < est_b. \tag{4.8}$$

The latest start time of a path element is updated before insertion in the path using

$$min(d_{x,y}, lst_i) - C_{x,y}, \tag{4.9}$$

where $C_{x,y}$ is the worst case execution time of $J_{x,y}$. By doing this, the latest start time is propagated along the path with every new insertion. Tasks before the point of insertion may also require their latest start time to be updated

to ensure the timing constraints of the newly inserted job are not violated by future insertions. This can be propagated backwards along the path using

$$lst_z = min(d_z, lst_{z+1}) - c_z, \tag{4.10}$$

until we reach the head of the path.

4.3 Global Scheduling

As shown in Figure 3.1, non-work conserving policies can greatly improve the schedulability of periodic non-preemptive task sets on uni-processor platforms. This motivated us to look into applying such policies on multiprocessor platforms. The goal is to see if non-work conserving policies can also improve global multiprocessor scheduling **(RQ4)**.

In this section, we introduce the first non-work conserving global scheduling policy. Non-work conserving policies are those that are able to make the decision to keep a processor idle even when there is a ready workload. The primary reason for such a decision is to ensure that future jobs will be able to meet their deadlines and as such, these policies typically try to look into the future. A non-work conserving scheduler has an idle-time insertion policy (IIP) which decides when to insert idle times in the schedule and how long they should last.

Our global scheduler is called by a processor whenever it finishes execution of a job, whenever an idle time expires or whenever a job is released and the processor is free. Whenever the scheduler is called, it selects the job with the earliest deadline from the ready queue. However, our global policy also uses the idea of a *critical window* [36] where the chosen earliest deadline job is scheduled if and only if its execution maintains the feasibility of a set of jobs that will be released in this window. Otherwise, we insert an idle time until the next release event.

The critical window consists of the next job of any task that has no pending job in the ready queue. To guarantee that these candidate jobs will be scheduled safely, the scheduler looks ahead to the future of the system and extracts the latest start time of each candidate job such that it will still meet its deadline. *The chosen earliest deadline job is scheduled if and only if it can complete its execution before the latest start time for the processor that is performing the look ahead*, otherwise, an idle time is inserted on that processor.

Any processor that calls the scheduler will have to do a look ahead for the whole system because under global scheduling, jobs can be run on any processor, i.e., there is no task to processor assignment. We refer to this algorithm as global critical window earliest deadline first (G CW-EDF).

4.3.1 Global CW-EDF Methodology

Algorithm 4.6 shows the pseudo code for G CW-EDF. At any time t at which the scheduler makes a decision, we begin by picking the earliest deadline job $J_{i,j}$ in the ready queue. We then create a set of candidate jobs. A job is considered a candidate job if it is the next job of a task with no pending job in the ready queue. For each candidate job, the absolute deadline is computed as $D_i^{next} = \left(\left\lceil \frac{t}{T_i} + 1 \right\rceil\right) T_i$.

The set of candidate jobs is then sorted in descending order of absolute deadlines, transforming it into an ordered list in line 2 of algorithm 4.6. Let us call this list N and denote its length by k. In addition to N, the algorithm also keeps track of the busy time of each processor, B_c in a list B where B_c is

Algorithm 4.6: Global Critical-Window EDF (G CW-EDF) (t, τ)

```
   Input: t, the current time and τ, the task set
1  Find the earliest deadline job in the ready queue J_{x,y} ;
2  Build the next ready jobs N;
3  Sort processor busy times B;
4  c ← 1 ;
5  for job J_i in N do
6  |  compute L_i from Equation 4.11 ;
7  |  if min{D_i^next, L_{i-1}} − C_i < B_c then
8  |  |  L_i = min{D_i^next, L_{i-1}} − C_i ;
9  |  else
10 |  |  LST_c = L_{i-1} ;
11 |  |  c ← c + 1 ;
12 |  |  L_i = D_i^next − C_i ;
13 if t + C_x ≤ LST_1 then
14 |  Schedule J_{x,y} ;
15 else
16 |  Schedule an idle interval until the next decision point;
```

the worst-case completion time of a processor c's current workload. If $J_{i,j}$ is scheduled on processor c at time t, the processor updates B_c as $B_c = t + C_i$.

We traverse the processors in ascending order of B_c which means that the first processor we test is the processor with the earliest busy time. This decision was made following the intuition that the processor that gets access to the global scheduler is the one with the earliest busy time and will likely be unavailable to run jobs in the near future. The chosen earliest deadline job is scheduled if its worst completion time is less than the computed latest finish time for the processor that has called the scheduler LST_1.

To achieve this, the algorithm keeps the processors ordered in ascending order of busy times. It computes the latest start time L_i for each job in N using Equation 4.11 beginning from latest deadline job in N. The latest start time L_i for each job in N is calculated assuming that the job will be placed on a particular processor. The algorithm begins the computation of latest start times from the earliest busy time in B and only moves to another processor when we decide that candidate jobs can no longer fit on that processor. In line 9, the algorithm updates the latest start time for a processor $LST_c = L_{i-1}$ whenever a job in N cannot be placed on the processor c, i.e, if $C_i > L_{i-1} - B_c$. This happens when there is no room on processor c and hence, the look ahead is continued from processor $c + 1$.

$$L_p(t) = \begin{cases} D_p^{next} - C_p & \text{if } C_p > L_{p-1} - B_c \vee p = 1 \\ min\{D_p^{next}, L_{p+1}(t)\} - C_p & \text{otherwise} \end{cases} \tag{4.11}$$

We illustrate an example in Figure 4.4. The ready queue currently contains jobs of τ_1 and τ_2 with a job of τ_1, $J_{1,y}$, being the earliest deadline task. The algorithm does the following:

1. It constructs a set of candidate jobs by considering the next jobs of all tasks that currently have no job on the ready queue. This set contains jobs of τ_3, τ_4 and τ_5. When sorted, we treat these jobs in order $J_{3,y}$, $J_{4,y}$ and then $J_{5,y}$. This is the makeup of N.

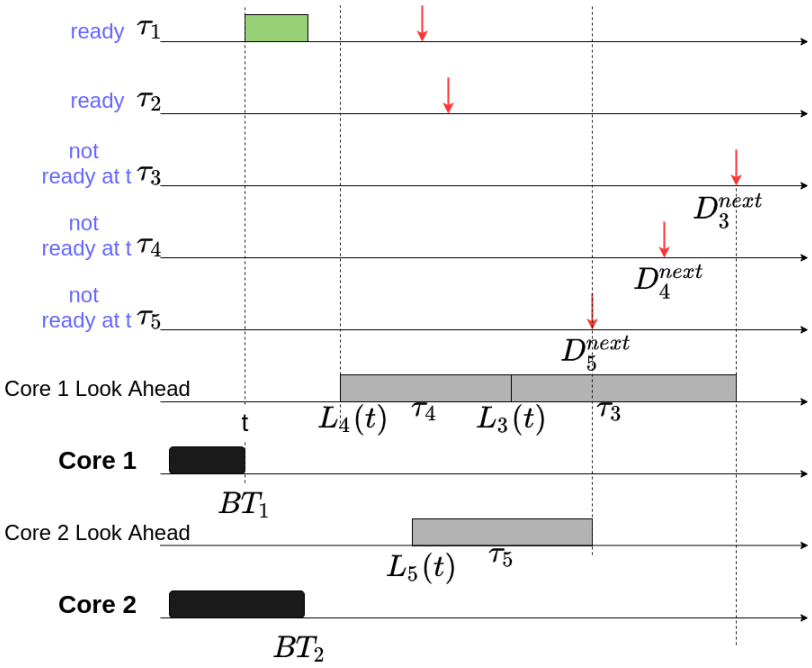


Figure 4.4: **Latest start time computation for global CW-EDF**

2. The processors are ordered in ascending order of busy times meaning that the processors are treated in order of B_1 and then B_2.

3. Candidate jobs are placed on processors from earliest to latest busy time thus:

   - We begin with attempting to place $J_{3,y}$ on processor 1. We compute the latest start time as: $L_3 = D_3^{next} - C_3$ because it is the first element in N.
   - Next, we place $J_{4,y}$. We compute $L_4 = L_3 - C_i$ because $D_4^{next} > L_3$.
   - The last job in N is $J_{5,y}$ and can no longer fit on processor 1. So, we move onto processor 2 and freeze LST_1 for processor 1 at L_4.

After all items in N have been treated, the chosen earliest deadline job is dispatched only if it can complete execution before the latest start time of the scheduler-calling processor. In this case, L_2 for processor 1.

The time complexity of the global CW-EDF is $O(nlog(n) + mlog(m))$ where n is the number of tasks and m is the number of processors. The $nlog(n)$ component is because we sort the number of jobs in the set of candidate jobs and in the worst case, every task in the system has a job in this set. Likewise, the $mlog(m)$ component is derived from sorting the busy time of all processors.

Chapter 5

Parallel Task Scheduling

The use of multiprocessor platforms allows us to exploit *intra-task parallelism* where a single task can have threads that are executed simultaneously. We represent such parallel tasks as directed acyclic graphs (DAG). Each node of the DAG represents a sub-task and the edges of the DAG represent the logical dependencies of the sub-tasks. Edges in the DAG define the execution flow of the task. However, there can be multiple valid sequences to execute a DAG task and the decision is up to the scheduler in use.

There are two broad approaches to such DAG schedulers namely **direct** and **decomposed** scheduling. In a direct scheduling approach, the scheduler is aware of the logical dependencies and precedence constraints of the DAG. Decomposed scheduling on the other hand expresses DAG precedence in the timing requirements. In a decomposed DAG, each sub-task is assigned a safe offset and deadline in a way that ensures that precedence is maintained. The resulting deadline constrained sub-tasks are then scheduled using sequential task concepts. The advantage this presents is that the scheduler becomes simpler and sequential task scheduling can be applied. It also allows us to handle mixtures of sequential and parallel tasks in a system uniformly.

In this chapter, we present a parallel-task scheduling approach that uses a decomposition algorithm in addition to partitioned scheduling as introduced in chapter 4. The solution and corresponding evaluation in chapter 7 answer parts of research question **RQ1**.

5.1 DAG Decomposition

Given a DAG task, a decomposition algorithm aims to return the DAG with all sub-tasks assigned offsets and deadlines. The rules are that a sub-task must only be released after the deadlines of all its predecessors and must have a deadline earlier than the release of its successors. By doing so, a scheduler that respects the timing constraints of the individual sub-tasks of a DAG will intuitively guarantee the precedence constraints of the whole DAG.

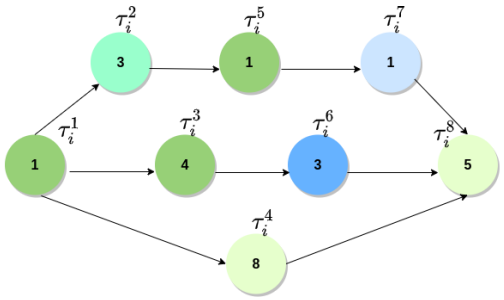
In the end, we want to come up with a decomposed DAG that is likely to be schedulable. We pay attention to the workload supplied to the system at any point in time and try to ensure that it is never larger than the computing resources available. This is because such a decomposition will not be schedulable. We quantify a workload by its density $\Delta = \frac{C_i}{D_i}$ which is a measure of how much computation must be carried out in one unit of time for the workload to meet its deadline.

In order to have higher chances for a successful decomposition, we form some additional guidelines for a decomposition algorithm. They are:

- **G1** *The density of a sub-task should be less than 1.*
  This means that no relative deadline must be less than the execution time of the sub-task. If this happens, the decomposed DAG is unschedulable on a unit-speed machine.
- **G2** *The total density of the active sub-tasks at any point in time should be less than m.*
  An active sub-task is a sub-task that has been released but has not completed execution and m is the number of processors in the system. If at any point in time, the total density of active sub-tasks exceeds m, then the decomposed DAG is unschedulable on m processors.

General idea of the decomposition algorithm

The first step in our decomposition algorithm is to build the ideal schedule, i.e., *the schedule in the presence of infinite processors.* At each point where a sub-task completes its execution in this schedule, we demarcate a **section** of the schedule. Figure 5.1 shows a sample DAG and 5.2 shows the ideal schedule of the sample DAG with the demarcated sections.



The values within the circles represent the WCET of a sub-task.

Figure 5.1: **Sample DAG task**

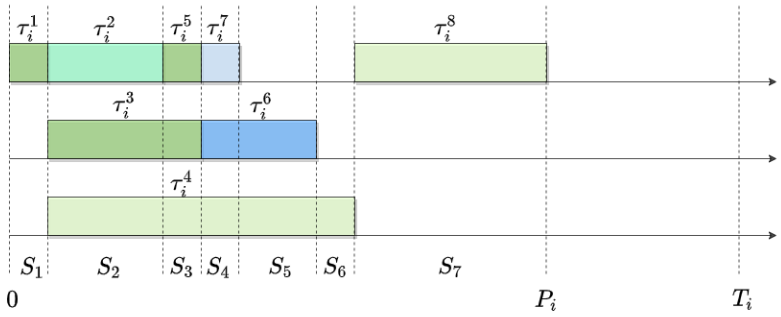


Figure 5.2: **Ideal schedule with sections**

Our assignment of offsets and deadlines are done on a section by section basis. For each section S_i, we assign a relative deadline that is greater than or equal to the execution time of the section. This creates some slack between the end of the section S_i and the begining of the next section S_{i+1} as shown in Figure 5.3.

Sub-tasks can sometimes belong to multiple sections and because we consider the non-preemptive execution model, we go one extra step of stitching sub-tasks back together and assigning offsets as the offset of the earliest section to which

it belongs and relative deadline as the sum of all the deadlines of the sections to which it belongs. This step is illustrated in Figure 5.4 and forms the decomposed DAG.

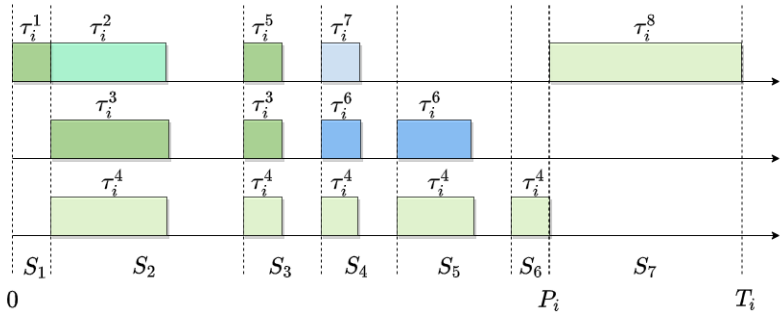


Figure 5.3: **Ideal schedule with sections and their assigned slack**

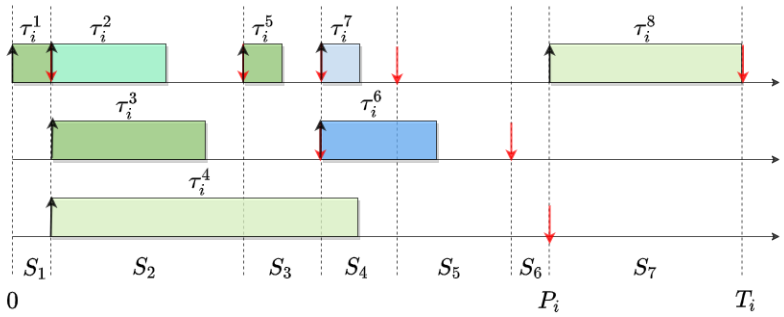


Figure 5.4: **Decomposed DAG with offsets and deadlines**

5.1.1 Terminology

Each section in the ideal schedule has a worst-case execution requirement e_j and we classify sections as either *light* or *heavy* based on the amount of parallelism in the section. The parallelism is quantified by the number of *threads*, θ, present. A section is *light* if

$$\theta < \frac{C_i}{T_i}, \tag{5.1}$$

and *heavy* when

$$\theta \geq \frac{C_i}{T_i}. \tag{5.2}$$

The contribution of the light sections to the *critical path* P_i is represented as P_i^{light}. Similarly, the contribution of the heavy sections to the total computation time is C_i^{heavy}.

$$P_i^{light} = \sum_{\forall j \in LS_i} e_j, \tag{5.3}$$

$$C_i^{heavy} = \sum_{\forall j \in HS_i} \theta_j e_j, \tag{5.4}$$

where LS_i and HS_i are the set of light and heavy sections respectively.

The intuition behind this segregation is that to ensure schedulability, sections with θ larger than the number of processors available should be treated differently, i.e., given more slack. We choose C_i/T_i as a threshold for sections because

C_i/T_i for a schedulable DAG is always less than m. Thus, if a light section has $\theta > m$, the whole DAG is unschedulable anyway.

5.1.2 Assigning Offsets and Deadlines

We assign each section a relative deadline d_j which is a time share of the period of the task T_i. The three scenarios to consider are those in which every section is *light*, every section is *heavy*, and when we have a mix of both light and heavy sections. For each of these cases, we try to formulate the deadline assignment such that guidelines **(G1)** and **(G2)** are followed. In the case that all sections are either light or heavy, we guarantee that the resulting decomposed task will follow the guidelines (G1) and (G2) for schedulability. We do this by showing the upper bound on densities of both whole sections Δ_j^s and threads Δ_j^t in a section. This is because the schedule only has one active section at a time and as such, the active workload at any time is made up of only one section. Deadlines are assigned in each of the three cases as described below.

Case 1: All sections are heavy (LS = $\emptyset$)

In the case that all sections are heavy, $C_i^{heavy} = C_i$ because every section is included in C_i^{heavy}. We assign deadlines for each section as:

$$d_j = \frac{T_i\ \theta_j\ e_j}{C_i} \tag{5.5}$$

This means that the relative deadline allocated to each section is a function of both the execution time of the section and the number of threads in the section. The density of any thread, $\Delta_j^t = \frac{e_j}{d_j}$. In this case,

$$\Delta_j^t = \frac{C_i}{T_i\ \theta_j}. \tag{5.6}$$

This density is maximised when θ has its smallest possible value and because a section is determined heavy when Equation 5.2 holds, the minimum possible value of θ is $\frac{C_i}{T_i}$, i.e,

$$\Delta_j^t = \frac{C_i}{T_i\ \theta_j} \leq \frac{C_i\ T_i}{T_i\ C_i} \leq 1, \tag{5.7}$$

showing that guideline G1 is obeyed. The density of the section on the other hand, is $\Delta_i^s = \theta_j \Delta_j^t$ and

$$\Delta_j^s = \frac{\theta_j\ C_i}{T_i\ \theta_j} = \frac{C_i}{T_i}. \tag{5.8}$$

Since we bound DAG tasks to obey the necessary condition, $\frac{C_i}{T_i} \leq m$, then section density is upper bounded at m thus obeying guideline G2.

Case 2: All sections are light (HS = $\emptyset$)

In this case, $P_{light} = P_i$ because every section is light and part of the critical path. Section deadlines are assigned as:

$$d_j = \frac{T_i\ e_j}{P_i}, \tag{5.9}$$

i.e, the relative deadlines are dependent only on the execution time of the section. The density of any thread is $\frac{e_j}{d_j}$ and in this case has the value

$$\Delta_j^t = \frac{e_j\ P_i}{T_i\ e_j} = \frac{P_i}{T_i}. \tag{5.10}$$

This density is maximised when P_i has its largest possible value and a necessary condition for the schedulability of a DAG on a unit speed processor is that the

length of its critical path is smaller or equal to its deadline. Therefore the maximum allowed value of P_i is T_i, otherwise there is no way to schedule this on a unit speed processor.

$$\Delta_j^t = \frac{P_i}{T_i} \leq \frac{T_i}{T_i} \leq 1, \tag{5.11}$$

thus obeying guideline G1. The density of the whole section is $\Delta_j^s = \theta_j \Delta_j^t$ and

$$\Delta_j^s = \frac{\theta_j \ P_i}{T_i}. \tag{5.12}$$

This density is maximised when the value of θ_j is maximised and because we consider a section to be light when Equation 5.1 holds, the maximum value of θ_j is $\frac{C_i}{T_i}$ which is $\leq m$ to fulfil the necessary condition. Thus,

$$\Delta_j^s = \frac{C_j \ P_i}{T_i \ T_i} \leq \frac{C_j \ T_i}{T_i \ T_i} \leq \frac{C_j}{T_i} \leq m \tag{5.13}$$

which obeys guideline G2.

Case 3: We have both heavy and light sections (LS $\neq \emptyset$ and HS $\neq \emptyset$) In tasks with a mix of heavy and light sections, the rules for deadline assignment are different for light and heavy sections. We prioritise the heavy sections by giving light sections a deadline that is exactly the value of their execution times, i.e. for a light section, $d_j = e_j$. Thus, the total time share occupied by light tasks is equal to P_i^{light}. The density of any thread is $\Delta_j^t = \frac{e_j}{e_j} = 1$ satisfying guideline G1. The density of the section remains $\Delta_j^s = \theta_j \Delta_j^t$. For a light section, its maximum value is m as follows from Equation 5.1 and the maximum thread density.

After light sections have been assigned time portions exactly equal to P_i^{light}, heavy sections are left to share $T_i - P_i^{light}$ thus deadlines are assigned as follows:

$$d_j = \frac{(T_i - P_{light}) \ \theta_j \ e_j}{C_{heavy}}. \tag{5.14}$$

Thread density is $\Delta_j^t = \frac{C_{heavy}}{\theta_j \ (T_i - P_{light})}$ and the density of a section is $\Delta_j^s = \theta_j \Delta_j^t = \frac{C_{heavy}}{(T_i - P_{light})}$. Maximising either of the density values is dependent on the relationship between C_i^{heavy} and P_i^{light} and this is in turn dependent on the structure of the DAG. As this is not information that is immediately apparent from the timing parameters, we rely on the intuition that after giving a light section only exactly what it needs, heavy sections have been given the best possible chance.

Upon computation of all relative deadlines of the sections in the ideal schedule, we assign offset of a sub-task as a sum of all the relative deadlines of its predecessors and deadline as the sum of the relative deadlines of all sections to which it belongs. For a node that has parts of its execution in sections k to r of the ideal schedule:

$$\begin{aligned} \phi_i^j &= \sum_{i=0}^{k-1} d_i, \\ D_i^j &= \sum_{i=k}^{r} d_i \ . \end{aligned} \tag{5.15}$$

After the decomposition step, every sub-task is then treated as its own task with the period T_i^j equal to the period T_i of the whole DAG, an offset ϕ_i^j relative to this period, and a relative deadline $D_i^j \leq D_i$. These parameters can be supplied to any of the sequential scheduling solutions presented in chapter 4.

Chapter 6

Implementing Multiprocessor Schedulers on Bare-Metal Hardware

In this chapter, we focus on the implementation of multiprocessor schedulers. In section 6.1, we explain our extension of FIFO-OT, a sustainable scheduler which we adopt for multiprocessor platforms and in section 6.2, we explain our hardware implementations of multiprocessor FIFO-OT and some other popular multiprocessor schedulers.

6.1 Sustainable Multiprocessor Scheduling

In the next section, we explain how we extend FIFO-OT which was previously defined for uni-processor platforms to multiprocessor platforms.

6.1.1 FIFO-OT for Multiprocessor Platforms

Perhaps the most intuitive way to ensure that an online schedule recreates a given offline schedule is to store a table containing each scheduling decision. The online scheduler then looks up this table at every decision point and dispatches the jobs according to the table. However, such tables typically require substantial amounts of memory which may not always be available in an *embedded environment*. There is a need for scheduling solutions that require much less memory.

In FIFO policy, jobs are dispatched according to their release order. Hence, the only way to force the FIFO policy to recreate a given schedule is to force the jobs to be released in the order we want them to appear in the FIFO schedule. Nasri et al. [35] have shown that this can be done by assigning offsets to individual jobs. Moreover, they showed that in the context of uni-processor platforms, most jobs do not require distinct offsets instead, they can share the same offsets as other jobs. They hence proposed a solution to form partitions of jobs that share the same offset and introduced an algorithm to minimise the number of such partitions. In our work, we deal with more than one processor and hence need new techniques to assign job offsets.

After assigning offsets, the memory footprint is reduced by only storing an offset table instead of a table of scheduling decisions. The process of generating an offset table begins from a schedule Φ. A non-preemptive schedule Φ for a processor is an ordered sequence of tuples $(J_{i,j}, s_{i,j})$ that assigns a start time $s_{i,j}$ to each job in a hyperperiod. In order to ensure that a FIFO scheduler will

be forced to schedule the given jobs in the order determined by schedule Φ, we try to assign a new release time $r_{i,j}$ to each job $J_{i,j}$ such that when it is put in a FIFO queue, it is scheduled in the right time slot and starts its execution no later than $s_{i,j}$.

Jobs in a schedule, Φ created for a multiprocessor platform also have a processor assignment in addition to the timing information. A multiprocessor schedule can allow migration and as such, all jobs of a task do not necessarily have to be executed on the same processor. To handle this, we allow for offsets to be larger than a period to force a processor to only run the jobs of a task that belong to it.

Algorithm 6.1 processes one task at a time, handling all the jobs of a task associated with a particular processor before moving on to the next. It uses a greedy approach to find the minimum number of *offset partitions* where an offset partition is a series of jobs of a task with the same offset value. Each job is initially assigned a release time, r, according to its starting time in Φ, i.e., $r_{i,j} = s_{i,j}(\Phi)$ in line 1 of algorithm 6.1. The algorithm then keeps track of potential offset intervals (POI) which is the range of offsets with which a job can be safely released. Two jobs can share an offset if their POIs intersect. The algorithm computes POI of the first job of any task as w and keeps track of all its neighbours, updating w with the intersection of the neighbouring POIs in w on line 13. In the event that the POI of a job $J_{x,y}$ does not intersect with w, jobs until $J_{x,y}$ are saved to a partition with the last w as their collective safe interval on line 16. The release times of all jobs in the partition are updated to match a chosen offset in the range and a new partition then begins from $J_{x,y}$ with w reset to $\text{POI}_{x,y}$ on line 19. In the end, we get offset partitions for each task from the perspective of a particular processor.

This process decouples the decisions of the processors and gives FIFO-OT the chance to perform with the efficiency of a partitioned scheduler. This is reflected in the offset tuning process. A job number relative to a processor is represented as $J_{i,l}$ while the absolute number of the job in the hyperperiod is $J_{i,j}$. If all jobs of a task do not belong to a processor, then the lth job of τ_i on a processor is not necessarily the jth job of τ_i in the hyperperiod. $(j - l)$ represents the number of jobs skipped from the perspective of a particular processor and the minimum offset of $J_{i,l}$ is then $(j - l - 1)T_i$. The latest possible time to release a job, $I^l_{i,j}$ is its starting time in Φ, $s_{i,j}$ and the earliest time it is safe to release a job, $I^s_{i,j}$, is after all its predecessors in Φ have been released. Therefore, a potential offset interval is computed as shown in Equation 6.1.

$$POI_{i,j} = [I^s_{i,j} - r^0_{i,l}, I^l_{i,j} - r^0_{i,l}] \tag{6.1}$$

The values for $I^s_{i,j}$ and $I^l_{i,j}$ are computed as:

$$I^s_{i,j} = \begin{cases} (j-1)T_i, & \text{if } R_{i,j} = \emptyset \\ max\{(j-1)T_i, max\{R_{i,j}\} + 1\}, & \text{if } R_{i,j} \neq \emptyset \wedge i < x \\ max\{(j-1)T_i, max\{R_{i,j}\}\} & \text{otherwise} \end{cases} \tag{6.2}$$

$$I^l_{i,j} = s_{i,j}(\Phi)$$

where $R_{i,j}$ is the set of all jobs scheduled on the same processor as job $J_{i,j}$ and with an earlier release time than job $J_{i,j}$,

$$R_i, j = \{r_{x,y} | r_{x,y} \leq r_{i,j} \wedge J^c_{x,y} == J^c_{i,j}\} \tag{6.3}$$

Since all the jobs of a task do not necessarily execute on the same processor, we add a final *padding* value to each task to ensure that the scheduler is properly reset at the end of a hyperperiod. For example, if the last job of a task on a

processor is job 4 and the task has 10 jobs in a hyperperiod, we pad the next arrival of the task on that processor such that it skips 6 periods. This padding value is computed as $(n_i - l_i)T_i$ where n_i represents the total numbers of jobs of τ_i in the hyperperiod and l_i represents the number of jobs of τ_i that belong to the processor in question. It reduces to 0 in the case that all the jobs of a task belong to a single processor.

Algorithm 6.1: Multi-processor Offset Tuning (τ, Φ)

Input: A task set τ and a corresponding feasible schedule Φ

$r_{i,j} \leftarrow s_{i,j}(\Phi)$ for $J_{i,j}$ in H;
$P = \{P^1, ..., P^m\}$;
for $c = 1$ *to* m **do**
  $P^c = \{P^c_1, ..., P^c_n\}$;
  for $\tau_i \in \tau$ **do**
    $k \leftarrow 1$;
    $l \leftarrow 1$;
    $P^c_i \leftarrow \emptyset$;
    $w \leftarrow [0, H]$;
    for $J_{i,j}$ *on processor* c *in* Φ **do**
      $l \leftarrow l + 1$;
      $\text{POI}_{i,j} \leftarrow [[I^s_{i,j} - r^0_{i,l}, I^l_{i,j} - r^0_{i,l}]]$ according to Equation 6.2 ;
      if $w \cap POI_{i,j} \neq \emptyset$ **then**
        $w \leftarrow w \cap POI_{i,j}$;
      else
        Add partition $p = \{J_{i,k}, ...J_{i,l-1}\}$ with offset w^s to P^c_i;
        $\forall$ jobs in p, $r_{i,x} \leftarrow (x-1)T_i + w^s$;
        $k \leftarrow l$;
        $w \leftarrow POI_{i,j}$;
    Add final partition $p = \{J_{i,k}, ..., J_{i,l}\}$ with offset w^s to P^c_i;
    $\forall$ jobs in p, $r_{i,x} \leftarrow (x-1)T_i + w^s$;
return P ;

6.2 Implementation on a Hardware Platform

In this section, we explain how we have implemented various schedulers on bare-metal hardware. We describe the implementation of a sustainable multi-processor scheduler as described in section 6.1.1 alongside some popular partitioned and global policies.

6.2.1 Multiprocessor FIFO-OT

In our implementation of FIFO-OT, a processor calls the scheduler at release events and upon completion of any job. Each time the scheduler is called, the job with the oldest release time is scheduled on the processor that made the call, i.e., in a FIFO order. We keep distinct offset tables for each processor to ensure that the processors **do not share scheduling data** and hence they can call their schedulers without a need for semaphores or mutual exclusion to protect global shared data.

We handle job releases by the scheduler itself, i.e., the scheduler is also responsible for updating the release times of future jobs based on the current time.

After each release of a job, the job number and next release is updated using the values stored in our offset table. We decided to not use software timers to handle job release events because anyway we would need to adjust the offsets manually according to our offset table. Our offset table entries contain an offset value and a job ID after which this offset no longer applies. The scheduler keeps track of the current offset in use for a task and updates it whenever the current job number is greater than the associated job ID. At the end of each hyperperiod, all offsets are reset to the first entry of the offset table.

6.2.2 Global Policies

Each processor calls the scheduler either at a job release event or a job completion event. However, since any processor can call the scheduler at any time, there might be multiple active instances of the scheduler function. As a result, the global shared variables that contain scheduling data (such as the ready queue, etc.) must be protected against race conditions using semaphores. We implement two variants of global scheduling namely, global EDF (**G EDF**) and global FP (**G FP**). In our **G FP** scheduler, we sort the task set by priorities during the initialization phase. We also maintain a ready queue with the same ordering. When a task has a pending job, its entry in the ready queue is set to 1 and when it does not have a pending job, its entry is set to 0. Thus, to select which job to run, the FP scheduler extracts the **highest set bit** in the ready queue with one instruction (in O(1)). Our **G EDF** scheduler maintains a similar ready queue but since task priorities are not fixed, the scheduler needs to keep track of the priority of the current job of each task and maintain a sorted list of the pending jobs. This adds to the implementation overhead of **G EDF**.

6.2.3 Partitioned Policies

We implement three partitioned scheduling policies: partitioned FP (**P FP**), partitioned EDF (**P EDF**) and partitioned CW-EDF (**P CW-EDF**). In these implementations, each processor has a distinct set of tasks and as such, there is no shared scheduling data. This removes the need for semaphores and each processor can freely access the scheduler. Our implementations of **P FP** and **P EDF** follow the same logic as **G FP** and **G EDF** descried above. **P CW-EDF** is a non-work conserving scheduler and has an idle time insertion policy. The decision to idle or not is also made within the scheduler.

Chapter 7

Experimental Evaluation

In this chapter, we experimentally evaluate the solutions introduced in chapters 4, 5, and 6. In section 7.1, we discuss results of scheduling sequential workloads. Section 7.2 deals with results of scheduling parallel tasks and in section 7.3, we show measured overheads of the implementation of our sustainable scheduler on actual hardware.

Abbreviation	Description
FF DBF Approx.	This refers to the first-fit partitioning using Fisher's DBF approximation test [7] as a fitting criteria (see Equation 3.3).
FF Davis	This refers to the first-fit partitioning using Davis' schedulability test for FP schedulability (see Equation 3.6).
FF Jeffay	This refers to the first-fit partitioning using Jeffay's test for EDF schedulablity introduced in [23] (see Equation 3.1).
FF CW-EDF	This refers to the first-fit partitioning, where each processor is scheduled by a sustainable scheduling policy. Schedulability is evaluated by building the CW-EDF schedule of one hyperperiod and checking if there is a deadline miss.
FF RM-FP	This refers to the first-fit partitioning, where each processor is scheduled by a sustainable scheduling policy. Schedulability is evaluated by building the rate-monotonic (RM-FP) schedule of one hyperperiod and checking if there is a deadline miss.
NPT	Our necessary test for partition-ability introduced in section 4.1.3.
LIQ	Liquid-path partitioning, our semi-partitioning solution introduced in section 4.2.
G CW-EDF	Our global non-work conserving scheduler introduced in section 4.3.1.
FED	Federated scheduling introduced in [26].
DirEDF	This refers to direct global EDF scheduling where we make a global EDF scheduler which is aware of precedence constraints.
G EDF	Global EDF scheduling.
DECOM1	Decomposition method according to [41].
DECOM2	Our proposed DAG decomposition method in Section 5.1.

Table 7.1: **Table of abbreviations**

7.1 Sequential Tasks

7.1.1 Task Generation

Our task generation method follows the description of an automotive benchmark application [25]. All task periods are chosen from {1, 2, 5, 10, 20, 50, 100, 200,

1000} (milliseconds) with non-uniform probabilities. This method has been adopted for task generation by many other works [35, 37, 33]. To generate n synthetic periodic tasks, we first pick n periods according to the probability distribution of periods in [25]. Next, we generate a set of n random utilisations that sum up to our target system utilisation for the task set. We do this by way of the *RandFixSum* algorithm from [43]. *RandFixSum* generates n random numbers that sum up to a given value. We then compute worst-case execution time and complete the properties of each task.

7.1.2 Experimental Setup

For our experiments, we generate 1000 task sets for each data point and vary the following parameters:

- **Number of processors**: Multiprocessor platforms come in a variety of configurations. While we limit our approach to homogeneous multiprocessor platforms, the number of processors is varied in experiments. Our experiments cover platforms with 2, 4 and 8 processors.
- **Utilisations**: Utilisation is a measure of how busy a system is. We generate all utilisations as percentages of the number of processors, m i.e $0 < U \leq m$. This is because task sets with utilisation greater than m are not feasible.
- **Number of tasks**: Finally, we vary the number of tasks to quantify the behaviour of the algorithms with respect to n. The number of tasks considered for each set of experiments is selected from the interval $[m + 1, 5m]$ where m is the number of processors.

Performance Metrics

Our main performance metric for sequential task sets is the **schedulability ratio** defined as the ratio of schedulable task sets to the total task sets generated. When we discuss partitioning, we also refer to **partition-ability ratio** and **non partition-ability ratio**. Partition-ability ratio is the ratio of task sets that can successfully be scheduled by a partitioning heuristic to the total task sets generated and non-partition-ability ratio is the ratio of task sets that are certainly not partition-able by any partitioning algorithm (hence are rejected by our necessary partition-ability test) to the total task sets generated.

7.1.3 Evaluation of Global CW-EDF

The impact of different heuristics on global CW-EDF schedulability. As mentioned in section 4.3.1, our global CW-EDF algorithm has two options for the look ahead step. We can either begin our look ahead from the processor with the earliest or latest busy time. In this section, we discuss the impact of both strategies. In our algorithm, we make a decision to begin the look ahead from the processor with the estimated earliest busy time. This decision was made following the intuition that the processor that gets access to the global scheduler is the one with the earliest busy time therefore, it will likely not be available to run jobs in the near future. In these experiments, we quantify the effectiveness of this heuristic (referred to as LatestCore in Figure 7.1) by comparing it with one that uses the processor with the earliest finish time (referred to as EarliestCore in Figure 7.1). Figure 7.1 shows that the latest finish time heuristic can only schedule 1% of task sets on 8 processors while the earliest finish time heuristic schedules 55% when we have 12 tasks. The same difference in performance is observed when we have 20 tasks on either 4 or 8 processors.

The impact of the number of tasks on global CW-EDF schedulability. In this set of experiments, we look into the impact of the number of tasks

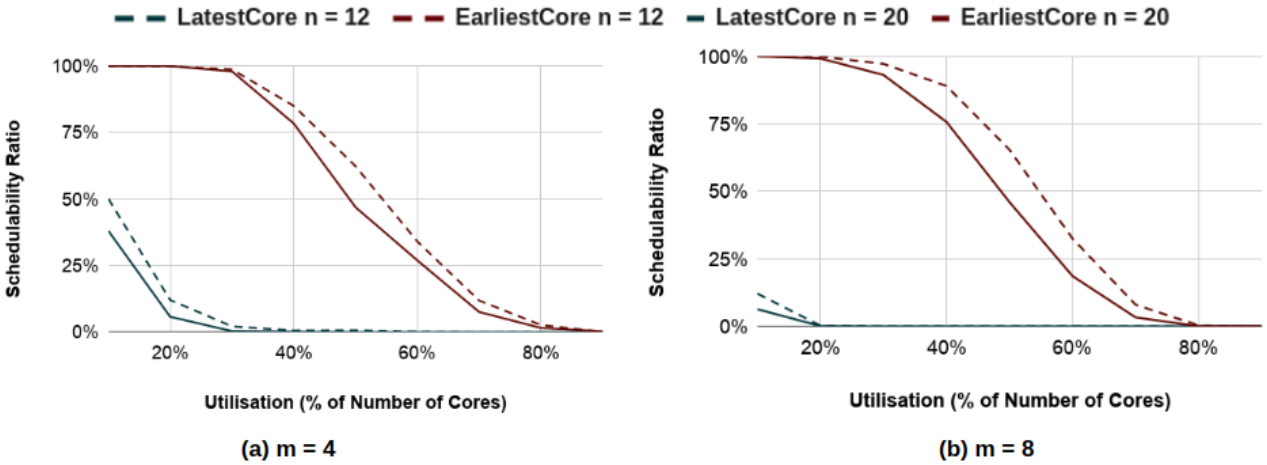


Figure 7.1: **The impact of different heuristics on global CW-EDF**

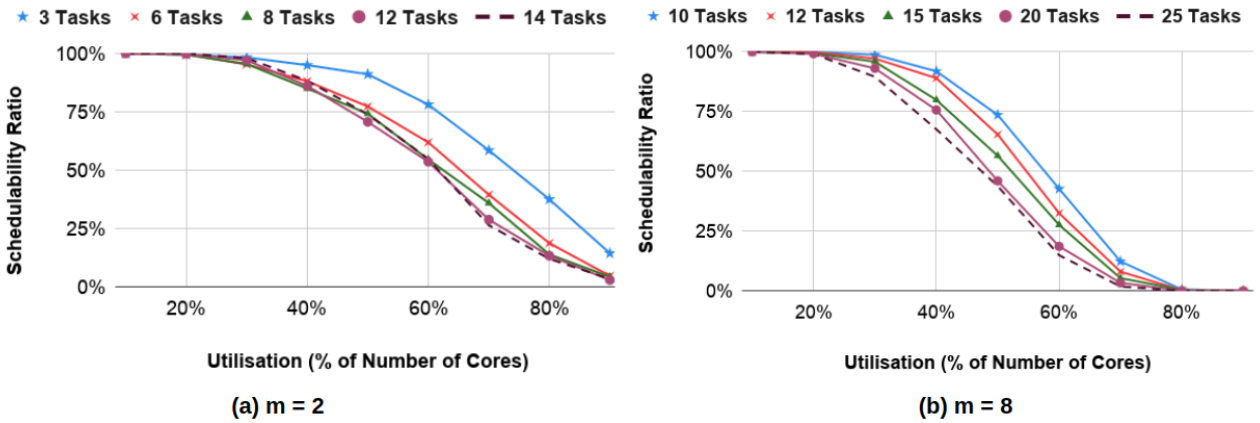


Figure 7.2: **The impact of the number of tasks on global CW-EDF**

on schedulability. We vary the number of tasks and utilisation value while keeping the number of processors constant. The performance is as shown in Figure 7.2 and we see that schedulability decreases with increasing number of tasks, however, this effect decreases as number of tasks increase. On 8 processors for instance, the average schedulability ratio of 10 tasks is 5% points higher than that of 15 tasks while the average schedulability ratio of 20 tasks is only 1% higher than that of 25 tasks.

The impact of the number of processors on global CW-EDF schedulability. Here, we look into the impact of varying number of processors on the schedulability of global CW-EDF. Tasks perform better with fewer processors but we must keep in mind that the utilisation values are expressed as percentages of the number of processors. With a smaller number of processors, the average utilisation share of each task is smaller than the case of a higher number of processors. From figure 7.3, the performance gap between 2 processors and 4 processors is 6% when we have 12 tasks and the performance gap is 1% between 4 processors and 8 processors. This suggests that the impact of number of processors decreases as number of processors increase.

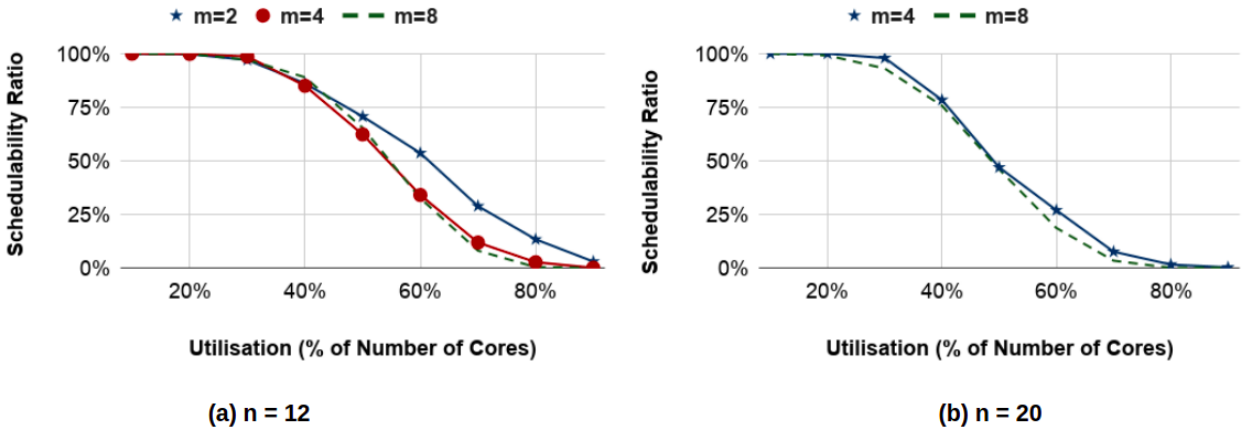


Figure 7.3: **The impact of the number of processors on global CW-EDF**

7.1.4 Evaluation of Partitioning Solutions

The goal of this section is to **(i)** quantify the pessimism of the existing heuristic partitioning methods, and **(ii)** find out the limits of the partition-ability, where it is impossible to partition a task set on a given number of processors.

Evaluation of partitioning heuristics. In this section, we evaluate the partitioning solutions for sequential tasks. We begin by looking at the relation between the success of partitioning heuristics and the schedulability tests they use to evaluate the fitness of a task to a partition. We focus on the first-fit partitioning heuristic because it is one of the better performing fitting heuristics as shown in [30] and corroborated by our data.

We use the abbreviations introduced in Table 7.1 to refer to the fitness tests that were evaluated, i.e., **FF DBF Approx.**, **FF Davis**, **FF Jeffay**, **FF RM-FP**, **FF CW-EDF**, and **NPT**. Note that **FF RM-FP** and **FF CW-EDF** are simulation-based tests, i.e., we simulate schedules for one hyperperiod using RM-FP and CW-EDF scheduling policies. EDF was excluded as it was found to make the same decisions as RM-FP for most of our task sets. This behaviour is corroborated by [35]. The performance of our proposed clique partitioning heuristic (in section 4.1.2) was the same as first-fit. To avoid cluttering the diagrams, we excluded this result from the diagrams.

Partitioning under **FF DBF Approx.** deteriorates very rapidly as number of tasks and utilisation increase and building a schedule gives us the highest schedulability ratio. Taking Figure 7.4(a) for instance, when we assume that we have a sustainable scheduler, e.g., using **FF CW-EDF**, we can schedule an average of 77% of task sets across all utilisation values. Using tests that do not assume this, e.g., **FF Jeffay** and **FF Davis**, we can only schedule an average of 59% of task sets across all utilisation values.

Quantification of pessimism in partitioning heuristics. The necessary test **NPT** as presented in Figure 7.4 serves as an upper bound that separates non-partition-able task sets from those that may or may not be partition-able. As it can be seen, with a larger number of processors (e.g., 8), partitioning heuristics that are based on sustainable scheduling (generated by **FF CW-EDF** or **FF RM-FP** are almost as good as an optimal partitioning strategy because they can find a schedule for 98% of task sets that pass the necessary test across all utilisation values (see Figure 7.4(b)). Other tests that are not based on sustainable partitioning like **FF Davis** and **FF Jeffay** can find a

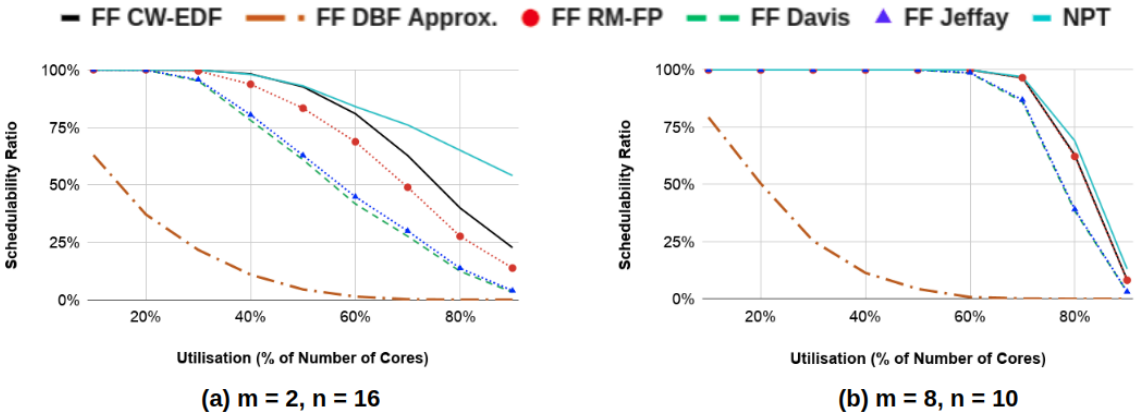


Figure 7.4: **Performance of first-fit partitioning with different schedulability tests**

schedule for 93% of task sets that pass the necessary test. Taking Figure 7.4(a), sustainable partitioning, i.e., using **FF CW-EDF**, we can schedule 91% of task sets that pass the necessary test while other partitioning heuristics can only schedule an average of 67% of task sets that pass the necessary test across all utilisation values. Hence, using sustainable partitioning increases schedulability ratio by 24 percentage points in this case.

7.1.5 A Comparison between Global, Partitioned, and Semi-Partitioned Solutions for Sequential Tasks

In this section, we plan to compare the best solution from each of the three main scheduling approaches: global, partitioned, and semi-partitioned. The goal is to understand which of these approaches provides a higher schedulability for sequential task sets. The data shown in the chart of Figure 7.5 compares the following **FF CW-EDF**, **FF Jeffay**, **G CW-EDF**, **LIQ** and **NPT** introduced in Table 7.1.

An interesting observation in Figure 7.5 is that partitioned scheduling gets better as number of processors and tasks increase irrespective of the test used. This is a contrast to global scheduling where increasing the number of tasks caused a large decline in performance (comparing Figure 7.5(a) and 7.5(b)). Figure 7.5(a) shows that when 10 tasks are scheduled on 8 processors, the average schedulability ratio of **FF CW-EDF** is 85% and it rises to 94% when there are 30 tasks in 7.5(b).

Another important observation is that task sets that fail the necessary test for partition-ability could still be feasible. Of all the methods compared, only liquid-path scheduling **LIQ** is able to achieve this as shown in Figure 7.5(a). The chance of scheduling non-partition-able task sets gets slimmer as the number of tasks and processors increase (comparing 7.5(a) and 7.5(b)). In the case of 8 processors and 30 tasks, we see that all the task sets generated passed the necessary test. The actual performance of algorithms also closely match this as all task sets are still schedulable at 70% utilisation and we can schedule half the tasks sets at 90% utilisation.

For the sake of readability of the diagrams, we only mention the confidence intervals in the text. In the scenarios presented in Figures 7.5(a) and 7.5(b), the largest observed confidence intervals with respect to the average schedulability

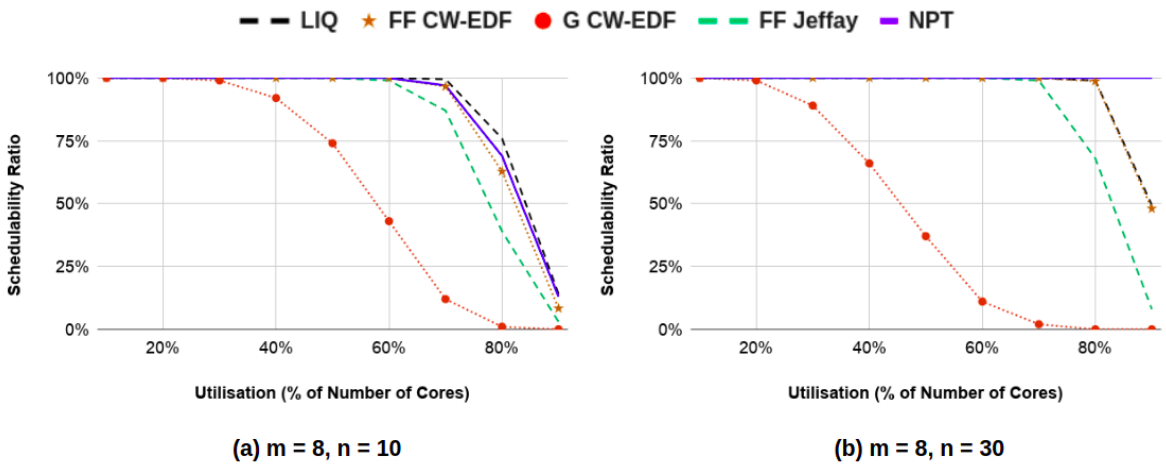


Figure 7.5: **A comparison between global, partitioned, and semi-partitioned scheduling for sequential task.**

ratio of each algorithm were ±4.7% and ±6.1%, respectively.

7.2 Parallel Tasks

7.2.1 Task Generation

We generate a DAG task set of size n by first generating n random utilisation values via the UUniFast algorithm [8] which generates random utilisation values that add up to a target value U. Periods are picked from the set $\{x.10^y | 1 \leq x \leq 9, 3 \leq y \leq 5\}$ with equal probability of occurrence. This method of period assignment covers three orders of magnitude and is in line with industrial task sets and synthetic task sets from other research works [39, 31, 42]. The WCET of the whole DAG is then computed from the utilisation and period values.

With regards to the internal structure of the DAG, we use the method in [31] which generates series-parallel DAG tasks with nested fork-joins by recursively expanding sub-tasks (nodes) to either terminal nodes or parallel sub-graphs until one of the limits of the DAG is reached. These limits are supplied as parameters to our generation algorithm and are (i) the maximum number of branches from any node, (ii) the maximum length of the critical path, or (iii) the maximum number of nodes in the DAG. Sequel to the generation of the DAG structure, WCET of the sub-tasks are assigned by again calling UUnifast with the WCET of the whole DAG and the number of nodes therein. Each node has a probability of either being a terminal node (p_{term}) or recursively expanded to have more branches ($p_{parallel}$). These values are described in Table 7.2 along with other tunable parameters of a DAG.

7.2.2 Empirical Results

In our experiments on DAG scheduling, we vary the same parameters of number of processors, number of DAG tasks and utilisation as in sequential tasks. Our performance metric is again, schedulability ratio.

A combined performance comparison of parallel task solutions. We evaluate schedulability of DAG tasks by comparing decomposition approaches with direct scheduling. Additionally, within the decomposition approach, we

Property	Description	Value
n	Number of DAG tasks in the task set.	Varied
Max Jobs Per Hyper-period	Upper bound on the number of jobs in the hyper period.	10,000
Max Branches	Maximum number of branches	3
p_{term}	Probability that a node is a terminal node, i.e., no more child nodes originate from this.	0.3
$p_{parallel}$	Probability that a node is a parent node.	0.7
p_{add_edge}	Probability that an edge exists between sibling nodes.	0.2
Max Critical Path Nodes	Maximum number of nodes in the critical path of the DAG.	50
Min Nodes	Minimum number of sub tasks in a DAG.	5
Max Nodes	Maximum number of sub-tasks in a DAG.	50

Table 7.2: **Tunable DAG properties**

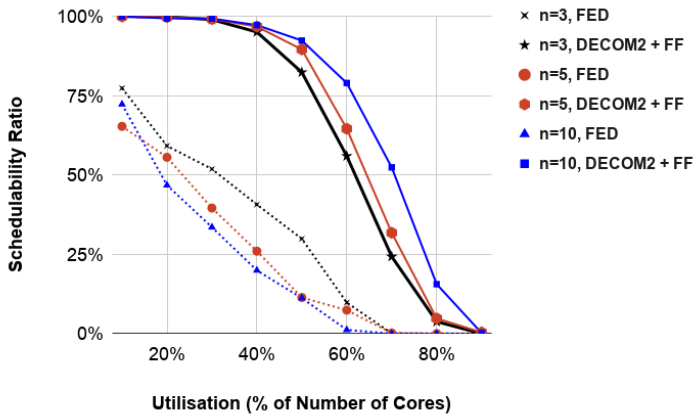


Figure 7.6: **The impact of number of tasks on DAG schedulability (m=4)**

compare the performance of globally scheduling the decomposed DAG tasks and partitioning them. Essentially, we compare the combination of decomposition and scheduling strategies. We compare **FED**, **DirEDF**, **DECOM1** and **DECOM2** as introduced in Table 7.1.

From Figure 7.6, we see that the performance of federated scheduling of DAG tasks decreases as number of DAG tasks increase. However, partitioned scheduling of decomposed DAG tasks improves as the number of DAG tasks increase. This is in line with our findings from sequential task experiments where partitioned scheduling improved with increasing number of tasks, i.e., **FF CW-EDF** and **FF Jeffay** in Figures 7.5(a) and 7.5(b). Additionally, in Figure 7.7, we see that the combination of our decomposition strategy (DECOM2) and first-fit partitioning yields the best schedulability ratio among all methods. Generally, the combination of decomposition and first-fit partitioning performs better than both direct scheduling of DAG tasks and global scheduling of decomposed DAG tasks.

Figure 7.7 shows 10 DAG tasks on 4 and 8 processors. In these scenarios, **DECOM2** with first-fit can schedule an average of 71% and 72% of task sets across all utilisation values on 4 and 8 processors, respectively, while DirEDF can only schedule an average of 20% and 18% on 4 and 8 processors, respectively. Furthermore, in the case of 4 processors, the decomposed globally scheduled solu-

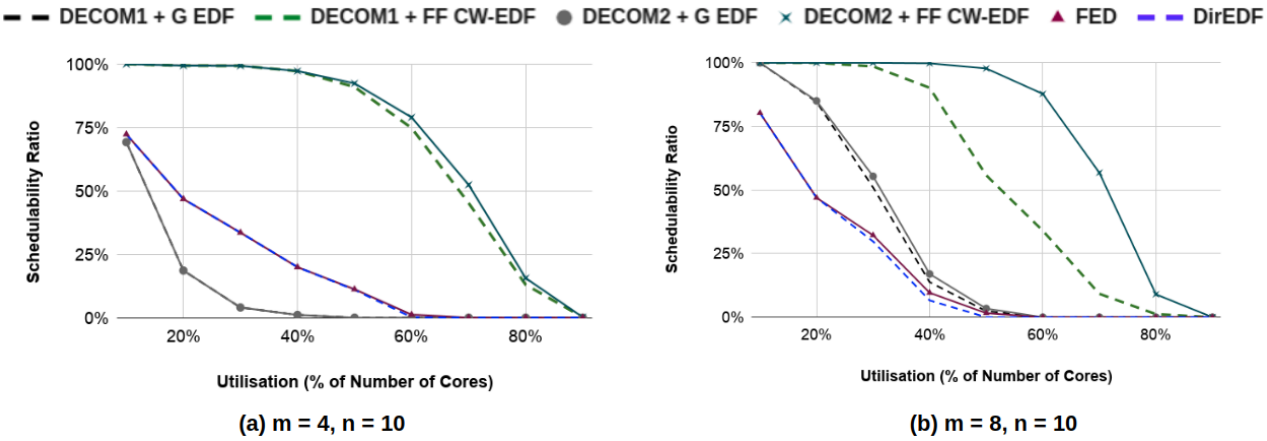


Figure 7.7: **A combined performance comparison of parallel task solutions**

tions perform worse than federated scheduling **FED** while federated scheduling performs worse in the case of 8 processors. This is because federated scheduling assigns an exclusive set of processors to each DAG that has a utilisation greater than 1. Thus, when there are less processors available for the same number of DAG tasks, each DAG has a higher utilisation value and therefore, more DAG tasks claim multiple processors to themselves leaving fewer processors for low-utilisation tasks to share and decreasing the overall schedulability.

7.3 Hardware Implementation

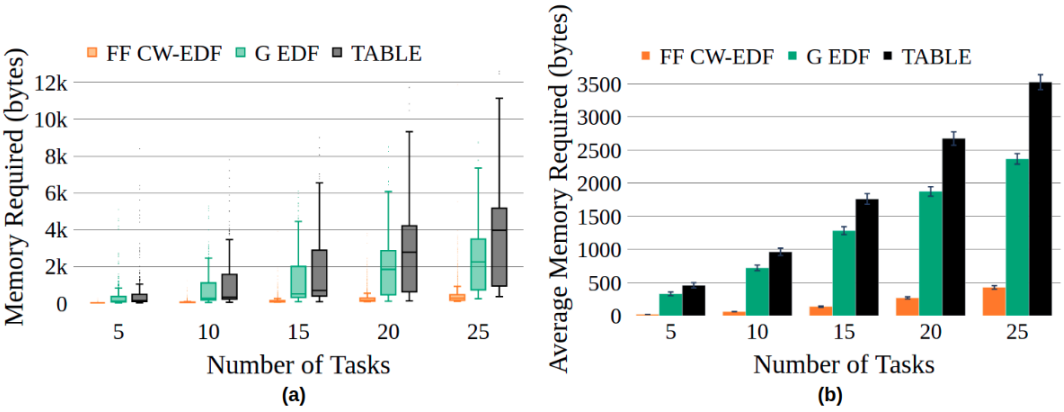
In this section, we evaluate the performance of the algorithms mentioned in section 6.2 in terms of dynamic memory consumption and runtime overhead. For our implementation, we have used the Raspberry Pi3 Model B. The board is equipped with the Broadcom BCM2837 64bit CPU with frequency set to 1.2GHz. There are 4 processors in total and 1GB of RAM. We do not use the operating system shipped with the board - *Raspberry Pi OS* - but build bare-metal implementations of our schedulers.

Memory Consumption

Offline scheduling solutions typically require some additional memory apart from the task parameters. In this section, we evaluate the memory consumption necessary to implement FIFO-OT on multiprocessor platforms.

Our implementation uses the automotive benchmark where task sets all have periods *(in ms)* $\in \{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$. This shows that the first limitation is that a period does not exceed 1000, therefore the largest offset ϕ, is also bounded by 1000. Consequently, to store our offset values, we only needed 16 bits. Each offset value in the table is accompanied by a job ID that denotes the last job in a consecutive set of jobs on which the offset applies. A job ID depends on the number of jobs in a hyperperiod because offsets get reset every hyperperiod. Our simulations limit this to 10,000 jobs in a hyperperiod, hence we need 16 bits to store a job ID. As such, each offset entry cost us 32 bits.

As a baseline, we compute the cost of storing the whole scheduling table. To store the whole table for one hyperperiod, we need to know the task ID and the arrival time. We assume that arrivals are reset each hyperperiod and as such 16 bits is sufficient to store this value. Task IDs are limited to 8 bits giving us



Error bars represent 95% confidence intervals.

Figure 7.8: **Memory consumption of offset tables in bytes**

a total of 24 bits per table entry. Additionally, we compare the offset tables generated by different policies. We compare the offset tables resulting from global EDF with tables generated for partitioned offline schedules. These are shown as **G EDF** and **FF CW-EDF**, respectively in Figure 7.8. The data presented is limited to task sets that are schedulable for each policy and in Figure 7.8, we see that offset tables cost less than a table scheduler irrespective of the policy used. However, offsets are much larger when there is a lot of migration in the schedule as can be seen in the large difference between the size of tables produced by global and partitioned policies. Offset tables built from *FF CW-EDF* never exceed 10% of the amount of memory required for a table scheduler across scenarios.

Runtime Overhead

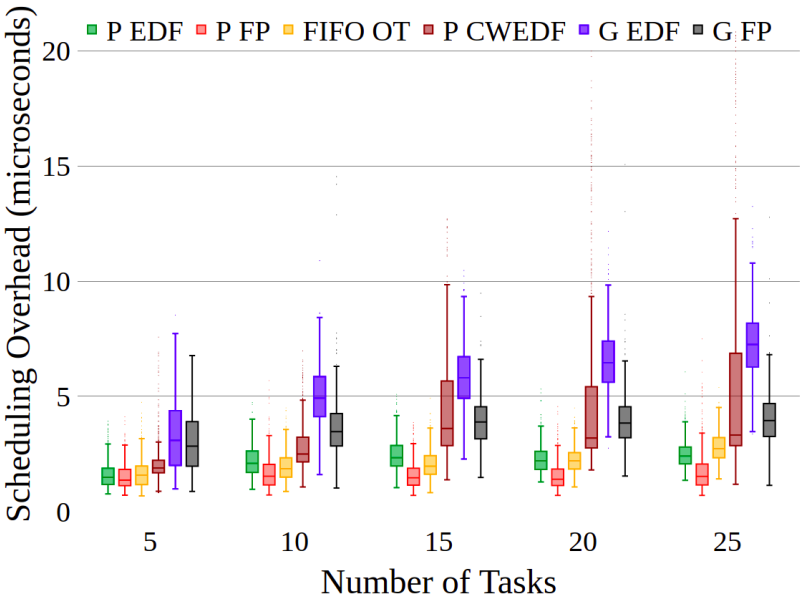
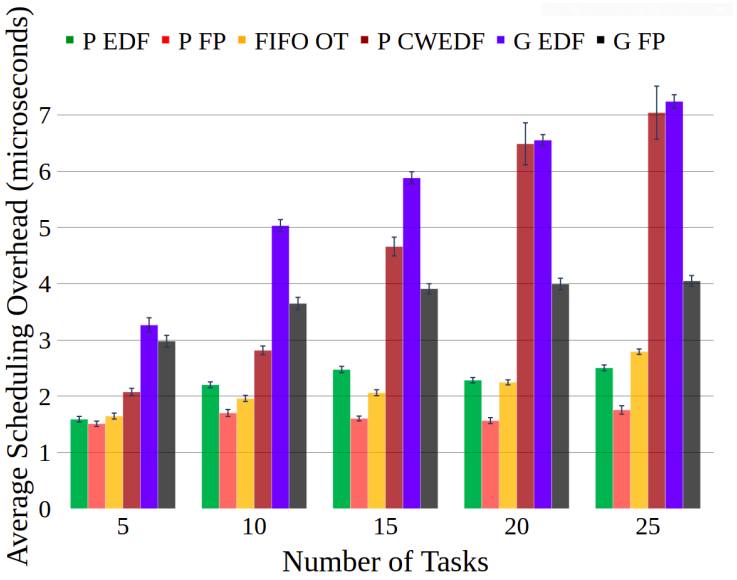


Figure 7.9: **Scheduling overhead**

We measure scheduling overheads as how long it takes for the scheduler to make a decision each time it is called. We keep track of time by counting the ticks of the system timer on the Raspberry Pi. Figure 7.9 shows the scheduling overhead of Global EDF **(G EDF)**, Global FP **(G FP)**, Partitioned EDF **(P EDF)**, Partitioned FP **(P FP)**, Partitioned CW-EDF **(P CW-EDF)** and finally our sustainable scheduler **(FIFO-OT)**. Our implementation has a runtime overhead that is 2% lower than partitioned EDF and 30% higher than partitioned FP scheduling across all task set sizes evaluated. While having lower-scheduling overhead, the schedulability ratio of partitioned FP without using our sustainable scheduling policy was 74% (across all utilisation values in Figure 7.5) while it was 90% when using our sustainable scheduling policy.

Additionally, the global schedulers incur the highest-average overheads. Our implementations of global FP and global EDF use semaphores to ensure mutually exclusive access to the scheduler and this makes their average overhead much larger than the partitioned counterparts because processors are often blocked on semaphores and have to wait for each other to be able to make scheduling decisions. FP has a smaller overhead than EDF in both the global and partitioned cases because it does not always have to loop through the task set to make a scheduling decision.



Error bars represent 95% confidence intervals.

Figure 7.10: **Average scheduling overhead**

Chapter 8

Conclusions

In this chapter, we present a summary of our contributions, our answers to the research questions posed in section 1.1, and proposals for future work.

8.1 Summary of Contributions

A necessary test for partition-ability. We have designed a necessary test that returns the minimum number of processors required to partition a given sequential workload in section 4.1.3. Our proposed test is based on identifying conflicts in a task set and constructing a graph model of the conflicts where nodes are tasks and edges exist between any two tasks that do not satisfy the necessary conditions for schedulability. Such a graph, gives us information on which tasks cannot be partitioned together on a single processor. Consequently, a clique in this graph represents a set of tasks that cannot be partitioned together and hence the minimum number of processors required to partition the task set is lower bounded by the size of the largest clique in the graph.

Sustainable partitioning for non-preemptive task sets. In section 4.1.1, we have shown the impact of using sustainable scheduling on the success of partitioning heuristics. We have further extended FIFO-OT for multiprocessor platforms so that it can not only be used by partitioned but also semi-partitioned and global scheduling policies.

Liquid-path scheduling. In section 4.2, we introduced the first semi-partitioning algorithm for non-preemptive tasks on multiprocessor platforms. Our algorithm works by assigning jobs to processors for one hyperperiod instead of a whole task, i.e, it is a job-based fitting technique.

A global non-work conserving scheduler. In section 4.3.1, we extended CW-EDF, a uni-processor non-work conserving policy, to design the first global non-work conserving scheduling policy for multiprocessor platforms.

Partitioned scheduling of decomposed DAG tasks. To use classical partitioning for parallel tasks, we first decompose each parallel task into a set of sequential tasks with precedence forced by offsets and relative deadlines. In chapter 5, we have introduced a DAG decomposition algorithm that is tailored to non-preemptive execution of DAG nodes. We propose that the resulting decomposed DAG tasks are scheduled by partitioning to increase schedulability and reduce the scheduling overhead.

Implementing various scheduling policies on a bare-metal hardware platform. We have implemented a set of schedulers on a bare-metal multiprocessor platform to evaluate their runtime overhead and memory consumption. These include existing partitioned and global schedulers and our sustainable multiprocessor scheduler as shown in chapter 6.

8.2 Research Questions

RQ1. *What is the impact of a sustainable scheduling policy on the success of partitioning heuristics when applied to non-preemptive sequential and/or parallel tasks?*

Across all the scenarios that were examined, sustainable partitioning gives us the highest-schedulability ratio. Taking Figure 7.4(a) for instance, sustainable partitioning can schedule an average of 77% of task sets across all utilisation values while other strategies can only schedule an average of 59% of task sets across all utilisation values. To apply sustainable partitioning to parallel tasks, we first decomposed the tasks to a series of sequential tasks. The schedulability ratio of sustainable partitioning of a decomposed parallel task was found to surpass both direct and decomposed global scheduling in all examined cases.

RQ2. *How can the pessimism of the existing partitioning heuristics be quantified given the lack of an optimal partitioning algorithm?*

We have built a necessary test that serves as an upper bound on non partition-able task sets. In most observed cases, our sustainable partitioning strategy is able to schedule 98% (see Figure 7.5(a)) of task sets that pass our necessary test while other solutions in the state of the art were able to schedule only 79% of task sets that pass the necessary test. In our best observed performance (6 tasks on 4 processors), sustainable partitioning is able to schedule 100% of the task sets that pass the necessary test while the best performance (10 tasks on 8 processors) of other strategies was 93%.

RQ3. *Can a semi-partitioned scheduling solution be designed to schedule task sets that cannot be partitioned?*

We have designed a semi-partitioned solution called liquid-path scheduling and observed that of all the methods compared, only liquid-path scheduling is able to schedule task sets that fail the necessary test for partitioning as shown in Figure 7.5(a).

RQ4. *How viable are non-work-conserving policies for global scheduling in terms of schedulability and overheads?*

Our global non-work-conserving policy was able to schedule an average of 65% of task sets on 2 processors and 51% on 8 processors across all utilisation values (see Figure 7.3). Its performance decreases with increasing number of processors and tasks (see Figures 7.2 and 7.3) and as such does not scale well.

RQ5. *What are the overheads (in terms of runtime and memory usage) of global, partitioned, and semi-partitioned scheduling on multiprocessor platforms?*

On average, our sustainable scheduler used 10% of the memory required for a table-based scheduling solution. Our implementation has a runtime overhead that is 2% lower than partitioned EDF and 30% higher than partitioned FP scheduling (see Figure 7.9). This overhead is the price we pay to have 25 percentage points more schedulability than the non-sustainable scheduling policies (see Figure 7.5).

In conclusion, our work has shown that partitioning solutions can provide much higher schedulability ratio with much smaller runtime overhead than the global solutions (even in comparison to our newly developed non-work-conserving global scheduling algorithm). We also showed that the proposed partitioning solutions are highly efficient and capable to schedule most of the partition-able task sets. Moreover, we have shown that it is indeed possible to surpass the schedulability of partitioning solutions by designing a semi-partitioned scheduling solution that uses sustainable schedulers on each core.

8.3 Future Work

So far, we have focused on periodic tasks and assumed no release jitter. In the future, it would be interesting to extend this work to include sporadic tasks and also extend the solutions to handle systems with release jitter.

Our necessary test can only answer whether a task set is certainly not partitionable. However, it does not tell if that task set could or could not be scheduled by *any* other scheduling approach such as global or semi-partitioned scheduling. Hence, a nice future work would be to make a test that provides a bound on the feasibility of a task set under non-preemptive scheduling in general.

Moreover, we have developed the liquid-path scheduling, a job-fitting technique that employs some constraint propagation. So far, we have not added backtracking or backjumping to the algorithm. In the future, we would like to see how much further we can push performance of the solution when using backtracking.

Finally, the comparisons in this thesis have been based on schedulability ratios but it can be further extended to see which methods minimise the number of processors necessary to schedule a task set.

Bibliography

[1] Karsten Albers and Frank Slomka. An event stream driven approximation for the analysis of real-time systems. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, pages 187–195. IEEE, 2004.

[2] Björn Andersson and Eduardo Tovar. Multiprocessor scheduling with few preemptions. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 322–334. IEEE, 2006.

[3] Steve Balacco. The embedded software and tools tarket: A market update and outlook. `https://www.slideshare.net/vdcresearch/embedded-software-and-tools-market-update-outlook`, 2009. Last accessed: May 28, 2020.

[4] Sanjoy Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems (RTS)*, 32(1-2):9–20, 2006.

[5] Sanjoy Baruah. Federated scheduling of sporadic dag task systems. In *IEEE International Parallel and Distributed Processing Symposium*, pages 179–186. IEEE, 2015.

[6] Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *IEEE International Real-Time Systems Symposium (RTSS)*, pages 159–168. IEEE, 2006.

[7] Sanjoy Baruah, Louis Rosier, and Rodney Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-time Systems (RTS)*, 2(4):301–324, 1990.

[8] Enrico Bini and Giorgio Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems (RTS)*, 30(1-2):129–154, 2005.

[9] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[10] Alan Burns. Scheduling hard real-time systems: A review. *Software Engineering Journal*, 6(3):116–128, 1991.

[11] Alan Burns, Robert Davis, Pengyu Wang, and Fengxiang Zhang. Partitioned edf scheduling for multiprocessors using a c = d task splitting scheme. *Real-Time Systems (RTS)*, 48(1):3–33, 2012.

[12] Giorgio C Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*, volume 24. Springer Science & Business Media, 2011.

[13] Yang Cai and MC Kong. Non-preemptive scheduling of periodic tasks in uni-and multiprocessor systems. *Algorithmica*, 15(6):572–599, 1996.

[14] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 421–433. IEEE, 2018.

[15] Edward Coffman Jr, Michael Randolph Garey, and David Johnson. Approximation algorithms for bin packing: A survey. *Approximation Algorithms for NP-hard Problems*, pages 46–93, 1996.

[16] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):1–44, 2011.

[17] Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems (RTS)*, 35(3):239–272, 2007.

[18] Ming Fan and Gang Quan. Harmonic-fit partitioned scheduling for fixed-priority real-time tasks on the multiprocessor platform. In *IEEE International Conference on Embedded and Ubiquitous Computing*, pages 27–32. IEEE, 2011.

[19] Ming Fan and Gang Quan. Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multicore platform. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 503–508. IEEE, 2012.

[20] Nathan Fisher and Sanjoy Baruah. The partitioned multiprocessor scheduling of non-preemptive sporadic task systems. In *ACM International Conference on Real-Time Networks and Systems (RTNS)*, pages 99–108. ACM, 2006.

[21] José Fonseca, Geoffrey Nelissen, Vincent Nelis, and Luís Miguel Pinho. Response time analysis of sporadic dag tasks under partitioned scheduling. In *IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, 2016.

[22] Nan Guan, Wang Yi, Qingxu Deng, Zonghua Gu, and Ge Yu. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *Journal of Systems Architecture*, 57(5):536–546, 2011.

[23] Kevin Jeffay, Donald F Stanat, and Charles U Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *IEEE Real-time Systems Symposium (RTSS)*, pages 129–139. US: IEEE, 1991.

[24] Xu Jiang, Xiang Long, Nan Guan, and Han Wan. On the decomposition-based global edf scheduling of parallel real-time tasks. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 237–246. IEEE, 2016.

[25] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[26] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, pages 85–96. IEEE, 2014.

[27] Jing Li, Zheng Luo, David Ferry, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Global edf scheduling for parallel real-time tasks. *Real-Time Systems (RTS)*, 51(4):395–439, 2015.

[28] Bill Lydon. Technical deep dive: Multi-core industrial controllers. `https://www.automation.com/en-us/articles/2016-1/technical-deep-dive-multi-core-industrial-controll`, 2016. Last accessed: June 12, 2022.

[29] Dániel Marx. Graph colouring problems and their applications in scheduling. *Periodica Polytechnica Electrical Engineering*, 48(1-2):11–16, 2004.

[30] Jaishree Mayank and Arijit Mondal. Non-preemptive multiprocessor scheduling for periodic real-time tasks. In *IEEE International Symposium on Embedded Computing and System Design (ISED)*, pages 1–6. IEEE, 2017.

[31] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, pages 211–221. IEEE, 2015.

[32] John W Moon and Leo Moser. On cliques in graphs. *Israel journal of Mathematics*, 3(1):23–28, 1965.

[33] Mitra Nasri, Sanjoy Baruah, Gerhard Fohler, and Mehdi Kargahi. On the optimality of rm and edf for non-preemptive real-time harmonic tasks. In *ACM International Conference on Real-Time Networks and Systems (RTNS)*, pages 331–340. ACM, 2014.

[34] Mitra Nasri and Bjorn B Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 12–23. IEEE, 2017.

[35] Mitra Nasri, Robert I Davis, and Björn B Brandenburg. Fifo with offsets: High schedulability with low overheads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 271–282. IEEE, 2018.

[36] Mitra Nasri and Gerhard Fohler. Non-work-conserving non-preemptive scheduling: Motivations, challenges, and potential solutions. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, pages 165–175. IEEE, 2016.

[37] Mitra Nasri and Mehdi Kargahi. Precautious-rm: A predictable non-preemptive scheduling algorithm for harmonic tasks. *Real-Time Systems (RTS)*, 50(4):548–584, 2014.

[38] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. A response-time analysis for non-preemptive job sets under global scheduling. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, pages 9:1–9:23. IEEE, 2018.

[39] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. Response-time analysis of limited-preemptive parallel dag tasks under global scheduling. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, pages 21:1–21:23. IEEE, 2019.

[40] Manar Qamhieh, Laurent George, and Serge Midonnet. A stretching algorithm for parallel real-time dag tasks on multiprocessor systems. In *ACM International Conference on Real-Time Networks and Systems (RTNS)*, pages 13–22. ACM, 2014.

[41] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D Gill. Parallel real-time scheduling of dags. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

[42] Maria A Serrano, Alessandra Melani, Marko Bertogna, and Eduardo Quiñones. Response-time analysis of dag tasks under fixed priority scheduling with limited preemptions. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1066–1071. IEEE, 2016.

[43] Roger Stafford. Random vectors with fixed sum. `https://nl.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum`. Last accessed: May 28, 2020.

[44] Texas Instruments. *Multicore Digital Signal Processor*, 2011. Data Sheet.

[45] Beyazit Yalcinkaya, Mitra Nasri, and Björn B Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1228–1233. IEEE, 2019.