Fine-grained Scheduling of Real-Time Recurrent DAG Tasks upon Multiprocessor Platforms

Master of Science Thesis in Embedded Systems

Shixun Wu

Delft University of Technology

TU Delft

Supervised by
Dr. Mitra Nasri
Pourya Gohari Nazari

Fine-grained Scheduling of Real-Time Recurrent DAG Tasks upon Multiprocessor Platforms

by

Shixun Wu

Master of Science Thesis in Embedded Systems

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

Shixun Wu

Supervised by
Dr. Mitra Nasri
Pourya Gohari Nazari
Wednesday 17th August, 2022

Author
Shixun Wu

Title
Fine-grained Scheduling of Real-Time Recurrent DAG Tasks upon Multi-processor Platforms

Supervisors
Dr. Mitra Nasri, Eindhoven University of Technology
Pourya Gohari Nazari, Eindhoven University of Technology

Graduation Committee
Dr. Neil Yorke-Smith, Delft University of Technology
Dr. George Iosifidis, Delft University of Technology
Dr. Mitra Nasri, Eindhoven University of Technology

Project duration
October 1, 2021 – July 20, 2022

MSc Presentation Date
August 22, 2022

Preface

More than two thousand years ago the Chinese philosopher Lao Tzu said in the Tao Te Ching that a journey of a thousand miles begins with the first step. Even the longest and most difficult journeys have a starting point, and a brave attempt to take the first step is necessary to solve problems no matter how big or small.

My master thesis started with the analysis of an anomaly in the scheduling of a task set. In the process, I kept encountering difficulties. When I could not see the problem, I tried to analyse it from the simplest example, starting from the first step and finding the key to the solution in the details. This thesis was also the starting point for my lifelong learning, which gave me the methodology to solve and analyse problems and the conviction and courage to take the first step.

I would like to especially thank Mitra Nasri for overseeing my thesis. Thank you for answering my various queries and for communicating with me and inspiring me when I was at a standstill. Thank you Pourya Nazari for taking the time to check my work and give me advice. I would also like to thank my family, who have always supported me in my studies and life, and I am glad that you are behind me. And to my friends and classmates who have been with me during this time, thank you for sharing my stress and bringing me happiness.

Shixun Wu
Delft, 12th August 2022

Abstract

With the strong demand for computing capacity in industrial applications and the rapid development of the hardware industry in recent years, multiprocessor platforms have been widely used in real-time embedded systems. The quest for performance has led to existing multiprocessor platforms often featuring complex interconnected hardware components and multiple levels of cache. This brings negative interference to the execution of tasks and challenges the predictability of real-time systems. However, existing non-preemptive execution models are an effective solution to eliminate these negative effects. In addition to this, the tasks that modern real-time systems process are becoming increasingly complex. Traditional sequential task models cannot cope with this situation and a stronger expressive model is required. A directed acyclic graph (DAG) that can express the complexity and parallelism of these tasks is a suitable model.

In this thesis, we focus on priority-based scheduling algorithms for multiple parallel DAG tasks on non-preemptive multiprocessor platforms, investigating, analysing, and improving existing global and partitioned scheduling algorithms.

We propose the concept of stacks to simulate a multi-processor platform and apply release-time tuning techniques based on its simulation of a task set in a hyperperiod. This allows us to impose tighter constraints on the execution of each job released from a task set in a fine-grained manner. We improved the existing priority-based global scheduling algorithm through the release-time tuning technique mentioned above. We also tried to construct a partitioned scheduling algorithm using the simulation results of stacks, i.e. so that all jobs are restricted to run on only one processor, consistent with the simulation results. Furthermore, to compare different scheduling algorithms more efficiently and to facilitate future researchers, we have developed the evaluation framework, a customisable experimental platform that includes DAG generation, application of scheduling algorithms, generation and analysis of the test results, allowing users to specify their experiments according to their goals.

Experiments with randomly generated workloads show that our improved algorithm consistently outperforms the state-of-the-art priority-based scheduling algorithm for different task graph structures and parameter configurations.

Contents

List of Figures

List of Tables

1 Introduction

The increased calculating capacity demand in industrial applications is driving the need for multiprocessor platforms in real-time systems [25, 8]. There's no denying that multiprocessor platforms have the potential to improve performance. Nevertheless, sufficient parallelisation of the software is required to effectively use the potential of the multiprocessor platform [38]. The rapid development of artificial intelligence in the twenty-first century has sparked widespread attention and significant investment in both the research and industrial communities, and it is undeniable that these applications have high computational demands, are highly parallelised, and handle large amounts of data obtained from one or more sensors [41]. Therefore, the performance optimisation of the multiprocessor platform has also become an important and practically valuable research direction when designing AI-enabled real-time systems [17, 3].

1.1. Objectives

Multiprocessor platforms are widely used in real-time systems [2]. However, driven by complex functionality requirements, real-time system functionality is no longer implemented as a single cyclic event. The directed acyclic graph (DAG) task is used to model system functionality dependencies, which is used to represent the relationships between a number of system functionalities (subtasks) [41]. For example, the functionality implementation from sensing the environment to controlling a car can be abstracted and transformed into a periodic DAG task [42]. In addition to this, non-preemptive DAG task scheduling is often deployed to avoid the preemption overhead costs associated with task migration and caching within multiprocessor [10, 37]. This means that sub-tasks of a DAG task are not allowed to be preempted during execution.

One of the key techniques for meeting the needs of real-time systems is to exploit parallelism on multiprocessor systems. In the case of DAG tasks, this means solving the problem of parallel scheduling of subtasks with interdependencies. This problem has been solved by investigating priority-based

scheduling algorithms [26, 21, 46, 27]. This technique generates a priority order for each subtask in a DAG task to restrict the order of their execution. Although existing research has developed several heuristic priority assignment algorithms using the temporal and structural characteristics of DAG tasks, we note that these algorithms do not respond well to systems with multiple DAG tasks.

In this work, we focus on the problem of scheduling multiple parallel DAG tasks on a non-preemptive multiprocessor platform. In the next subsection, we describe some of the open questions in existing scheduling algorithms.

1.2. Research Questions

The currently proposed priority-based scheduling algorithms are dedicated to analysing the structure of the DAG task based on nodes and using the information extracted from it to formulate the priority of each node [26, 21, 46, 27]. We note a trend in priority-based scheduling algorithms, that they all attempt to extract more information from the DAG structure as a reference for scheduling. From **ALAP** (As-Late-As-Possible) and **ASAP** (As-Soon-As-Possible) [26], which develop a priority based on only one metric, to **EOPA** (Execution Order Priority Assignment) [46], which classifies nodes and considers their dependencies. Even **GoSu** (graph convolutional task scheduler) [27] utilises powerful tools such as deep reinforcement learning to extract information about the nodes and structure of the DAG.

Moreover, it is worth noticing that the methods listed above are only about one DAG task, they cannot associate multiple DAG tasks. We want to consider all DAG tasks and the instance they release during an observation window (hyperperiod). We thought that a more fine-grained approach to analysing tasks would have a high potential to develop a more successful scheduling algorithm. This leads to the following research questions:

RQ1: *Can a more granular level analysis compare with only one DAG structure analysis for recurrent DAG tasks assist priority-based scheduling policy to improve schedulability?*

The priority-based scheduling algorithm only assigns a priority to each node, but we believe there is more scope for manipulating the execution of jobs released by each node. For example, we can adjust the offset and control the release time of the tasks. For recurrent DAG tasks, we can configure the offset for each node, which is the intra-DAG offset. To achieve this aim we ask the following question:

RQ2: *How to improve schedulability of priority-based scheduling policies for recurrent DAG tasks by using intra-DAG offset?*

Several studies have now proposed alternative solutions for scheduling recurrent DAG tasks [39, 23, 16]. With those approaches, a DAG is decomposed into a set of periodic tasks, each of which is composed of a node from the DAG. The time constraints of deconstructed tasks, particularly their offsets and deadlines, maintain the precedence restrictions. One of the methods [16] uses a partitioning algorithm to assign tasks to processors, so that

tasks cannot be migrated between processors. We thought about the potential of combining the partitioning method with a priority-based scheduling algorithm. This leads to the following question:

RQ3: *Does the partitioning method will improve schedulability of our priority-based scheduling policy?*

1.3. Contributions

To address the first research question (**RQ1**). We propose the concept of **stacks**, which is a container for jobs that follow a non-work-conserving policy. Furthermore, we propose a suitable **job batch placing algorithm** to place the jobs released by tasks in the appropriate places in the stacks. This process analyses the instances released by tasks and allows us to extract the job-level information from the stacks.

We apply the intra-DAG offset and propose the **reassembly stacking algorithm**, a new scheduling algorithm for parallel DAG tasks. For answering the second research question (**RQ2**) we evaluate existing scheduling algorithms against our algorithm.

To answer the research question (**RQ3**), we propose the **partitioned reassembly stacking algorithm**, a new partitioned scheduling algorithm for parallel DAG tasks. we evaluate the performance of this algorithm through empirical experiments.

In addition to this, we have developed an **evaluation framework** to facilitate the evaluation of the performance of different scheduling algorithms.

1.4. Organization

The remaining part of this thesis proceeds as follows: Chapter 2 will consider both the sources and methods of research which will include important concepts and definitions, as well as primary algorithms. Chapter 3 contextualises the research by providing a precisely defined model and proposing our research question. Chapter 4 gives a summary and critique of the state of the art. Chapter 5 present our solutions. Chapter 6 establishes a quantitative framework to evaluate multiple scheduling algorithms. Chapter 7 gives the evaluation results and the discussion of this, while conclusions and further research recommendation are discussed in Chapter 8.

2

Background

Chapter 2 will consider both the sources and methods of research which will include important concepts and definitions, as will as primary algorithms.

2.1. Concepts

We provide the necessary background information to help understand the rest of the paper.

Feasibility A task system is considered feasible if there exists a schedules meeting all timing constraints [6].

Scheduling Algorithm The scheduling algorithm of a real-time system is responsible for dispatching jobs for execution on an available processor [11].

Assume a processor can be assigned a job a a time, *i.e.*, no two jobs run at same time on the same processor. During this period of execution of a task or set of tasks, at each time unit, the algorithm selects one or several of the ready jobs to be executed on the processors according to a particular priority assignment.

Schedulability Let *A* denote a scheduling algorithm. A task system T is said to be A-schedulable, if *A* meets all deadlines when scheduling each of the potentially infinite different collections of jobs that could be generated by the task system, upon the specified platform [5].

2.2. Schedulability Tests

A schedulability test is required to be applied to check whether a task is schedulable using a special scheduling algorithm. A schedulability test accepts a task set, specified multiprocessor platform, and scheduling algorithm as input, then determines whether such task set is schedulable or not.

To compare scheduling algorithms, the schedulability test can be utilized. If all sets of tasks that are schedulable by algorithm B are also schedulable by algorithm A, but not vice versa, then scheduling algorithm A is dominants scheduling algorithm B. Schedulability tests are classified as follows:

Necessary tests If the task set *does not pass* the test, then it is certainly *not schedulable* by the given algorithm.

Sufficient tests If the task set *passes* the test, then it is certainly *schedulable* by the given algorithm.

Exact tests If the task set *passes* the test, then it is certainly *schedulable* by the given algorithm, if the task set *does not pass* the test, then it is certainly *not schedulable*.

2.3. Recurrent Task

A recurrent task is said to be periodic if successive jobs of the task are required to be generated a period T unit times apart. The recurrent tasks can be classified into the following categories:

Periodic task A recurrent task is said to be periodic if consecutive jobs of the task are required to be generated exactly a period apart.

Sporadic task A task is said sporadic if minimum interval between the generation of consecutive jobs of the task is period.

Aperiodic task An aperiodic task is one that occurs with no repetitions and does not have a specified period.

2.4. Deadline Type

The period and the deadline are typically the least two elements that define a recurrent real-time task. The relationship between these two can be categorised as follows.

Implicit deadline The relative deadline of the task is equal to the period of the task.

Constrained deadline The relative deadline of the task is small than or equal to the period of the task.

Arbitrary deadline The relative deadline does not subject to any constraint with regards to the period.

2.5. Multiprocessor Platforms

Multiprocessor platforms can be categorized depending on the relationship between the computing capabilities of the different processors as follows [4, 41].

Homogeneous architecture These are multiprocessor platforms with identical processors, in the sense that each processor in the platform has the same computing capability as the others.

Heterogeneous architecture These are multiprocessor platforms with distinct processors. The multiprocessor consists of a dedicated application processor, specialized for specific purpose, which is characterised by its own computing capabilities.

2.6. Multiprocessor Scheduler

Real-time scheduling techniques for multiprocessors are mainly classified into global scheduling, partitioned scheduling, and semi-partitioned scheduling.

Global scheduling In global scheduling, all tasks are stored in a global queue, and the same number of the highest priority tasks as processors are selected for execution.

Partitioned scheduling In partitioned scheduling, tasks are first assigned to specific processors, and the tasks are then executed on those processors without a migration. Using such an approach, a multiprocessor task scheduling is reduced into a set of uni-processor task scheduling after the tasks have been partitioned.

Semi-partitioned scheduling In semi-partitioned scheduling, most tasks are fixed to a specific processor as partitioned scheduling to reduce the number of migrations. A few tasks, on the other hand, are not restricted and may be migrated across processors to maximise the utilisation of available processors.

2.7. Execution Models

Depending on whether a process can be interrupted by another job, execution can be preemptive, non-preemptive, or limited-preemptive.

Preemptive execution Allows jobs executing on the processor to be potentially interrupted by the scheduler, which needs to execute other higher priority tasks and resume their execution at a later point in time.

Non-preemptive execution Such preemption is forbidden, once a job begins execution, it continues to execute until it has been completed.

Limited preemptive execution Preemption is allowed but various kinds of restrictions are placed upon the occurrence of preemption during scheduling.

2.8. Work-conserving and Non-work-conserving Scheduling Policies

Real-time scheduling algorithms can be categorized based on the insertion of idle time.

Work-conserving scheduling This scheduling method keeps the processor busy if there are jobs that have been submitted that have not been executed.

Non-work conserving scheduling This scheduling method allows the processor to be idle, even though there are jobs being submitted that have not been executed.

2.9. Bin Packing

The scheduling problem for real-time systems is to schedule the jobs released from a task set to the processor so that each job meets its time constraints as much as possible within the constraints of limited processor resources.

The bin packing problem, which is not the same but has similarities, has been extensively studied to find the minimum number of bins needed for a set of different volumes of items [24, 13]. Some heuristic bin packing algorithms can be referred to and derived for the study of real-time system scheduling problems.

Next-fit (NF) The NF keeps track of the bin containing the last item to be packed. After an item has been packed, the NF continues to pack its successive items. If it fits that bin then it is allocated to that bin, if not, it traverses from that bin until it finds the *next* one to fit.

First-fit (FF) FF traverses the bins in index order, packs the current item in the *first* non-empty bin in which it fits. If no such bin exists, FF packs the current item in an empty bin.

Worst-fit (WF) WF will pack the current item into an open bin with the smallest contents in which it fits. WF packs the item into an empty bin if there isn't an open bin that fits the current item. If more than one of these bins exist, WF chooses the one with the lowest index.

Best-fit (BF) BF will pack the current item into an open bin with the largest contents in which it fits. BF packs the item into an empty bin if there isn't an open bin that fits the current item. If more than one of these bins exist, BF chooses the one with the lowest index.

First-fit Decreasing (FFD) Under the FFD, the items are first sorted in order of non-increasing volumes, and then the FF algorithm is applied.

Best-fit Decreasing (FFD) Under the BFD, the items are first sorted in order of non-increasing volumes, and then the BF algorithm is applied.

2.10. Directed Acyclic Graph

This thesis focuses on the scheduling of directed acyclic graph (DAG) task models, therefore some basic DAG concepts will be presented here.

Definition 2.10.1 *A **Graph** is a pair of two sets G = (V, E), which V denotes a set of vertices (also called nodes) and $E \subseteq V \times V$ denotes a set of edges, each edge connects two vertices in such graph.*

Definition 2.10.2 *A **directed graph** is a graph in which edges have an orientation.*

Definition 2.10.3 *A graph is an **acyclic graph** if it does not contain any cycles.*

By the above definition, we can clearly express the concept of a DAG, a graph whose edges have orientation and which does not contain any cycles. The figure2.1 presents an example directed acyclic graph, the number in vertices is the index.

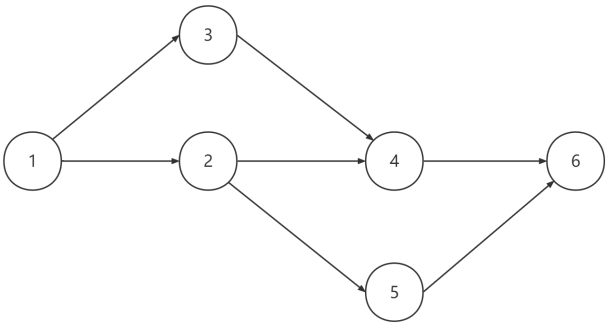


Figure 2.1: An example of Directed Acyclic Graph

There are also a few concepts that deserve to be mentioned to help us better summarise the properties and characteristics of a DAG.

Definition 2.10.4 *A vertex with no incoming edges is called a **source vertex** (node), a vertex with no outgoing edges is called a **sink vertex** (node).*

The v_1 in figure 2.1 is a source vertex and the v_6 is a sink vertex. We assume that a DAG has exactly one source vertex (denoted v_{src}) and one sink vertex (denoted as v_{sink}).

Definition 2.10.5 *In a DAG, if there exists edge from v_i to v_j, denote $(v_i, v_j) \in E$, then v_i is a predecessor of v_j, and v_j is a successor of v_i.*

Definition 2.10.6 *If there exists a path from v_i to v_j, then v_i is an ancestor of v_j, and v_j is a descendant of v_i.*

In this thesis we use $pred(v)$, $succ(v)$, $ance(v)$, and $desc(v)$ to denote predecessors, successors, ancestors, and descendants of vertex v respectively.

Definition 2.10.7 *A **path** in a DAG starting from vertex v_0 and ending at vertex v_k is a sequence of vertices $(v_0, v_1, v_2..., v_k)$, such that $\forall i \in [0, k), (v_i, v_{i+1}) \in E$.*

Definition 2.10.8 *A **complete path** is a path starting from the source vertex and ending at the sink vertex.*

Definition 2.10.9 *The **topological ordering** of a DAG is a linear ordering of all its vertices such that if G contains an edge from v_i to v_j, v_i comes before v_j in this ordering [14].*

3

System Model and Problem Definition

This chapter introduces our task model and defines the scope of this work. We start in section 3.1 with a description of the parallel DAG tasks model with some essential concepts. The following sections from 3.2 to 3.4 elaborate other properties of the system model. In section 3.5 we present the research problem of this work.

3.1. Parallel DAG Task Model

The directed acyclic graph task models have been proposed by Baruah *et al.* [6]. This task model uses a DAG to abstract the parallel tasks of a real-time system, therefore all the concepts introduced in the previous section 2.10 will be inherited in this task model.

3.1.1. Task

A finite set of recurrent tasks generates the workload in the real-time system. There are multiple ways to represent a recurrent DAG task but we give a clear definition in this thesis as follows.

A **set of tasks** is denoted as Γ, it contains a collection of n tasks, indicate as

$$\Gamma = \{\tau_1, \tau_2, ..., \tau_i\}, i \in [1, n].$$

A **recurrent parallel DAG task**, hereafter called a **task** in this report, generates a potentially unbounded sequence of workload.

We use three the following format to specify a task, as a tuple,

$$\tau_i = (G_i, T_i, D_i), i \in [1, n],$$

where i is the index of a task, n is the number of tasks in that set, G_i is a directed acyclic graph, and T_i is the period, D_i is the relative deadline of

this task. A recurrent task is said to be **periodic** if the successive workloads of the task are required to be generated a T times units apart. Tasks in a task set usually have non-identical periods, which invokes the concept of hyperperiod.

The **hyperperiod** is the minimum time interval after which the release pattern repeats itself when applying offline scheduling algorithms.

The hyperperiod denoted by H, the release pattern in $[0, H]$ is the same as that in $[kH, (K+1)H]$ for any integer $k > 0$. The hyperperiod is determined as such least common multiple of the periods for a set of periodic tasks, that are synchronously activated at time $t = 0$. Hyperperiod of task set Γ:

$$H(\Gamma) = lcm_{\tau_x \in \Gamma} \{T_x\} .$$

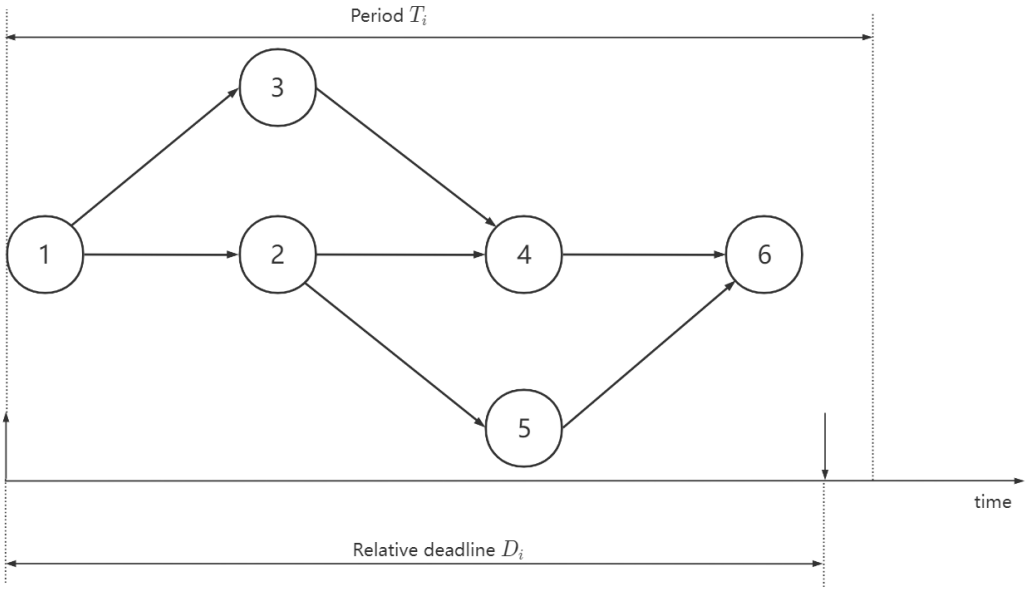


Figure 3.1: An example of a recurrent parallel DAG task model

Figure 2.1 gives an example of DAG tasks model. Graph G specifies the attributes and relationships of the internal vertices of this task. The form of G is $G = (V, E)$ and each $v \in V$ represent a sequential program (typical example is to run a piece of code). Each $v \in V$ is can be characterized by execution time and priority. In most cases we cannot accurately predict the execution time of each job. The job may run on different data, following different decision paths, having different system states, etc [11]. We bound this time using worst-case execution time (**WCET**) and best-case execution time (**BCET**). The edges denote dependencies between vertices, if there exists an edge $(v_i, v_j) \in E$, means vertex v_j depends on v_i, that v_i must be complete before v_j starts execution.

All vertices in a DAG can be represented as

$$v_k^i = (\beta_k, \omega_k, p_k, T_i, D_i), k \in [1, q], i \in [1, n],$$

which q is the number of vertices, β_k is BCET, ω_k is WCET, and p_k is the priority of this vertex respectively.

In section 2.10 we introduced path and complete path, now we can calculate the length of a path, with execution time. The length of a path is the sum of the execution time of all vertices along the path. Such execution time refers to WCET in this report. Here, we can describe an important attribute of a DAG, a **critical path** is the longest complete path in DAG, we denote as $CP(\tau)$.

3.1.2. Instance and Job

Instance and job are important concepts when we go on to discuss how tasks generate a series of workloads for processors.

Each release of a DAG task is called an **instance**. A DAG consists of several vertices, and an instance does not contain just one workload. We call a workload corresponding to a vertex a **job** .

An instance consists of jobs belonging to the same task. Figure 3.2 represents two consecutive instances of example tasks. The task $\tau_1 = (G_1, T_1, D_1)$, its DAG has four vertices, therefore its instance 1 and 2 both contain four jobs.

Job denotes as $J_{j,k}^i$ which k is the corresponding vertex index. The width of the blue rectangle in the diagram indicates the WCET of a job. All WCETs within an instance may be different, but the corresponding jobs in two instances belonging to the one task, e.g. $J_{1,4}^1$ and $J_{2,4}^1$, are the same.

We denote the j-times released instance of tasks i as

$$I_i^j = \left\{ J_{i,1}^j, J_{i,2}^j, ..., J_{i,k}^j \right\}, i \in [1, n], j \in [1, H(\Gamma)/T_i], k \in [1, q].$$

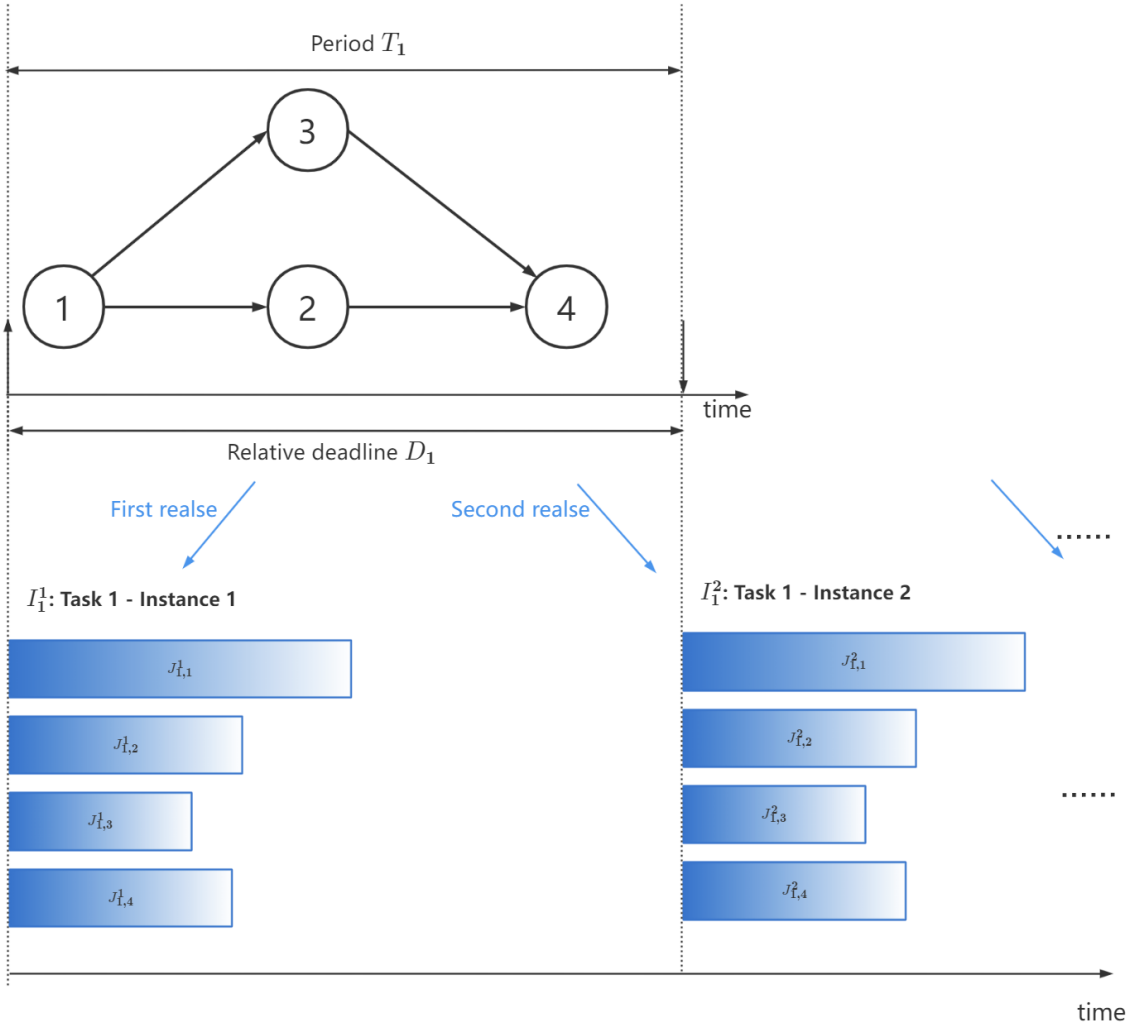


Figure 3.2: An example of instance

Here figure 3.3 gives a detailed description of the jobs. The period and deadline are attributes of the task, but they are directly related to the instance and job. The period **T** denote the amount of time between two consecutive instances of a task τ, which means all vertices $v \in V$ are released. If a task is released at time-instant t, then the next release of it is at time-instant $t+T$. The relative deadline **D** denotes the time constrains of a task. If all vertices $v \in V$ of a task τ are released at time-instance t, then all of those jobs must complete execution before time-instance $t + D$.

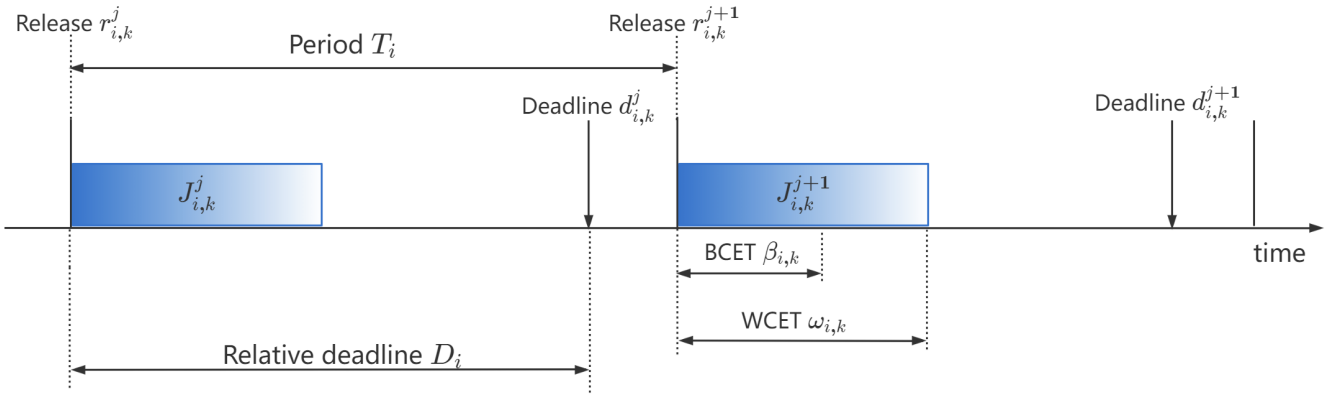


Figure 3.3: An example of jobs

Inheriting the previous representation we subscripts the job as $J^j_{i,k}$, the index j denotes this job belongs to j-times release instant I^j_i, i indicate which task this job belong to, and k is the corresponding vertex index. Unless otherwise stated, thereafter the meaning of j, i, k will be followed. It is easy to understand that, the vertex v^i_k corresponds to this job, its next released job is denoted as $J^{j+1}_{i,k}$.

The absolute deadline $d^j_{i,k}$ is a time-instant that is equal to the corresponding release time r^j_i plus relative deadline D_i. The BCET $\beta_{i,k}$ and WCET $\omega_{i,k}$ are used to bound the execution time of a job. Here we can give the format of job.

$$J^j_{i,k} = (r^j_{i,k}, \beta_{i,k}, \omega_{i,k}, d^j_{i,k}, p_{i,k})$$

The priority affects the choice of eligible jobs for execution, which brings priority-based scheduling [21].

We say a job is **eligible** at a certain time instant is all its predecessors in the instance it belongs to have finished and thus this job can immediately execute if there are available processors.

Within an instance, there are also dependencies between jobs. We assume that this inherently follows the relationship between vertices within the DAG. If $v^i_l \in pred(v^i_k)$, then $J^i_{j,l} \in pred(J^i_{j,k})$.

Formally, we assign priority p_k to v_k of the DAG, the respective job also inherits the priority. We say job $J_{j,k}$ has higher priority than job $J_{j,l}$ if $p_k < p_l$. In other words, smaller numbers representing higher priority.

3.2. Multiprocessor Model

The system model assumes the multiprocessor platform to be **homogeneous architecture** described in section 2.5, constructed by m identical processors, denoted as $P_1, ..., P_m$.

3.3. Scheduling Model

The scheduling model in this thesis is **priority-based scheduling**. Such scheduling gives a complete priority order to all vertices of a DAG task, a job released by a vertex still inherits the priority. At any time instant at run time, the scheduler always chooses the highest-priority eligible job for execution.

3.4. Execution Model

The execution model in this thesis is **non-preemptive execution** described in section 2.8.

3.5. Problem Definition

Here we can formulate the problem founded on the clear model definition above. We will make some assumptions to further clarify the scope of the research with the problem definition.

Given a recurrent parallel DAG task set Γ, defined in section 3.1, such task set contains n tasks with implicit deadline, assign priority and release times to each job in a hyperperiod upon multiprocessor platform defined in 3.2 which complies with the executive model defined in 3.4, that a priority-based scheduling algorithm can successfully schedule this task set Γ, which means every task respects its time constrain over an observation window as large as a hyperperiod while all jobs have execution time variation defined in 3.1.2.

4

Related Work

Existing scheduling methods and solutions that are consistent or relevant to this research direction are presented in this chapter. Based on the idea of algorithm development, we categorize them into groups and describe their basic methodology as well as their characteristics in section 4.1. Then in section 4.2, we introduce the real-time analysis method, which is relevant to how we evaluate a scheduling algorithm.

4.1. DAG Scheduling

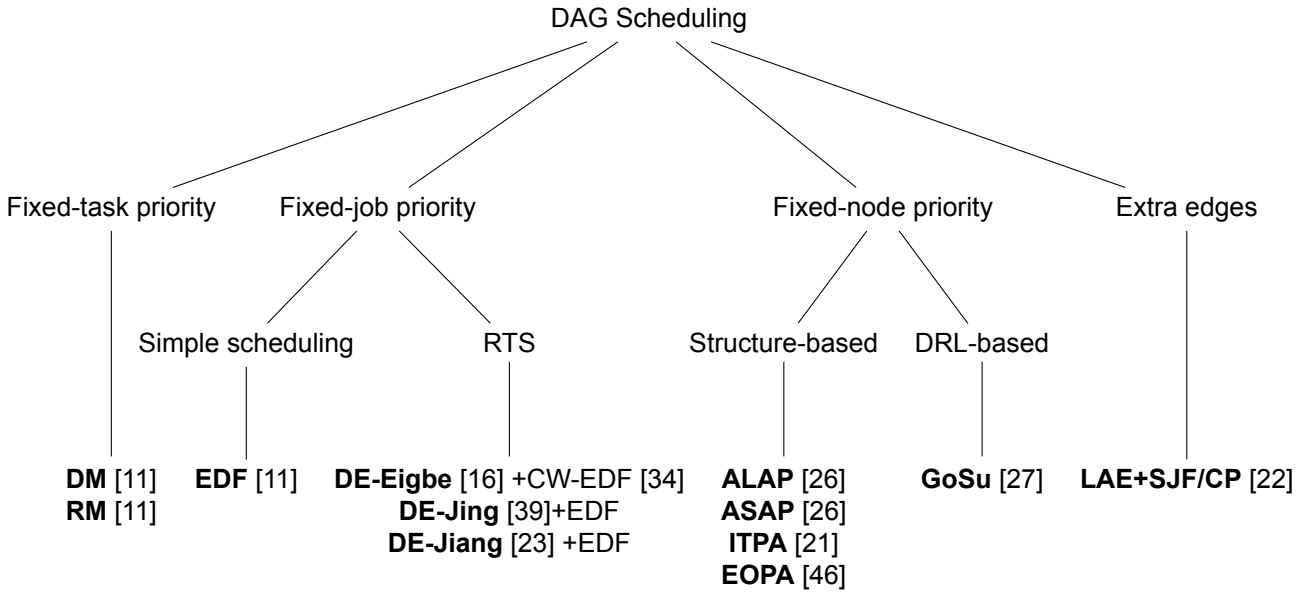


Figure 4.1: DAG scheduling algorithms

Here in figure 4.1 we propose a different classification from section 2.6.

We take the perspective of the fineness of the assignment of priors rather than that of a multiprocessor platform, which is in line with our research thinking.

4.1.1. Fixed-task Priority

The fixed-task priority scheduling algorithm assigns a priority to each DAG task. The Rate Monotonic (**RM**) priority assignment is a simple and famous rule that assigns priorities to each task according to their request rates [11]. Specifically, tasks with higher request rates, that is, with shorter periods, will have higher priorities.

The Deadline Monotonic (**DM**) assigns a priority to each DAG task according to the relative deadline of tasks [11]. The task with a lower relative deadline will be assigned higher priorities. This algorithm is an extension of RM, where tasks can have relative deadlines less than or equal to their period. In the implicit deadline case where the deadline of a task is equal to its period, , the DM and RM are the same.

4.1.2. Fixed-job Priority

Simple Scheduling

The fixed-job priority scheduling algorithm assigns a priority to each job of the task. The most widely used is the earliest deadline first (**EDF**) [11], which assigns priority to each job according to its absolute deadline, with the earlier the job needs to be completed the higher the priority.

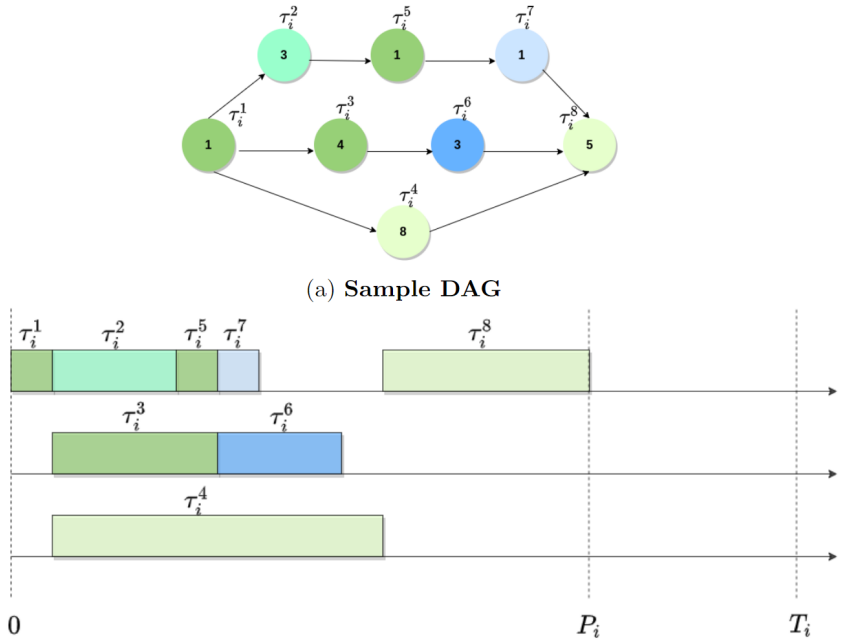
Reconstruct Task and Scheduling

The Reconstruct task and scheduling (**RTS**) refers to the decomposition-base scheduling. This method decomposes a DAG into a collection of periodic tasks, each of which is made up of a node from the DAG. Precedence constraints are maintained by the time constraints of decomposed tasks, especially their offsets and deadlines, as a consequence the decomposition is done.

Several studies [39, 23, 16] have explored and validated the potential of decomposition-base scheduling. The general approach of decomposition is to first assume the existence of an infinite number of processors and then build the ideal scheduling, i.e. the timing diagram, as shown in figure 4.2. Such ideal scheduling indicates the earliest start time and the earliest end time of each node (considering WCET). Based on this, the tasks are decomposed in various ways.

Jing *et al.* [39] propose a decomposition method **DE-Jing**, drawing a vertical line at every time instant where a node starts or ends. The task may have different degrees of parallelism in various node segments. The deadline of a node is then determined by allocating time to each segment, followed by the addition of all the times allocated to segments. After decomposition, the *non-preemptive global* EDF is applied to schedule the task set.

Jiang *et al.* [23] propose a similar solution **DE-Jiang**, dividing segments in a different way than the previous one, where tasks are divided into several segments by the earliest start timer of each node. The pioneering feature is



(a) **Sample DAG**

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

Figure 4.2: Ideal schedule of a DAG [16]

distinguishing segments into light and heavy segments, the heavy segments should have more laxity. The *non-preemptive global* EDF is still applied to schedule the decomposed task set.

Eigbe *et al.* [16] give a method that introduces a new concept *section*, dividing the ideal schedule into multiple sections, the splitting point is where each node ends its execution. For each section, a relative deadline is assigned that is greater than or equal to the execution time of the section. This means that several slacks are created between sections, as shown in figure 4.3. This decomposition method **DE-Eigbe** also inherits the concept of light and heavy sections, on the base of which each decomposed task is assigned offset and deadline. In addition to this, they have applied a new extension called FIFO with offset tuning (**FIFO-OT**) [33]. The key of this technique is to use a FIFO policy but adjust the offset of the work, i.e. change the release time. The **FIFO-OT** itself is not used to create a schedule, but rather to assist the scheduling algorithm to recreate an offline schedule at its runtime. For the scheduler, they have chosen to adopt **CW-EDF** [34], a non-work conserving conserving schedulers, which has been proven to be superior to EDF. They have also implemented **CW-EDF** to schedule the **DE-Jing** decomposed task set to compare the two decomposition methods.

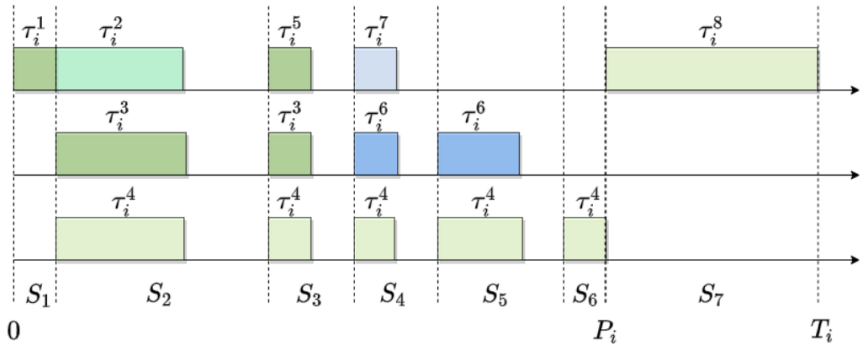


Figure 4.3: Ideal schedule with sections and their assigned slack [16]

4.1.3. Fixed-node priority

While scheduling DAG tasks, it is possible that at some time point many jobs are eligible for execution, and the number of eligible vertices is more than the number of available cores. The fixed-node priority scheduling algorithms are intended to overcome this uncertainty by utilising a special execution order.

Structure-based

The structure-based scheduling algorithms assign priority to each node by extracting and analysing the holistic structural information of the graph and the attributes of the nodes.

Two frequently used attributes for assigning priority are top-level (t-level) and bottom-level (b-level) [1]. The *t-level* of a node v is the length of the longest path (there can be more than one longest path) from the source node to node v. Here the length of a path is the sum of the execution of all nodes along the path. As such, the *t-level* highly correlates with node v's earliest start time. A scheduling algorithm called **ASAP** (As-Soon-As-Possible) employs the t-level [26]. It assigns priority to each node by calculating the t-level. The node with the smallest earliest start time obtains the highest priority.

The *b-level* of a node v_i is the length of the longest path from v to sink node, denote as b_i. The *b-level* is bounded from above by *the length of a critical path* (T_{CP}). The scheduling algorithm **ALAP** (As-Late-As-Possible) employs the b-level [26]. The priority of a node v is calculated by $CP(\tau) - b_i$, which gives higher priority to nodes with larger b-level.

He *et al.* proposed a scheduling heuristic **ITPA** (Intra-Task Priority Assignment) [21]. Such heuristic calculates the length of the longest path through each node, called *l-level*, which can be obtained by adding *t-level* to *b-level*. The node with the larger *l-value* is assigned a higher priority. Another necessary constraint is that priorities are assigned according to topological order. **ITPA** expects nodes in the critical path to be scheduled first and then other nodes.

Zhao *et al.* presented a scheduling heuristic **EOPA** (Execution Order Priority Assignment) [46]. **EOPA** is a non-preemptive DAG scheduling framework that partitions nodes in a DAG into providers (i.e., nodes in the critical

path) and consumers (nodes not in the critical path), intending to exploit both parallelism and dependency conditions. Given the partitioned nodes, the highest priorities are assigned to the providers, the second-highest priorities to nodes that might block the providers, and the lowest priorities to the other nodes.

DRL-based

Over the past few years, deep learning, or deep neural networks, have been increasingly popular in reinforcement learning throughout a variety of industries, including gaming, robotics, natural language processing, etc [30]. The combination of these techniques can be referred to as deep reinforcement learning (DRL). One area of application of DRL is performance optimisation, searching directly within the solution space of a combinatorial optimisation problem [18]. For instance, pointer networks [43] offer an organized method for using neural networks to solve combinatorial problems such as the Travelling merchant problem (TSP). There exists some research that applies DRL to scheduling tasks [28, 27, 44, 31].

Lee et al. [27] have seen that DRL found to be an effective solution to a variety of combinatorial optimization problems, and they have adopted this technique for scheduling DAG tasks. They propose a learning-based priority allocation model **GoSu** (graph convolutional task scheduler) for scheduling individual DAG tasks on multiprocessor systems with a non-preemptive mechanism.

Figure 4.4 illustrates the GoSu model structure with the working process. The GoSu model contains both a decoder and an encoder. The GoSu model contains a decoder and an encoder. The encoder takes as input a DAG that demands to be scheduled and converts it into a graph convolution network (GCN). The encoder then extracts temporal (e.g., execution times) and structural (e.g., precedence conditions) features from such graph representations. Utilising the features of all nodes extracted by the encoder, the decoder sequentially selects nodes to generate priority order.

The modules are end-to-end trained using DRL to provide a priority order for a DAG task input, with the learning objective being to reduce the makespan. Through the reinforce algorithm [45], the estimated makespan is employed as the reward signal for updating the model.

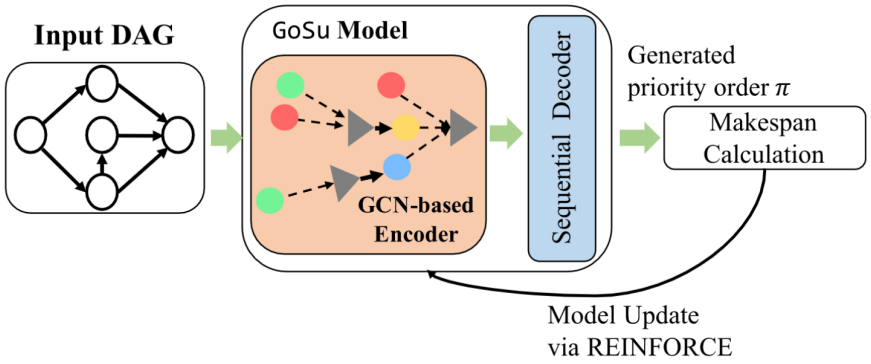


Figure 4.4: GoSu model [27]

4.1.4. Add Directed Edges

Hua *et al.* [22] proposed a DRL model **LAE** (Learning to add directed edges) for scheduling a DAG task. They are inspired by the finding that while the majority of work nodes are close to optimal, unsatisfactory schedules are frequently exclusively caused by the incorrect ordering of some of them. The quality of scheduling would be significantly improved if these ”tricky” nodes were correctly scheduled. The scheduling problem would be simplified and even very basic heuristics, e.g. shortest-job-first (**SJF**) and critical path (**CP**), may produce close to ideal solutions if the correct ordering of these nodes could be predicted a priori by a predictive machine.

Although the ordering of nodes can be determined by priorities, fixed-priority scheduling still cannot easily find the optimal solution. They propose a very novel approach to breaking tricky ties between nodes by adding directed edges, i.e. by explicitly requiring that one node be given priority over another in terms of execution and resource allocation.

Both SJF and CP could thus find optimal schedules after adding directed edges, shown in figure 4.5. Such an approach simplifies DAG scheduling by providing more restrictions and changes the original problem into a simpler proxy by including directed edges. In contrast to the DRL-based approach we mentioned in subsection 4.1.3, the LAE also uses a similar DRL technique. The LAE first uses a graph neural network to extract the features of the nodes, and then it employs a neural policy network to calculate the likelihood of each edge allowed to be added . Finally, the model is trained using reinforcement learning techniques.

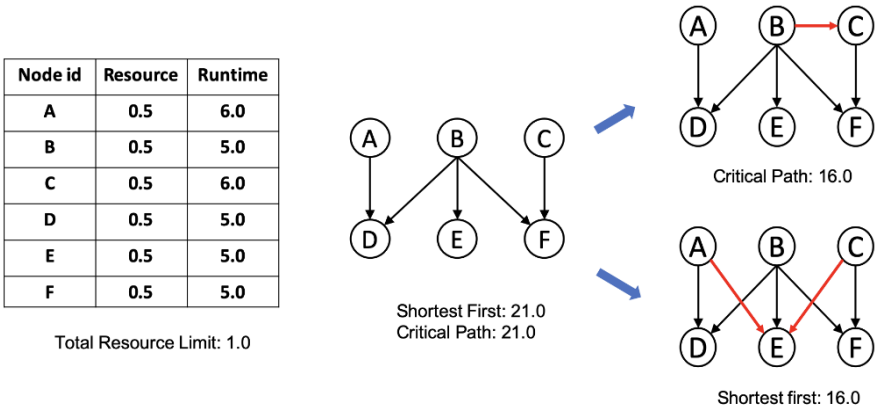


Figure 4.5: An example of improving DAG scheduling by adding edges [22]

4.2. Schedulability Analysis

Nasri *et al.* [36] provided a sufficient schedulability test for global limited preemptive fixed-priority scheduling.

In order to obtain bounds on the best and worst-case response times for each job, the analysis constructs a schedule abstraction graph (**SAG**) that abstracts all possible orderings of job dispatch times deriving from the underlying scheduling policy. Due to the offset and execution-time variance that jobs suffer, there are exponentially many execution scenarios, making it hard to estimate with certainty when each task will finish. For this reason, they took into account the earliest start time (**EFT**) and the earliest finish time as a result (**LFT**). A job J will be completed in the time period $[EFT, LFT]$.

The expansion and merging phases alternate as the SAG is constructed repeatedly. The expansion phase expands (one of) the shortest path(s) λ in the graph, by taking into account all tasks that can be dispatched following the job-dispatch sequence defined by λ. By combining the terminal vertices of paths that have the same set of dispatched jobs whenever it is practical, the merge phase reduces the graph's rate of growth. When there are no more vertices to expand, all pathways reflect a viable schedule for all occupations taken into account, indicating that all potential schedules have been investigated.

In conclusion, this analysis dominates existing analytical methods [40, 35, 7, 20, 29] and many other methods for sequential tasks. However, it does not scale to highly parallel DAG tasks or systems with a large number of cores (e.g., more than 64) [41].

4.3. Summary

A summary of related work presented in this chapter is presented in table 4.1. The results of these studies, from scheduling algorithms to schedulability tests, are presented in the summary.

Let us now look back to figure 4.1. The development of DAG scheduling algorithms can be summarised in two directions. One is to analyse the information provided by the task set and exploit this information to add constraints to the execution of the task set. The *structure-based* class of heuristics analyses the DAG structure and temporal characteristics. The GoSu and LAE go even further, extracting information with deep learning networks. They either add restrictions to reduce uncertainty in scheduling by generating fixed-node priorities or by adding additional directed edges. Another is to simplify the DAG task. This is represented by the *reconstruct task and scheduling* class. The task is decomposed by analysing the ideal scheduling of the DAG task to convert the task into a periodic task, thus eliminating the precedence constraints within the DAG.

However, we also observe that the common denominator for their improvement is the increased exploitation of the DAG structure and temporal characteristics, and the more comprehensive discussion and targeting of the various situations in the scheduling process. All the algorithms mentioned here take more information from the DAG task than the classical DM, RM, and EDF. On the second point, EOPA, for example, is a unique approach to classifying nodes as providers and consumers compared to the other three algorithms in the same category. DE-Jiang and DE-Eigbe, for example, classify segments as light and heavy compared to DE-Jing. They all attempt to classify the different situations that may be encountered and provide corresponding measures to deal with them.

Authors	Abbreviation	Research topic
Saifullah, Ferry, Li, Agrawal, Lu, and Gill [39]	DE-Jing + EDF	The decomposition-based global EDF scheduling of parallel DAG tasks on multiprocessors
Jiang, Long, Guan, and Wan [23]	DE-Jiang + EDF	The decomposition-based global EDF scheduling of parallel DAG tasks on multiprocessors
Eigbe, and Nasri [16]	DE-Eigbe + CW-EDF	The decomposition-based global CW-EDF scheduling of real-time tasks upon multiprocessor platforms
Kwok, and Ahmad [26]	ASAP	Static scheduling algorithms based on t-level for allocating DAG to multiprocessors
Kwok, and Ahmad [26]	ALAP	Static priority assignment based on b-level for allocating DAG to multiprocessors
He, Jiang, Guan, and Guo [21]	ITPA	Intra-Task priority assignment in real-time scheduling of DAG tasks on multiprocessors
Zhao, Dai, Bate, Burns, and Chang [46]	EOPA	A non-preemptive DAG priority-based scheduling framework of DAG tasks on multiprocessors
Lee, Cho, Jang, Lee, and Woo [27]	GoSu	A DAG task scheduler using deep reinforcement learning and graph convolution network
Hua, Qi, Liu, and Yang [22]	LAE+SJF/CP	Using machine learning to schedule DAG tasks
Nasri, Nelissen, and Brandenburg [36]	SAG	Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling

Table 4.1: Related work summary

5

Our solutions

In this chapter, we present our solutions for scheduling recurrent parallel DAG task sets on multiprocessor platforms. First in section 5.1 we introduce a scheduling anomaly. Then we show a case study that explores in depth how an existing scheduling algorithm can be compromised by such an anomaly that makes scheduling unsuccessful. We want to avoid this anomaly as well as possible, and based on the results of our investigation of this case study, we then present our solution in section 5.2.

5.1. Motivational Examples

In this section, we describe a particular example in subsection 5.1.1, which clearly illustrates a counter-intuitive reality. In subsection 5.1.2, we present and analyse how this anomaly affects scheduling.

5.1.1. Scheduling Anomaly

Faster computation does not always lead to an earlier completion of a DAG task. If a task set is scheduled on a multiprocessor with a certain priority assignment, a fixed number of processors, fixed execution times, and precedence constraints, then reducing execution times can increase the schedule length [19].

Let us consider a DAG task with ten nodes shown in figure 5.1. The number inside each node in the diagram is the index and also the priority, and we have marked each node's WCET with a blue number in its top-left corner. The node v_0 and v_1 obtain zero execution time, which are "dummy" nodes. Assume that this task set has only one periodic DAG task, namely, there is only one instance of this task in the hyperperiod. We denote the instance as $I^1 = \{J_1, J_2, ..., J_{10}\}$. All jobs of I^1 are release at time instant $t = 0$.

If such an instance is executed on a multiprocessor with three processors, where the highest priority job is assigned to the first available processor, the schedule is illustrated in figure 5.2. The total completion time of the first instance is 12 unit times.

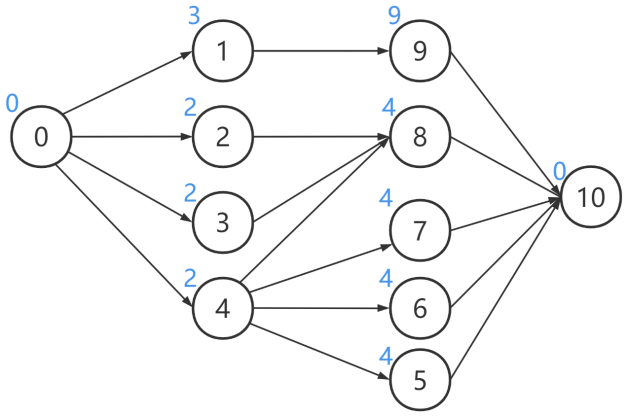


Figure 5.1: An example DAG task: the number inside each node is the index and also its priority, the top-left blue number is the worst-case execution time

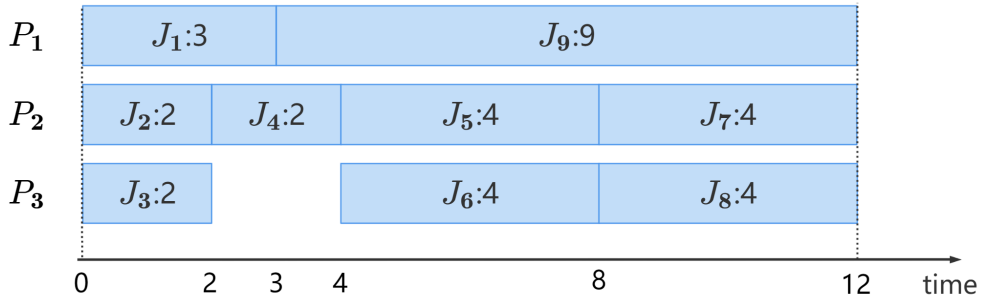


Figure 5.2: Schedule of first instance

Let us consider the case where the execution time is reduced. If the execution time of J_4 is reduced from 2 to 1, we see that the schedule length will increase with respect to the original schedule, and the total completion time will be 16, as shown in figure 5.3.

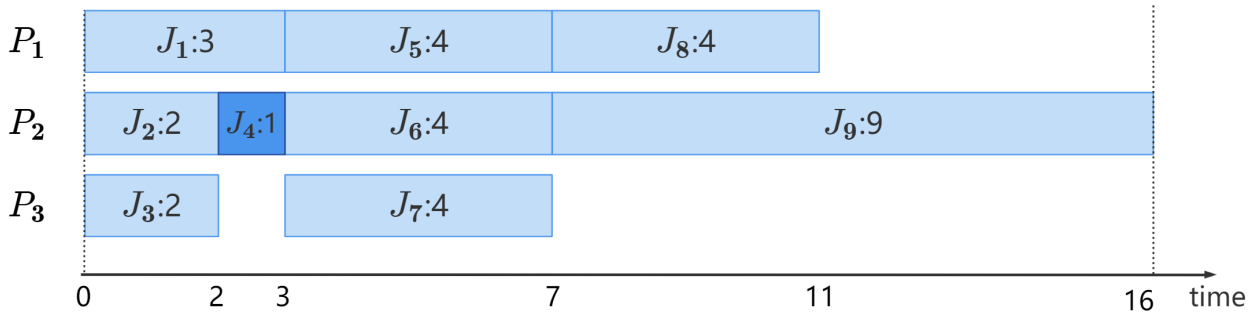


Figure 5.3: Schedule of first instance for anomaly case

The phenomenon is caused by the early completion of J_4 at $t = 0$, allowing J_5, J_6, and J_7 to be executed. They take up all the processors, preventing J_9 being processed in time.

5.1.2. Case Analysis

We present here a case study to illustrate the impact of execution-time uncertainty on scheduling. We first consider the situation where only WCET is concerned, and then consider the case where there is execution-time variation.

Case 1: No Execution-time Variation

Let's consider a task set with two tasks $\Gamma = \{\tau_1, \tau_2\}$ this time, and a multiprocessor with two processors. These two tasks are given by $\tau_1 = (G_1, T = 500, D = 500)$, $\tau_1 = (G_2, T = 1000, D = 1000)$. Here we applying ALAP, introduced in subsection 4.1.3, to produces the priorities for each node. There we give the DAG of those tasks in figure 5.4. The priority for each node is the orange number on the top-right of the node, the blue number at the bottom of the node is the WCET.

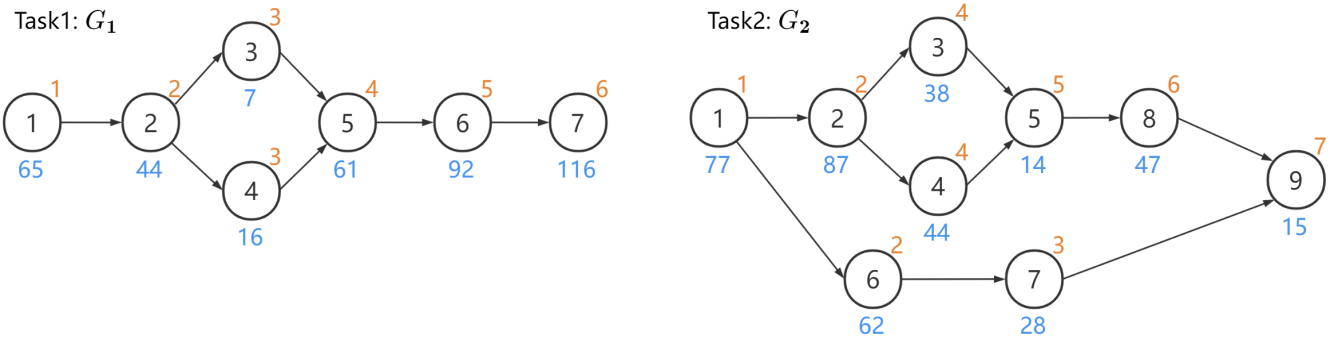


Figure 5.4: An example DAG task set Γ: the T and D of task 1 is 500, the T and D of task 2 is 1000, the index is the number in the node, orange number on the top-right of the node is its priority, the WCET is the blue number on the bottom of each node

The hyperperiod of Γ is $H(\Gamma) = 1000$, and there are total three instances I_1^1, I_1^2, and I_2^1 in one hyperperiod. The I_1^1 and I_2^1 are released at $t = 0$, the I_1^2 is released at $t = 500$. Here we the two instances released at $t = 0$, and give the schedule in figure 5.5. We can see that all jobs were successfully scheduled and no deadlines has been missed.

Case 2: With Execution-time Variation

Now, we consider the variation in execution time per job, and assume that BCET is 75 percent of WCET, which leads to a system that has more than one schedule. Therefore, we use SAG for response time analysis. The SAG

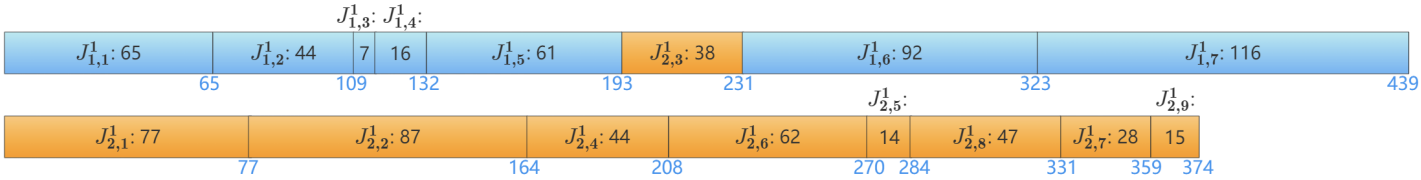


Figure 5.5: Schedule of I_1^1 and I_2^1 of Γ with ALAP

not only gives the schedulability results but also the worst-case completion time (**WCCT**) for each job.

We use SAG to analyze task set Γ within a hyperperiod. The result is not schedulable. There is a deadline miss for the job $J_{1,7}^1$, its deadline is 500. The WCCT of $J_{1,7}^1$ is 535. We observed that all the jobs in I_2^1 have WCCTs below 500 unit times, but they all have a deadline of 1000.

We also observed that priorities assigned to τ_1 are in a range from 1 to 6, the range from 1 to 7 for τ_2. The scheduler treats jobs with the same priority fairly. The I_1^1 and I_2^1 are released simultaneously at time point $t = 0$, but with very different deadlines.

As the ALAP only generates priority for each node of each task, this results in multiple instances released by a task being exactly the same except for the release time and absolute deadline. This is reasonable for a set with only one task, but multiple instances released at some time instant by multiple tasks can interfere with each other. In this example, the jobs of I_1^1 are severely interfered with by the jobs of I_2^1, causing many of the jobs of I_1^1 to fail to complete in time. Jobs with a deadline of 500 are not scheduled in time, but there are many jobs with a deadline of 1000 that can be completed within 500 unit times.

This shows that ALAP does not take into account the relationship between instances and hence has a poor perrformance when applied on recurrent DAG tasks. As for this example, we think that the scheduling algorithm should treat I_1^1 and I_2^1 differently. For example, give importance to the jobs of I_1^1 and scheduling them first. This case analysis leads us to consider an approach that takes into account the relationship between instances, or even between jobs.

5.2. Our Solution: Reassembly Stacking

In this section we will first introduce our solution, reassembly stacking (**RS**), a method for release-time tuning. Then a formal formulation of this algorithm is given. Finally we look back to Γ proposed in subsection 5.1.2 to see what the results of our solution combined with ALAP turn out to be.

5.2.1. Operational Processes

Our solution takes a comprehensive view of all the jobs in the hyperperiod of a task set. Instead of just considering priority, our solution takes into account the relationship between all jobs and sets a new release time for each of them.

This approach is also in line with the idea we proposed in subsection 5.1.2, to make certain jobs more important and to allow them to be scheduled first.

We show the overview of RS in figure 5.6. In the following we explain the behaviour of each step:

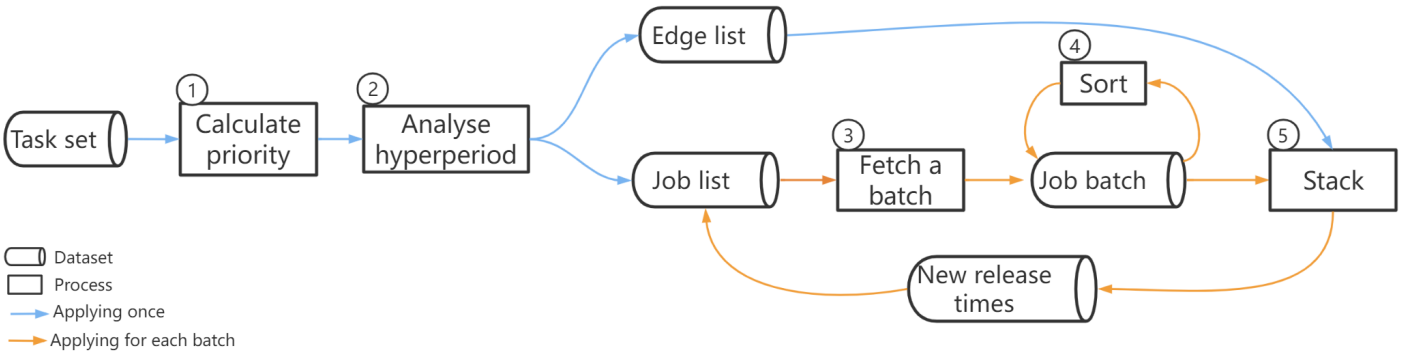


Figure 5.6: Operational processes of reassembly stacking

1. Calculate priority: For a given set of DAG tasks, this step generates **priority** for each node using the ALAP priorities.
2. Analyse hyperperiod: The hyperperiod of the DAG tasks set is calculated and a list of all jobs released within this hyperperiod is generated, i.e. a **job list**. Each job is assigned a priority. Precedence constraints between the jobs are also generated, i.e. the **edge list**.
3. Fetch a batch: This process is executed several times. Each time a **job batch** is generated. All the jobs in the **job list** with the same release time are packed into a batch, start with the early release time. For example, Γ release I_1^1 and I_2^1 at $t = 0$, release I_1^2 at $t = 500$, within a hyperperiod. Then the first job batch contains all jobs of I_1^1 and I_2^1. The second batch contains all jobs of I_1^2.
4. Sort: For each generated job batch, a **sort** process is performed. Sort all jobs in ascending order, first by **deadline** and second by **priority**. This means first sorting by the deadline, and if the jobs have the same deadline, then sorting by priority.
5. Stacks: Stack receives and stores the **edge list**, this is part of the initialization. Then, each time the stack receives a **job batch**, it generates a **new release time** for each of the jobs in it. It is then passed to the job list to update the release times. The stacks are the key part and we cover them in detail next subsection.

We summarize the reassembly stacking in algorithm 1. We will explain this algorithm based on the example task set Γ which present in figure 5.4. Here we take execution-time variation in to consideration. Lines from 1 to 5 correspond to the process (i) calculate priority in the figure 5.6, the produced priorities are shown as numbers on the top-right of nodes in the figure 5.4.

Lines from 6 to 11 generate *edge list* and *job list*, which complete the process (ii) analysis hyperperiod. For task set Γ, there are three instances

Algorithm 1 Reassembly Stacking Algorithm

Input: A task set Γ with n tasks, a multiprocessor model with m processors

Output: All jobs within a hyperperiod with precedence constrains

1: **for** $\tau_i \in \Gamma$ **do**
2: **for** $v_k^i \in \tau_i$ **do**
3: $p_k^i \leftarrow ALAP\ priority$
4: **end for**
5: **end for**
6: $H \leftarrow lcm_{\tau_x \in \Gamma}\{T_x\}$
7: initialise empty *job list* and *edge list*
8: **for** $\tau_i \in \Gamma$ **do**
9: $job\ list \leftarrow job\ list + jobs\ released\ within\ H$
10: $edge\ list \leftarrow edge\ list + edges\ between\ jobs\ released\ within\ H$
11: **end for**
12: initialize empty $Stacks \leftarrow \{S_1, S_2, ..., S_m\}$ where S_i is empty stack
13: pass $edge\ list$ to $Stacks$ for future use
14: $release\ time\ list \leftarrow all\ unique\ release\ time\ in\ job\ list$
15: sort $release\ time\ list$ in ascending order
16: **for** $r \in release\ time\ list$ **do**
17: $job\ batch \leftarrow jobs\ release\ at\ r$
18: sort $job\ batch$ in ascending order of deadline and priority
19: $new\ release\ times \leftarrow Stacks(jobs\ batch)$ according to Algorithm 2
20: replace $release\ times$ of jobs of corresponding $job\ batch$ in the $job\ list$ by $new\ release\ times$
21: **end for**
22: **Return** job list, edge list

I_1^1, I_1^2, and I_2^1 in one hyperperiod. We put all jobs of these instances to *job list*, and also put the precedence constrains between these jobs to *edge list*. Here we give the detailed edge list in table 5.2, also for the purpose of explaining algorithm 2 used in line 19 later.

The initialisation of stacks is completed from lines 12. In line 13, we pass the *edge list* to *Stacks*, make *Stacks* can access it. But not place it in *Stacks*. We get all unique release time in job list in line 14, that are $t = 0$ and $t = 500$ in this example.

The remaining part is responsible for looping through processes (iii) fetch a batch, (iv) sort, and (v) stack. Line 17 completes the steps to (iii) fetch a batch. For task set Γ, two batches are generated in total. The job batch 1 contains all jobs of I_1^1 and I_2^1 since they all release at $t = 0$. The jobs batch 2 contains all jobs of I_1^2 that they all release at $t = 500$.

As we obtain a job batch, the next step is (iv) sort these jobs in line 18. Here we give the detailed sorted job batches in table 5.3 and 5.4, also for the purpose of explaining algorithm 2 later.

We apply algorithm 2 to obtain a batch of new release times in line 19. In line 20, after a batch of *new release times* of a job batch is generated, it passed to *job list*. Then replace the *release times* of jobs of corresponding job batch by such *new release times*.

5.2.2. Stacks

The stacks follow a non-work-conserving policy and have special rules to place a new item. The idea of using stacks is to think of jobs as “blocks” (one dimensional) and the WCET as the size of the ”blocks“, which we fit into stacks. The goal is to place the jobs at stacks and make each jobs finish as early as possible. The start time of each job in the stacks is the **new release time** we want.

The above description looks like a bin packing problem, but we are not using the concept of bin. The bin packing only considers the capacity of the bin and the size of the item to be loaded, whereas stack is concerned with the position and order of the bolcks.

Stacks Construction

Given a task set Γ with n tasks, and a multiprocessor model with m processors, we build $Stacks = \{stack_1, stack_2, ..., stack_m\}$. That each stack is defined a sequence of jobs executed in a strict order. For example the $stack_p$ with $L+1$ blocks, denote as $stack_p = \{X_{p,0}, X_{p,1}, X_{p,2}, ..., X_{p,L}\}$. The $X_{p,0}$ is ”dummy” block, it is placed at a $stack$ when such $stack$ is declared. Hence there is no empty $stack$.

We assign a job $J^j_{i,k}$ from a job batch to a block and then place it at $stack_p$, we denote this block in the $stack_p$ as $X_{p,l} = (j_{p,l}, i_{p,l}, k_{p,l}, S_{p,l}, F_{p,l})$, the first three numbers are used to indicate which job corresponds to this block, the l represent its position in the $stack_p$ with $S_{p,l}$ is the start time, $F_{p,l}$ is the finish time. In all stacks, the ”dummy” jobs $J_{p,0}$ mentioned in the previous paragraph are always have $S_{p,0} = 0$ and $F_{p,0} = 0$.

We show an example of $Stacks$ in figure 5.7, $stack_1 = \{X_{1,0}, X_{1,1}, X_{1,2}, X_{1,3}\}$, and $stack_2 = \{X_{2,0}, X_{2,1}, X_{2,2}\}$, as we omit the ”dummy” jobs in figure. Different stacks within a $Stacks$ may contain different numbers of blocks. For a $stack_p$, we denote its size as $L_p + 1$. This allows us to conveniently indicate the last block of a $stack_p$ as J_{p,L_p}. For example, $L_1 = 3$, the last block of $stack_1$ is $X_{1,3}$.

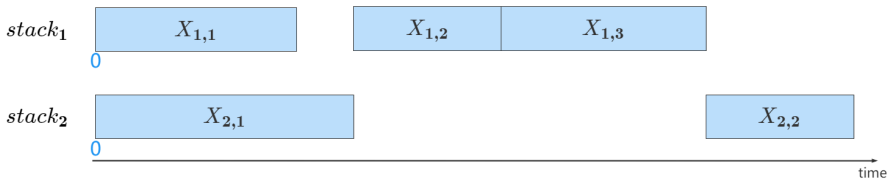


Figure 5.7: An example $Stacks$

Block Location

When $Stacks$ accepts a job $J^i_{j,k}$ as input, it construct the **target block** with determine the start time and end time of this block:

$$X = (j, i, k, S, F), \; where \; S = r^j_{i,k}, F = r^j_{i,k} + \omega_{i,k}. \tag{5.1}$$

We say $r^i_{j,k}$ is the **original release time** of such job. The block X here is not subscripted because it is not placed at any stack.

For a certain stack at a time instant, there can only be one job. It means that there should be no overlap between jobs. Suppose we are going to place the target job J to a $stack_p$, we need to check if it is **feasible** to place J at the interval $[S, F)$. Job is not necessarily placed at the end of the $stack_p$, therefore we consider all the jobs that already exist in the $stack_p$:

It is **feasible** to place a job J at $stack_p = \{J_{p,0}, J_{p,1}, ..., J_{p,l}\}$ if

$$\forall X_{p,l} \in stack_p | \ [S_{p,l}, F_{p,l}) \cap [S, F) = \emptyset. \tag{5.2}$$

As for the not-feasible situation, we will discuss it in the next part.

Block Stacking Policy

Here we will describe how a $Stacks$ that has been initialised handles a job $J^j_{i,k}$ from a job batch. We address this in a variety of situations as figure 5.8.

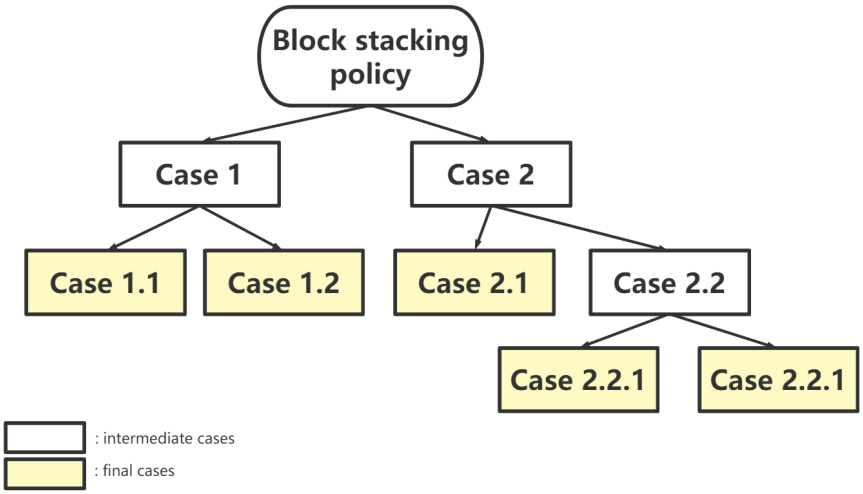


Figure 5.8: Different cases to be addressed by the block stacking policy

The yellow square is the final case, where the job handling is completed and the block corresponding to the job is placed at a stack. The white squares are the intermediate cases, where the block is constructed or the block parameters are updated. The conversion of cases is based on conditional decisions, which we will explain in detail next. Finally we will summarise the block stacking policy as algorithm 2 at the end of this subsection.

Please note that the following examples are not interrelated and are only specifically designed to illustrate how the stacking policy deals with each case.

Case 1 Job has no predecessor

For the case of a job without predecessors, we check if it is feasible to place the corresponding block to any stack with start at its **original release time**.

Case 1.1 There is a stack that feasible to place the block
Place the block to this stack, start at its **original release time**. Here we give a example to illustrate this case:

The handled job is $J_{2,1}^1$ with no predecessor, here we satisfy the conditions of case 1. The target block is $X = (1, 2, 1, 0, 71)$. The $Stacks$ as shown in figure 5.9.

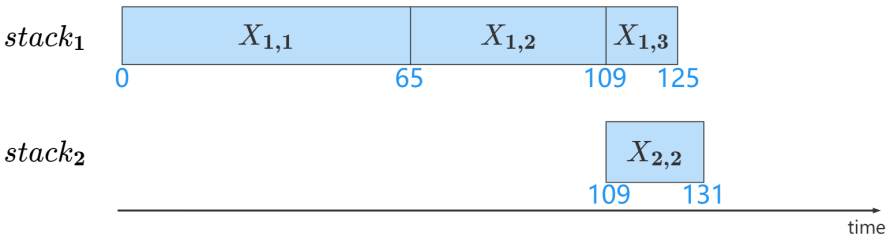


Figure 5.9: An example $Stacks$ for case 1.1

We start with check $stack_1$. The $stack_1$ is not feasible to place X since there is overlap between $X_{1,1}$ and X according to equation 5.2. Then we check $stack_2$, this stack is feasible to place block X. Here we satisfy the conditions of case 1.1, then place block X to $stack_2$ as figure 5.10 shows.

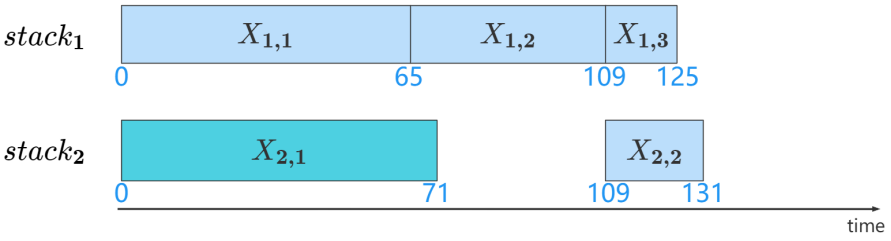


Figure 5.10: An example $Stacks$ after place the target job for case 1.1

Case1.2 There is no stack that feasible for the block
Place the job to **earliest available stack**, start at **overall least finish time**. Here we give a example to illustrate this case:

The handled job is $J_{3,1}^1$ with no predecessor, here we satisfy the conditions of case 1. The target block is $X = (1, 3, 1, 0, 33)$. The $Stacks$ as shown in figure 5.11. Both $stack_1$ and $stack_2$ are not feasible to place block X. Here we satisfy the conditions of case 1.2, there is no stack feasible for the job.

Then we need to find the **earliest available stack** $stack_p$, which is the stack that the last job in it has **overall least finish time** in comparison to all other stacks. The overall least finish time is given by:

$$F_p^{olft} = min(F_{1,L_1}, ..., F_{m,L_m}). \tag{5.3}$$

Since the X_{p,L_p} is the last block in $stack_p$, it is always feasible to place a block X after X_{p,L_p}. We place X to this $stack_p$ with update the X as:

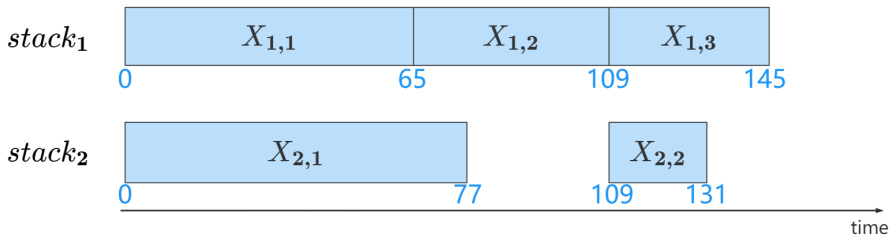


Figure 5.11: An example $Stacks$ for case 1.2

$$S = F_p^{olft}, \tag{5.4}$$

$$F = S + \omega_{i,k}. \tag{5.5}$$

Back to the example in figure 5.11. We apply equation 5.3 to obtain $F_2^{olft} = min(145, 131) = 131$. The **earliest available stack** is $stack_2$. Then we update the target block X to $X = (1, 3, 1, 131, 164)$ with equations 5.4 and 5.5. Here we place X to $stack_2$ as figure 5.12

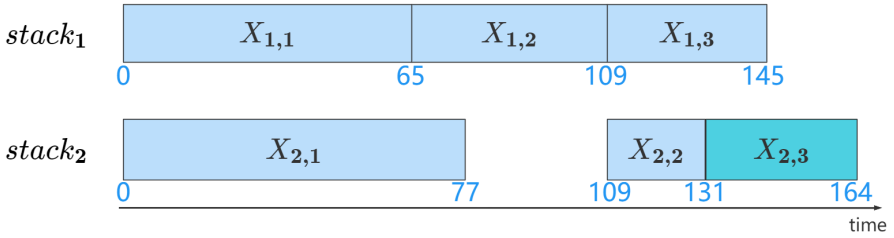


Figure 5.12: An example $Stacks$ after place the target job for case 1.2

Case2 Job has predecessor

For the case of a job with predecessors, we iterate all its predecessors, find its **latest predecessor**. It is a predecessor of the job that finishes the last completes its execution time. We also record its finish time, namely, **predecessors latest finish time**. Then check if it is feasible to place the block to the stack where the **latest predecessor** is, starting from **predecessors latest finish time**.

Considering the case with a precedence constraint, and there is no doubt that each job should start after the finish of all its predecessors. To consider this case we need to find the **latest predecessor** and the $stack_p$ where the **latest predecessor** is. We first get the predecessors of the job. Then we obtain the blocks in the $Stacks$ that correspond to each of those jobs, which form a set we denote as B.

Then find the **predecessors latest finish time** given by:

$$F_p^{plft} = max_{X_{p,l} \in B}(F_{p,l}). \tag{5.6}$$

Each job batch is ordered according to the release time, and the jobs within the job batch are already ordered according to priority. It is notable that the ALAP priority is a topological ordering, so all predecessors of the target job J are guaranteed to have been placed at stacks. That is, every predecessor of the job is guaranteed to have the corresponding block that already in $Stacks$ when we apply equation 5.6.

We update the X according to the following and equation 5.5:

$$S = F_p^{plft}. \tag{5.7}$$

Case2.1 It is feasible to place the block to the stack

Place the job to this stack, start at the **predecessors latest end time**. Here we give a example to illustrate this case:

The handled job is $J_{1,5}^1$ with predecessors $J_{1,3}^1$ and $J_{1,4}^1$, here we satisfy the condition of case 2. The target block is $X = (1, 1, 5, 0, 35)$. The $Stacks$ as shown in figure 5.13.

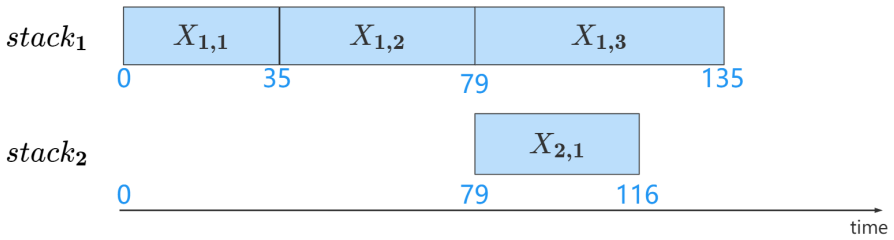


Figure 5.13: An example $Stacks$ for case 2.1

In this situation, the set $B = \{X_{1,3}, X_{2,1}\}$. We could obtain the **predecessors latest finish time** $F_1^{plft} = max(116, 135) = 135$ by equation 5.6. The block corresponding to the **latest predecessor** is the one with finish time 135, the $X_{1,3} = (1, 1, 4, 79, 135)$ in $stack_1$. We update the target block X to $X = (1, 1, 5, 135, 170)$ according to equation 5.7 and 5.5. According to equation 5.2, it is feasible to place X at $stack_1$, which satisfies the condition of case 2.1. Here we place X to $stack_1$ as figure 5.14

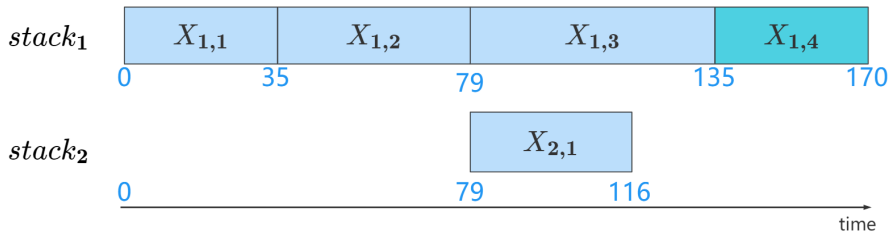


Figure 5.14: An example $Stacks$ after place the target job for case 2.1

Case2.2 It is not feasible to place the block to the stack

Case2.2.1 Iterate other stacks in ascending order, check if there is a stack feasible for place this block which start at **predecessors latest finish time**. Place the block to that stack if it exist. Here we give a example to illustrate this case:

The handled job is $J^1_{1,4} = (0, 50)$ with predecessor $J^1_{1,2}$, here we satisfy the condition of case 2. The target block is $X = (1, 1, 4, 0, 50)$. The $Stacks$ as shown in figure 5.15.

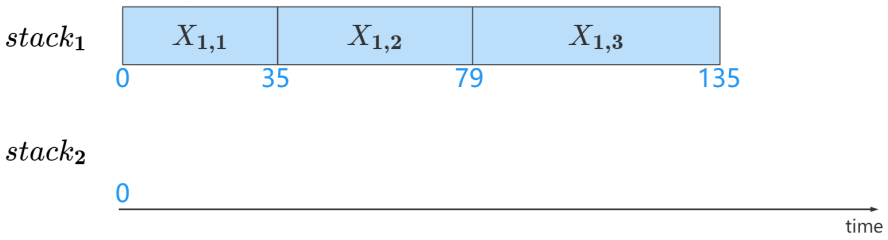


Figure 5.15: An example $Stacks$ for case 2.2.1

In this situation, the set $B = \{X_{1,2}\}$. We could obtain the **predecessors latest finish time** $F_1^{plft} = 79$ by equation 5.6. The block corresponding to the **latest finish predecessor** is $X_{1,2} = (1, 1, 2, 35, 79)$ in $stack_1$. We update the target block X to $X = (1, 1, 4, 79, 129)$ according to equation 5.7 and 5.5. According to equation 5.2, it is not feasible to place X at $stack_1$, which satisfies the condition of case 2.2.

Then we iterate other stacks in ascending order, check if there is a stack feasible for place this job. Found out that the $stack_2$ is feasible to place $X = (1, 1, 4, 79, 129)$, which satisfies the condition of case 2.2.1. Here we place X to $stack_1$ as figure 5.16

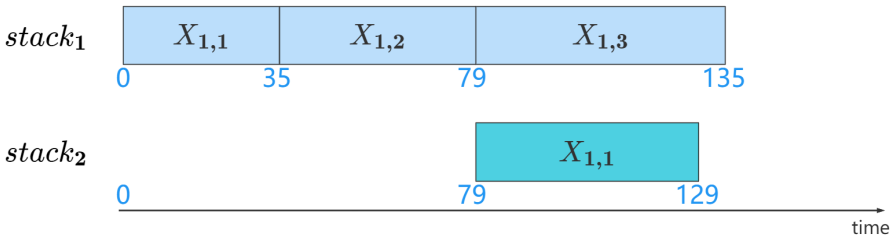


Figure 5.16: An example $Stacks$ after place the target job for case 2.2.1

Case2.2.2 Iterate other stacks in ascending order, check if there is a stack feasible for place this block which start at **predecessors latest finish time**. If there is not such stack. Place the block to **earliest available stack**, start at **overall least finish time**. Here we give a example to illustrate this case:

The handled job is $J \equiv J^1_{2,3} = (0, 38)$ with predecessor $J^1_{2,2}$, here we satisfy the condition of case 2. The target block is $X = (1, 2, 3, 0, 38)$. The $Stacks$ as shown in figure 5.17.

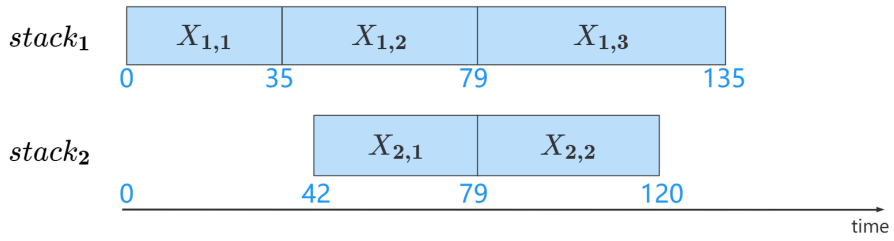


Figure 5.17: An example $Stacks$ for case 2.2.2

In this situation, the set $B = \{X_{2,1}\}$. We could obtain the **predecessors latest finish time** $F_2^{plft} = 79$ by equation 5.6. The block corresponding to the **latest finish predecessor** is $X_{2,1} = (1, 2, 2, 43, 79)$ in $stack_2$. We update the target block X to $X = (1, 2, 3, 79, 117)$ according to equation 5.7 and 5.5. According to equation 5.2, it is not feasible to place X at $stack_2$, which satisfies the condition of case 2.2.

Then we iterate other stacks in ascending order, check if there is a stack feasible for place this job. Found out that the only left stack $stack_1$ is not feasible to place $X = (1, 2, 3, 79, 117)$. There is no stack feasible for place X, which satisfies the condition of case 2.2.2.

For such a case, we apply equation 5.3 to obtain $F_2^{olft} = min(120, 135) = 120$. The **earliest available stack** is $stack_2$. Then we update the target block to $X = (1, 2, 3, 120, 158)$ with equations 5.4 and 5.5. Here we place X to $stack_2$ as figure 5.18

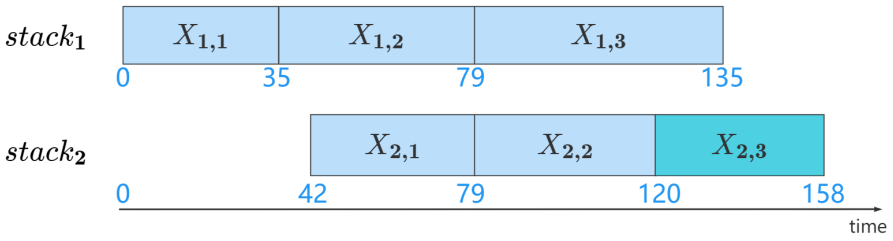


Figure 5.18: An example $Stacks$ after place the target job for case 2.2.2

The following algorithm 2 corresponds to line 19 in algorithm 1. Receives a batch of jobs as input and outputs the new release time for these jobs. Lines 4 to 18 correspond to the case 1 in subsection 5.2.2, where lines 5 to 12 look for a suitable stack as case 1.1 and the remainder is devoted to the case 1.2 where no such stack is found. Line 20 to 42 correspond to case 2, where lines 20 to 22 are used as a pre-processing when this situation occurs. The remainder is devoted to finding a suitable stack.

Algorithm 2 Job Batch Placing Stacking

Input: A job batch
Output: New release times of jobs

for $J^i_{j,k} \in jobs\ batch$ **do**
  declare J according to *Equation 5.1*
  $stack \leftarrow 1$
  if $pred(J) = \emptyset$ **then** ▷ Start of case 1
    while $stack \leq m$ **do**
      if feasible according to *Equation 5.2* **then** ▷ Start of case 1.1
        place J to $stack$
        break
      else
        $stack \leftarrow stack + 1$
      end if
    end while
    if $stack > m$ **then** ▷ Start of case 1.2
      find $stack_p$ and F_p^{olft} according to *Equation 5.3*
      $stack \leftarrow p$
      update J according to *Equation 5.4, 5.5*
      place J to $stack$
    end if
  else
    find $stack_p$ and F_p^{plft} according to *Equation 5.6* ▷ Start of case 2
    $stack \leftarrow p$
    update J according to *Equation 5.7, 5.5*
    if feasible **then** ▷ Start of case 2.1
      place J to $stack$
    else ▷ Start of case 2.2
      $stack \leftarrow 1$
      while $stack < m$ **do**
        $stack \leftarrow stack + 1$
        if feasible **then** ▷ Start of case 2.2.1
          place J to $stack$
          break
        end if
        if $stack = m$ **then** ▷ Start of case 2.2.2
          find $stack_p$ and F_p^{olft}
          $stack \leftarrow p$
          update J
          place J to $stack$
        end if
      end while
    end if
  end if
end for
Return start time S of all jobs

5.2.3. Illustrated Example

Let us look back to the task set Γ presented in subsection 5.1.2. Here we illustrate how our solution deals with this task set and then displays the schedule result.

Starting with algorithm 1, we first generate priorities for all nodes in task set Γ. Then after obtaining the hyperperiod $H = 1000$. By the completion of line 11, we have the following *job list* and *edge list*.

Job	Release time	BCET	WCET	Deadline	Priority
$J^1_{1,1}$	0	50	65	500	1
$J^1_{1,2}$	0	33	44	500	2
$J^1_{1,3}$	0	5	7	500	4
$J^1_{1,4}$	0	13	16	500	3
$J^1_{1,5}$	0	47	61	500	5
$J^1_{1,6}$	0	70	92	500	6
$J^1_{1,7}$	0	87	116	500	7
$J^1_{2,1}$	0	58	77	1000	1
$J^1_{2,2}$	0	66	87	1000	2
$J^1_{2,3}$	0	29	38	1000	4
$J^1_{2,4}$	0	34	44	1000	3
$J^1_{2,5}$	0	10	14	1000	6
$J^1_{2,6}$	0	46	62	1000	5
$J^1_{2,7}$	0	21	28	1000	8
$J^1_{2,8}$	0	36	47	1000	7
$J^1_{2,9}$	0	12	15	1000	9
$J^2_{1,1}$	500	50	65	1000	1
$J^2_{1,2}$	500	33	44	1000	2
$J^2_{1,3}$	500	5	7	1000	4
$J^2_{1,4}$	500	13	16	1000	3
$J^2_{1,5}$	500	47	61	1000	5
$J^2_{1,6}$	500	70	92	1000	6
$J^2_{1,7}$	500	87	116	1000	7

Table 5.1: Job list of Γ

From Job	To Job
$J^1_{1,1}$	$J^1_{1,2}$
$J^1_{1,2}$	$J^1_{1,3}$
$J^1_{1,2}$	$J^1_{1,4}$
$J^1_{1,3}$	$J^1_{1,5}$
$J^1_{1,4}$	$J^1_{1,5}$
$J^1_{1,5}$	$J^1_{1,6}$
$J^1_{1,6}$	$J^1_{1,7}$
$J^1_{2,1}$	$J^1_{2,2}$
$J^1_{2,2}$	$J^1_{2,3}$
$J^1_{2,2}$	$J^1_{2,4}$
$J^1_{2,3}$	$J^1_{2,5}$
$J^1_{2,4}$	$J^1_{2,5}$
$J^1_{2,1}$	$J^1_{2,6}$
$J^1_{2,6}$	$J^1_{2,7}$
$J^1_{2,5}$	$J^1_{2,8}$
$J^1_{2,7}$	$J^1_{2,9}$
$J^1_{2,8}$	$J^1_{2,9}$
$J^2_{1,1}$	$J^2_{1,2}$
$J^2_{1,2}$	$J^2_{1,3}$
$J^2_{1,2}$	$J^2_{1,4}$
$J^2_{1,3}$	$J^2_{1,5}$
$J^2_{1,4}$	$J^2_{1,5}$
$J^2_{1,5}$	$J^2_{1,6}$
$J^2_{1,6}$	$J^2_{1,7}$

Table 5.2: Edge list of Γ

Next, initialise $Stacks = \{stack_1, stack_2\}$. Then pass the *edge list* to $Stacks$, make $Stacks$ can access it. From the table 5.1 we can retrieve two unique release times $\{0, 500\}$, which already in ascending order.

Then we prepare two job batches for algorithm 2. The jobs with a release time of 0 and those with a release time of 500 are extracted from the job list to form a job batch respectively, and sort them. Here we give the detailed sorted job batches in table 5.3 and 5.4.

Job	Release time	BCET	WCET	Deadline	Priority
$J^1_{1,1}$	0	50	65	500	1
$J^1_{1,2}$	0	33	44	500	2
$J^1_{1,4}$	0	13	16	500	3
$J^1_{1,3}$	0	5	7	500	4
$J^1_{1,5}$	0	47	61	500	5
$J^1_{1,6}$	0	70	92	500	6
$J^1_{1,7}$	0	87	116	500	7
$J^1_{2,1}$	0	58	77	1000	1
$J^1_{2,2}$	0	66	87	1000	2
$J^1_{2,4}$	0	34	44	1000	3
$J^1_{2,3}$	0	29	38	1000	4
$J^1_{2,6}$	0	46	62	1000	5
$J^1_{2,5}$	0	10	14	1000	6
$J^1_{2,8}$	0	36	47	1000	7
$J^1_{2,7}$	0	21	28	1000	8
$J^1_{2,9}$	0	12	15	1000	9

Table 5.3: Sorted job batch 1 of Γ

Job	Release time	BCET	WCET	Deadline	Priority
$J^2_{1,1}$	500	50	65	1000	1
$J^2_{1,2}$	500	33	44	1000	2
$J^2_{1,4}$	500	13	16	1000	3
$J^2_{1,3}$	500	5	7	1000	4
$J^2_{1,5}$	500	47	61	1000	5
$J^2_{1,6}$	500	70	92	1000	6
$J^2_{1,7}$	500	87	116	1000	7

Table 5.4: Sorted job batch 2 of Γ

We then apply algorithm 2 to each job batch to generate the *new release times*. In the following we show the state of $Stacks$ after processing all the jobs from two job batches in figure 5.19.

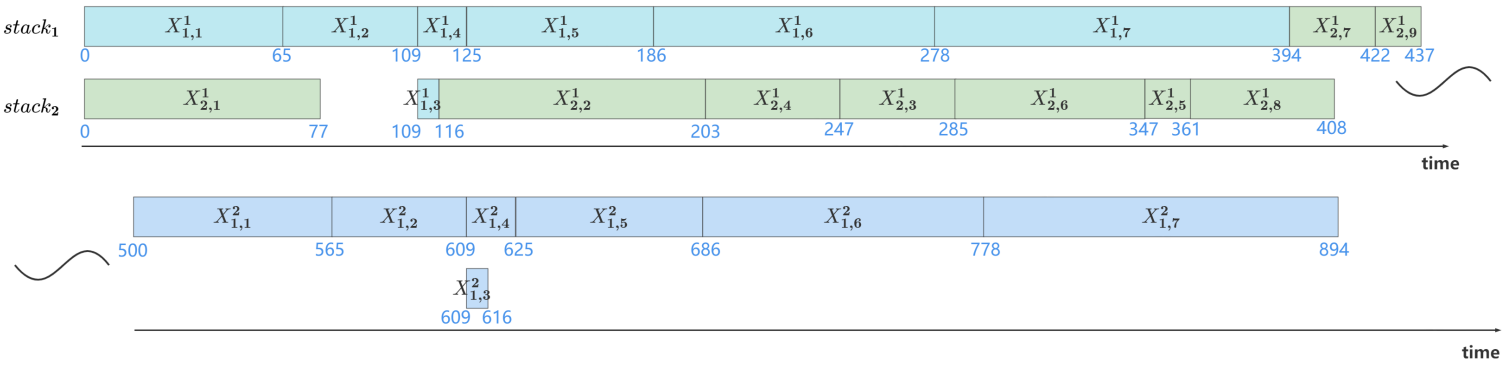


Figure 5.19: The state of $Stacks$

From the stacks we extract the *new release times*, which are the start times of each job in the graph. Then we update the job list, which means replace the *release time* by *new release times* for each job. The following table 5.5 shows the updated *job list* and we highlight the changes compared to table 5.1.

Job	Release Time	BCET	WCET	Deadline	Priority
$J^1_{1,1}$	**0**	50	65	500	1
$J^1_{1,2}$	**65**	33	44	500	2
$J^1_{1,3}$	**109**	5	7	500	4
$J^1_{1,4}$	**109**	13	16	500	3
$J^1_{1,5}$	**125**	47	61	500	5
$J^1_{1,6}$	**186**	70	92	500	6
$J^1_{1,7}$	**278**	87	116	500	7
$J^1_{2,1}$	**0**	58	77	1000	1
$J^1_{2,2}$	**116**	66	87	1000	2
$J^1_{2,3}$	**247**	29	38	1000	4
$J^1_{2,4}$	**203**	34	44	1000	3
$J^1_{2,5}$	**347**	10	14	1000	6
$J^1_{2,6}$	**285**	46	62	1000	5
$J^1_{2,7}$	**394**	21	28	1000	8
$J^1_{2,8}$	**361**	36	47	1000	7
$J^1_{2,9}$	**422**	12	15	1000	9
$J^2_{1,1}$	**500**	50	65	1000	1
$J^2_{1,2}$	**565**	33	44	1000	2
$J^2_{1,3}$	**609**	5	7	1000	4
$J^2_{1,4}$	**609**	13	16	1000	3
$J^2_{1,5}$	**625**	47	61	1000	5
$J^2_{1,6}$	**686**	70	92	1000	6
$J^2_{1,7}$	**778**	87	116	1000	7

Table 5.5: Updated job list of Γ

Finally, we schedule the task set Γ with ALAP. For the situation without execution-time variation, the schedule of I_1^1 and I_2^1 shows in figure 5.20. It turns out to still no deadlines missing. Furthermore, in comparison to figure 5.5, our algorithm allows these jobs to end 2 unit times earlier.

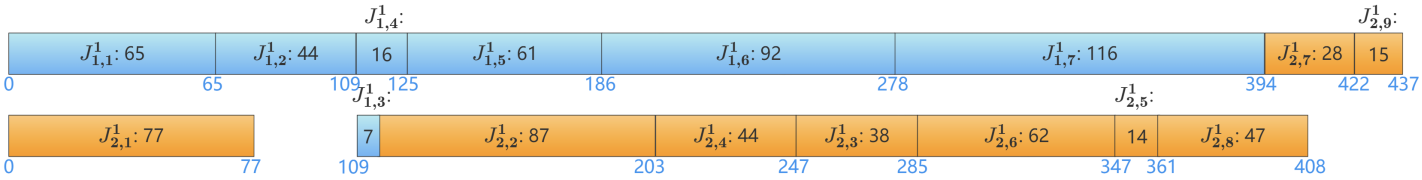


Figure 5.20: Schedule of I_1^1 and I_2^1 of Γ with RS+ALAP

For the situation with execution-time variation, we also apply the SAG for response time analysis. There are still no deadlines missing, and even all I_1^1 and I_2^1 jobs can be completed within 500 unit times. This also brings the additional benefit that I_1^2, the second release of τ_1, is not interfered by previously released jobs. These jobs will be more likely to be successfully scheduled.

5.3. Our Solution: Partitioned Reassembly Stacking

By reading this section we assume that you have already read and understood the previous section. As the solution to be explained in this section have overlapping parts with **RS**. We will not re-explain them in this section, but rather explain where they differ.

The stacks we use are essentially simulations of multi-core processors, and it is worth noting that the schedule in figure 5.20 is exactly the same as our simulation in figure 5.19. This suggests that we can somehow control the running of jobs on multiple core processors by adjusting the release times. Provided that we find a state of stacks where all jobs are completed before deadline, then there is a high probability that the updated jobs list is schedulable.

Thus we consider imposing tighter restrictions on these jobs. Not just tuning release times, but limiting the processors they execute, *i.e.* constructing a partitioned scheduling.

We propose partitioned reassembly stacking (**P-RS**). This solution is consistent with **RS** in section 5.2, except for the following four aspects:

1. We place all jobs in stacks, which means that each job definitely has a corresponding stack. We can extract the index of the stack where each job is located from the stack information. Here $Stacks$ return not only the *new release times* after a job batch has been input, but also the *index of the stack* where each job is located.
2. When updating the *jobs list*, besides updating the *release time*, a new column would be added to the *jobs list*. This is used to store the *stack index* for each job.

3. After the jobs list is completely updated, the *job list* is divided into m (number of processors) *job lists* according to the *stack index*, i.e., each job is assigned to a certain processor.
4. For a completely updated and completed jobs list, **RS** has chosen to use ALAP for schedule. Here we have already allocated a defined number of jobs to individual processor, therefore we apply the classic uniprocessor scheduling method namely first-come-first-serve (**FCFS**).

In order to achieve the first change mentioned above, we modify line 43 of algorithm 2 to "**Return** start time S, stack indexes of all jobs".

Correspondingly, to bring about the second and third modifications stated above, we replace lines 16 to 22 of algorithm 1 with the following algorithm 3. Line 6 implements the second modification. Lines 8 and 9 correspond to the third change.

Algorithm 3 Replacements of RS for Partitioned Reassembly Stacking

1: **for** $r \in release\ time\ list$ **do**
2: $job\ batch \leftarrow jobs\ release\ at\ r$
3: sort $job\ batch$ in ascending order of deadline and priority
4: $new\ release\ times, stack\ indexes \leftarrow Stacks(jobs\ batch)$
5: update $job\ list$ by $new\ release\ times$
6: add a column in $job\ list$ to record $stack\ indexes$ of each job
7: **end for**
8: split $job\ list$ by $stack\ indexes$ to $m\ job\ list$
9: **Return** m job lists

6

Evaluation Framework

The evaluation framework provides a customisable experimental environment and evaluation results. The goal of this evaluation framework is to compare different scheduling algorithms. This chapter describes the evaluation framework we have built for recurrent DAG tasks, starting with an introduction to the framework in section 6.1, followed by an explanation of its components in section 6.2.

6.1. Evaluating Scheduling Algorithms

We need to compare our solution with other scheduling algorithms to evaluate its performance. We observed some scheduling algorithms evaluated by conducting experiments on large sets of randomly generated tasks [27, 21, 28]. In addition, many parameters need to be taken into account when designing experiments. We envisage that such an evaluation framework can be used not only for the present research but also for future researchers. Repetitive experimental environment construction can be avoided and the focus can be on algorithm design and experimental parameter design.

We are not the first to come up with this idea, an evaluation framework ”DAG Scheduling and Analysis on Multiprocessor Systems” [46] has already been proposed. Such a framework supports various scheduling algorithms, schedulability test methods, etc. However, there are still some limitations to this framework, as it is designed for the priority-based scheduling algorithms and only supports global scheduling. Some partitioned scheduling algorithms cannot be evaluated here. Moreover, we would like to apply the state-of-the-art schedulability test method [36], which is not supported by this framework.

6.1.1. Performance Metric

The state-of-the-art metric widely used is the schedulability ratio [16, 28, 21, 27, 15, 12]. Without exception, we compare the different algorithms in terms of **schedulability ratio** as a metric, which is defined as the ratio of schedulable task sets to all task sets generated.

6.1.2. Parameters

In the following, we list the parameters that can influence the performance of the algorithm.

- **Number of tasks**: The number of DAG tasks in a task set. Higher quantities lead to more complex task systems.
- **Utilisation**: Utilisation reflects how busy a system is. We configure the utilisation as percentages of the number of processors. The utilisation of a task set would not be greater than m.
- **Number of processors**: We choose the multiprocessor platform with homogeneous architecture. The only configuration is the number of processors.
- **DAG structure**: The configuration of the DAG structure determines the branching complexity of these DAG task nodes, the number of nodes, the dependencies between them, etc.

6.1.3. Functionalities

We aim to make the evaluation framework as comprehensive as possible, covering most of the experimental processes. After the user has set up all the parameters and requirements for the experiment, the framework automatically completes the rest of the process. After completing the whole process, a graphical representation of the processed experimental data is given.

To elaborate, the evaluation framework should support the following functionalities:

- Selectable number of generated task sets
- Selectable DAG task sets parameters
- Selectable processors number
- Selectable processors utilisation
- Selectable scheduling algorithm for evaluation
- Provide a built-in schedulability analysis tool
- Analysis and storage evaluation results in tables
- Analysis results and generate statistical graphs

6.2. Software Architecture

The evaluation framework consists of four main components, those are task generator, task parser, schedulability tester, and results analyser. They form a ”pipeline” style workflow as shown in figure 6.1. The user determines how many DAG task sets need to be generated, the number of tasks in a set, the structural limits of the DAG, the number of processors to be tested, the utilisation, and the scheduling algorithm.

The task generator generates random DAG task sets and passes them to the task parser. The task parser repeatedly configures all of the information for all jobs released by one task set, within a hyperperiod, including constraints, by given scheduling algorithm. The schedulability tester utilises

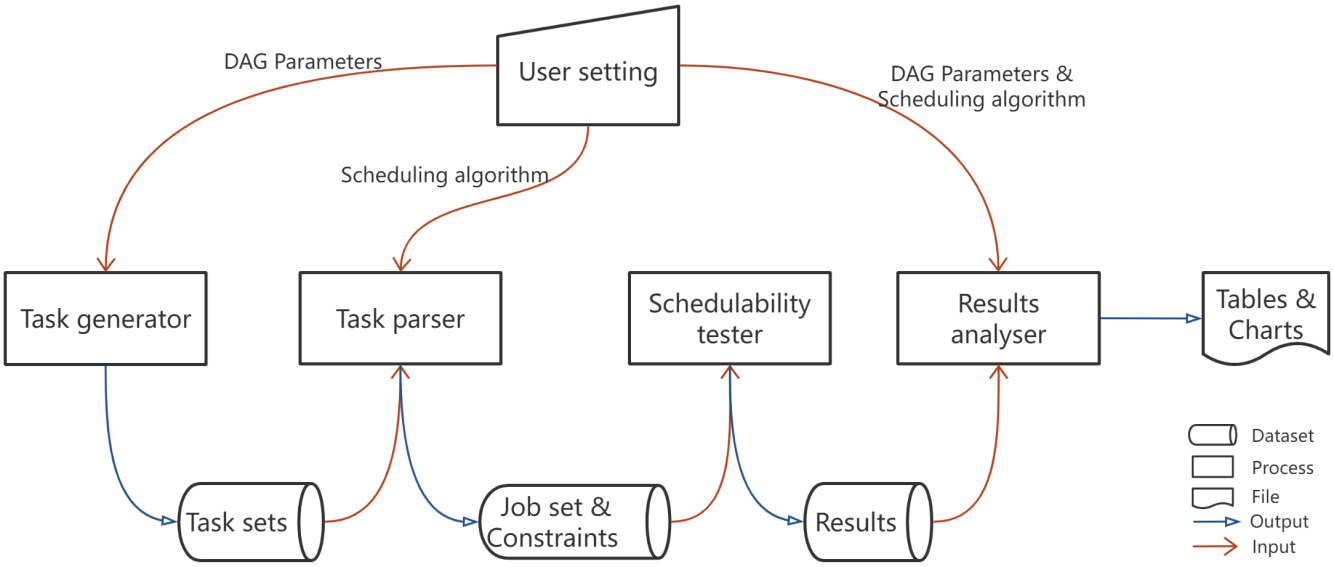


Figure 6.1: Evaluation framework software architecture

the information to determine if those task sets are schedulable, each by each, and stores the results. The final step is completed by the results analyser, which combines those results with the selected experiment parameters to produce the corresponding tables and charts.

6.2.1. Task Generator

The task generator generates a DAG task set by first calculating various utilization of each task (DAG). We apply the UUniFast algorithm [9] which first generates a utilization value by taking a random variable, uniformly distributed in $[0, 1]$, and multiplying it by the total utilisation. Then replace total utilisation with the difference between total utilisation and the generated. For the period assignment of each task, we use the sum of the WCET of all nodes of the task if utilization of it is 1, otherwise use that sum divided by utilization. But to avoid the hyperperiod becoming impractically large, we scaled the obtained periods in the interval $[500, 100000]$ with values that in the set $\{x \cdot 10^y | 1 \leq x \leq 9, 3 \leq y \leq 5\}$ [36]. After assigning periods, we proportionally scale the WCET of the nodes so that tasks keep their intended utilization.

For the generation of the internal structure of the DAG, we keep expanding each node by recursion [32] until the limits of the target DAG are reached. Determined by the configurable probabilities, a node may either be an end node or a node that allows more branches to be connected.

For the constraints of the target DAG, our generator supports configurable parameters to achieve this, which we list and explain in the table 6.1. The values in the last column are suggested values and we have reserved them for the empirical evaluation in chapter 7.

Parameter	Description	Default Value
Runs	Number of task sets	varied
Num N	Number of tasks (DAGs) in a task set	varied
Utilisations	Percentages of the number of processors	varied
Resource Types	Number of processors	varied
Max Jobs Per HP	Maximum number of jobs in the hypterperiod	1000
MIN PAR BRANCHES	Minimum number of siblings of a node	1
MAX PAR BRANCHES	Maximum number of siblings of a node	3
Prob Terminal	The probability of a node being end node	0.3
Prob Parallel	The probability of a node having successor	0.7
Prob Add Edge	The probability of having an edge between two sibling nodes	0.1
Max Critical Path Nodes	Maximum number of nodes in critical path	50
Min Nodes	Minimum number of nodes in a DAG	5
Max Nodes	Maximum number of nodes in a DAG	50
Max Node WCET	Maximum WCET of node	100

Table 6.1: Configurable DAG parameters

6.3. Task Parser

We have inherited and modified the open-source tool provided by Nasri et al [36]. The task parser receives the DAG task, parses all jobs released by the task within a hyperperiod, and generates the following files: *job sets* and corresponding *precedence constraints*. In addition, the parser prioritises each job according to the chosen scheduling algorithm.

To support partitioned scheduling, for example, P-RS + FCFS, the task parser generates *job sets* and corresponding *precedence constraints* for each processor. Following we give a list of all the scheduling algorithms in the table 6.2 that are now supported. The author was unable to provide an implementation therefore we cannot involve **GoSu** [27] in our framework.

6.4. Schedulability Tester

We incorporate the schedule abstraction graph (**SAG**) [36], mentioned in section 4.2, into the evaluation framework as our schedulability tester. The SAG supports global multiprocessor analysis, which is suitable for evaluating global scheduling algorithms. In such case, as figure 6.1 shows, the tester accepts the *job set* and *precedence constraints* of one task set, as well as

Abbreviation	Description
RM	Rate monotonic
DM	Deadline monotonic
EDF	Earliest deadline first
ASAP	As-Soon-As-Possible, a fixed-node priority assignment introduced in [26]
ALAP	As-Late-As-Possible, a fixed-node priority assignment introduced in [26]
ITPA	Intra-Task Priority Assignment, a fixed-node priority assignment proposed in [21]
EOPA	Execution Order Priority Assignment, a fixed-node priority assignment proposed in [46]
De-Eigbe + FIFO-OT	Applying the decomposition method according to [16]. Using first-fit partitioning, also adjust the release time by **FIFO-OT** [33].
De-Jing + EDF	Applying the decomposition method according to [39]. Using the first-fit partitioning and schedule jobs by **EDF**.
Reassembly stacking + ALAP	Our solution proposed in section 5.2
Partitioned RS + FCFS	Our solution proposed in section 5.3

Table 6.2: Supported scheduling algorithm

the number of processors. The output includes a variety of information, such as input file name, number of jobs, amount of memory used, etc. The most crucial output is the schedulability result, which produces a value of 1 if the job set is schedulable and a value of 0 otherwise.

For portioned scheduling, the tasks are pinned to processors, preventing any jobs associated with a particular task from running on more than one processor. To adapt to this situation, we have extended the SAG as figure 6.2.

Consider a task set that runs on m processors, as shown on the left side of the figure, with each processor assigned a *job set* and *constraints*. The SAG processes each of these datasets in turn, treating them as there is only one processor. In total, m schedulability results are generated. Finally, we do a logical *OR* operation on these results to obtain the schedulability of the task set.

6.5. Results Analyser

Tester generates a large number of scheduling results of randomly generated task sets. The analyser receives these results, calculates the schedulability ratio of each experimental situation and summarizes it as a table. It also generates various statistical charts to visually represent the experimental results, some generated charts will be shown in Chapter 7.

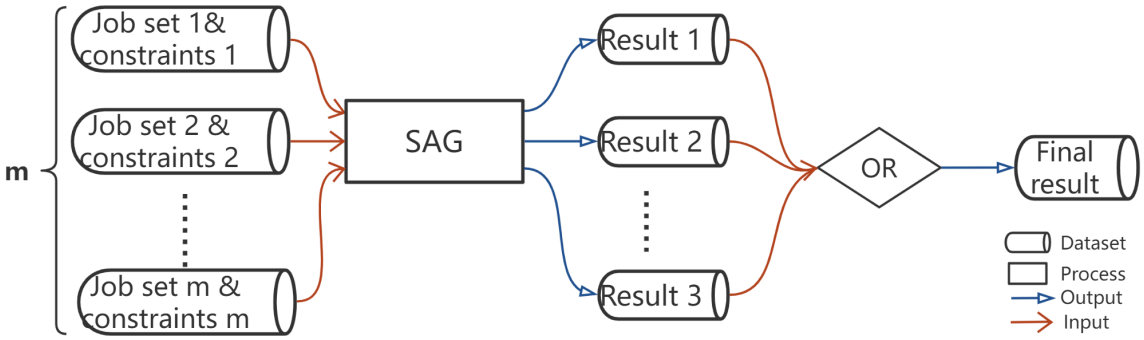


Figure 6.2: Tester for portioned scheduling

7

Empirical evaluation

In this chapter, we experimentally evaluate the solutions introduced in chapter 5. In section 7.1, the parameters of the experiments are presented. In section 7.2, we present and discuss the evaluation results of scheduling recurrent parallel DAG task set where jobs have execution-time variation.

7.1. Experiment setup

In this chapter, we use shorter abbreviations for brevity and formatting consistency. In table 7.1 we present the scheduling algorithms evaluated with a detailed explanation. We did not evaluate the scheduling algorithm *De-Jing + CW-EDF* in table 6.2 since it was empirically verified to perform worse than *DeEigbe* in most cases.

Our experiments are performed on the evaluation framework proposed in chapter 6. We randomly generate 500 tasks sets for each parameter setup in each experiment, *i.e.* a data point. Here we list the experimental parameters as follows:

- **Number of processors**: Our research applies to homogeneous multiprocessor platforms, and the number of processors m in the experiments can be tuned. Our experiments include platforms with 2, 4 and 8 processors.
- **Number of tasks**: The complexity of scheduling a task set increases with the number of DAG tasks it contains. We evaluate the impact of the number of tasks n on the behaviour of the scheduling algorithm. The range that we choose for our experiments is from 2 to 10. This guarantees the existence of experiments where the number of tasks is slightly greater than the number of processors.
- **Utilisation**: We configure the utilisation U as percentages of the number of processors. The numerical range for percentages is between 0 and 1. Our experiments ranged from 10% to 90% with an interval of 0.1.
- **DAG structure**: These DAG task nodes' branching complexity, number, inter-dependencies, etc. are all determined by the configuration

Abbreviation	Description
RM	Rate monotonic described in section 4.1.1. We only evaluate RM since our task model obtains implicit deadline that DM described in section 4.1.1 and RM perform the same.
EDF	Earliest deadline first priority assignment algorithm
ASAP	As-Soon-As-Possible, a fixed-node priority assignment proposed in [26], described in subsection 4.1.3.
ALAP	As-Late-As-Possible, a fixed-node priority assignment proposed in [26], described in subsection 4.1.3.
ITPA	Intra-Task Priority Assignment, a fixed-node priority assignment proposed in [21], described in subsection 4.1.3.
EOPA	Execution Order Priority Assignment, a fixed-node priority assignment proposed in [46], described in subsection 4.1.3.
DeEigbe	Eigbe method described in 4.1.2. Applying task decomposition method proposed in [16], then using first-fit partitioning to assign tasks to processors, adjust the release time by FIFO-OT [33].
RS	Our solution RS proposed in section 5.2, then schedule all jobs of one hyperperiod with ALAP.
P-RS	Our solution P-RS proposed in section 5.3, then schedule jobs on each processor with FCFS.

Table 7.1: Evaluated scheduling algorithm

of the DAG structure. We follow the default DAG parameters of the evaluation framework as shown in table 6.1.

7.2. Empirical Results

In this section we present some important and valuable experimental results in the first two subsections. Then in subsection 7.2.3, we discuss why there are such results.

7.2.1. Impact of the Number of Tasks

Figure 7.1 shows the behaviour of the scheduling algorithms with respect to the number of tasks n, the utilisation U is 80%, including experiments with different number of processors m. The data for some of the algorithms are incomplete because the time consumed by these experiments is too large (mainly due to the schedulability test) and these algorithms already exhibit a very low schedulability ratio when the number of tasks is 6.

We observe that the schedulability ratio of the scheduling algorithms de-

creases as the number of tasks increases. There is an exception at $n = 6$ and it only occurs at $m = 2$. We believe this is due to the random nature of task generation and the DAG structure, and does not affect our judgement of the overall trend. When $m = 2$, algorithms RS, P-RS perform outstandingly relative to the other scheduling algorithms. They approach a schedulability ratio of 0 at $m = 2$, but RS and P-RS still maintain a schedulability ratio of around 0.3 until $m = 10$, when it decreases to around 0.1. The algorithms RS, P-RS, and DeEigbe perform better at $m = 4$, and the schedulability ratio is still greater than 0 for task numbers greater than 6. When $m = 8$, three scheduling algorithms continue to outperform. DeEigbe performs best at $n = 7$ and $n = 8$, but as n increases to 10, the schedulability ratio of DeEigbe drops sharply to be completely not schedulable. However, RS still achieves a schedulability ratio of around 0.45 and P-RS also performs above 0.2.

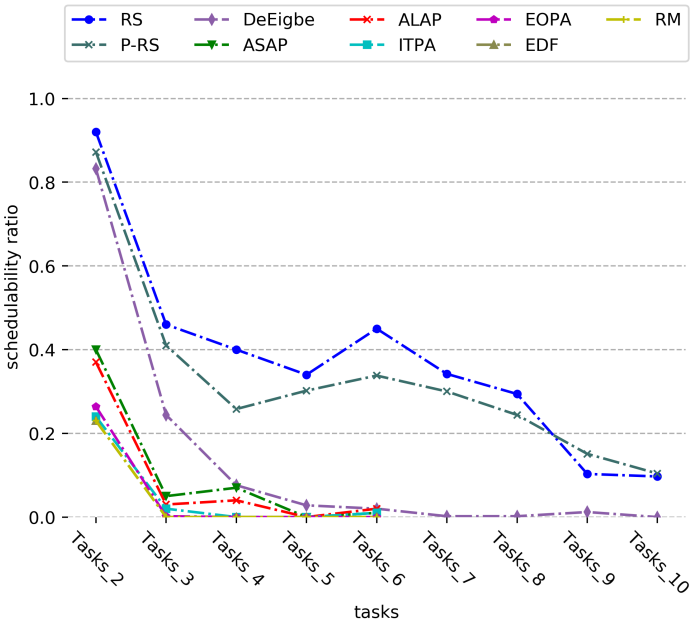
7.2.2. Impact of the Utilisation

Figure 7.2 shows the behaviour of the scheduling algorithms with respect to U, including experiments with different m.

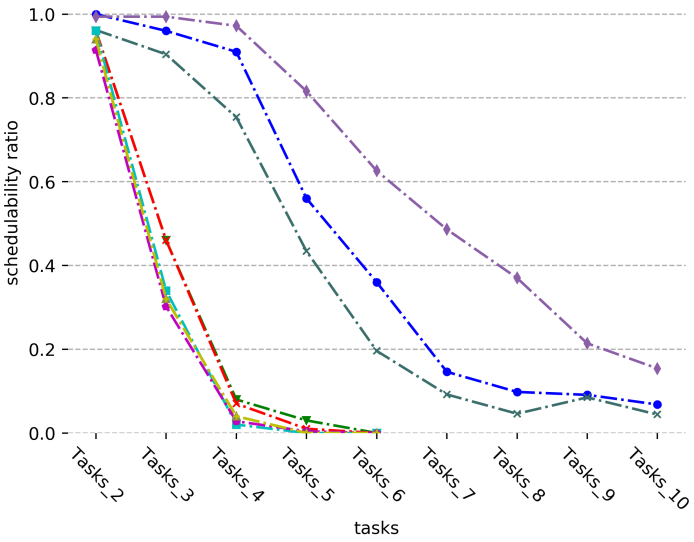
In the case of $n = 5$, we observe that the performance of all scheduling algorithms decreases with increasing utilisation. At $m = 2$ RS performs best, above all other scheduling algorithms. At $U = 60$% and $U = 80$%, it still outperforms the other algorithms, although the result are worse than RS. DeEigbe still performs remarkably well at $m = 4$. Also the results for RS and P-RS are significantly higher than the other scheduling algorithms. However, RS and DeEigbe show equal competitiveness in successfully scheduling all the generated task sets when $m = 8$, regardless of the high or low U.

Figure 7.3 shows the behaviour of three scheduling algorithms with respect to U, experiments are done on a higher number of tasks that $n = 10$ compare with figure 7.2.

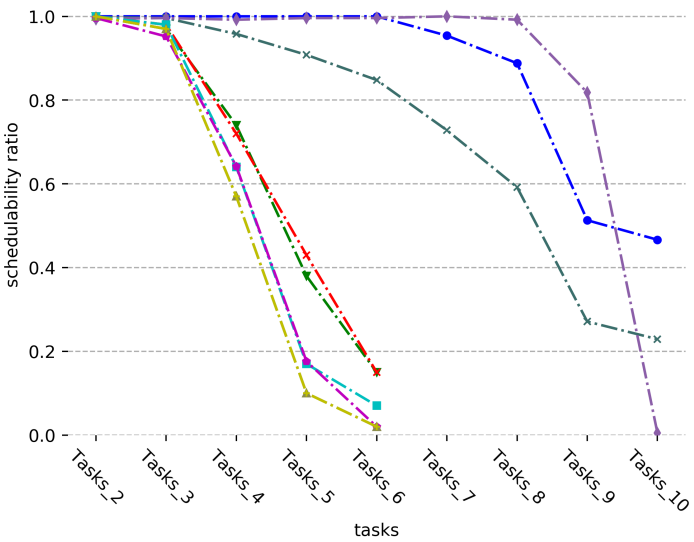
RS performs best when $m = 2$ and $m = 8$. At most there is a schedulability ratio difference of almost 0.5 compared to DeEigbe at 60% to 80% of U with $m = 8$. However DeEigbe still performs remarkably well at $m = 4$.



(a) $m = 2$

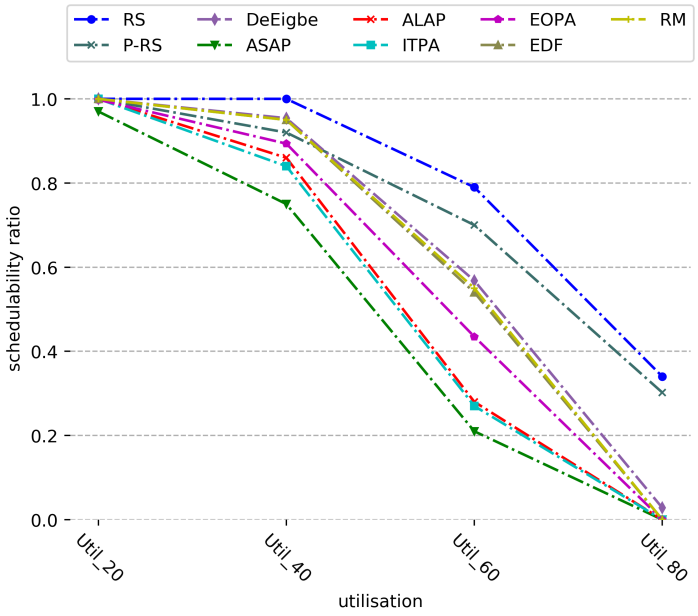


(b) $m = 4$

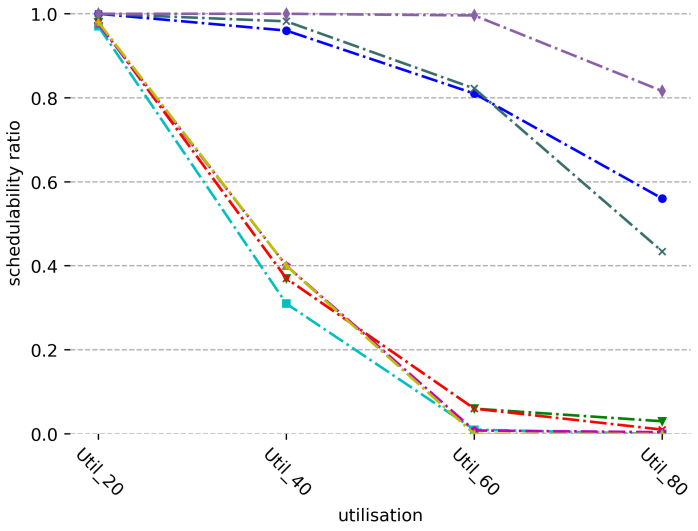


(c) $m = 8$

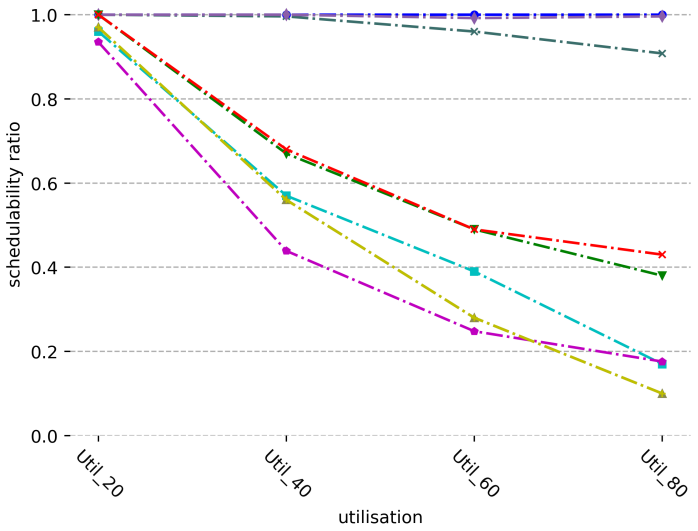
Figure 7.1: Schedulability between the number of DAG tasks and the number of cores



(a) $m = 2$

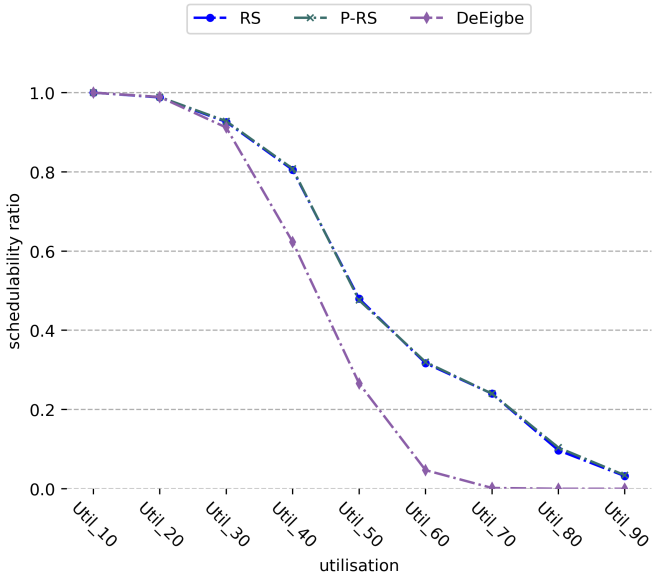


(b) $m = 4$

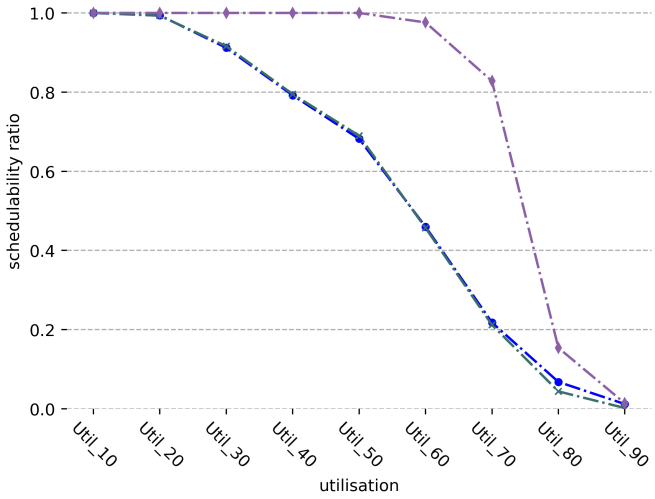


(c) $m = 8$

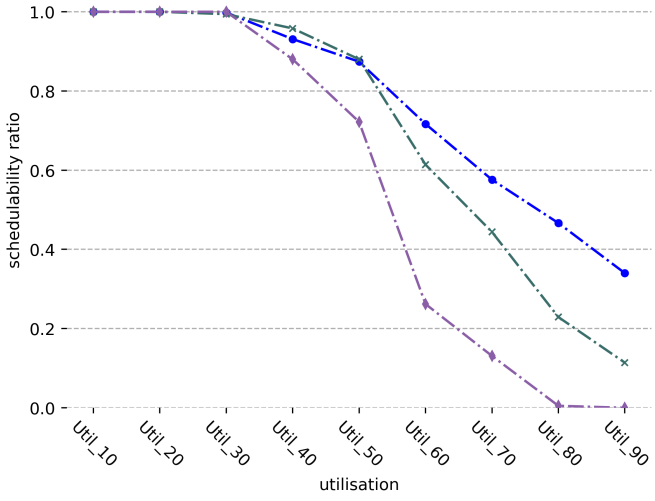
Figure 7.2: Schedulability between different utilisation rates and the number of cores (the number of tasks is 5)



(a) $m = 2$



(b) $m = 4$



(c) $m = 8$

Figure 7.3: Schedulability between different utilisation rates and the number of cores ((the number of tasks is 10))

7.2.3. Summary and Discussions

Here we conclude the evaluation results. We observe that in most cases P-RS does not perform better than RS. RS and P-RS obtains higher schedulability ratio than other global scheduling algorithms in all experiment setup. Furthermore, DeEigbe shows great competitiveness when number of processors is 4, but performs worse than RS and P-RS in all other cases.

Why DeEigbe obtains higher schedulability ratio than other scheduling algorithms when number of processors is 4?

It is worth noting that for DeEigbe, increasing the number of processors does not always improve the schedulability ratio. In figure 7.1,the results are better when m increases from 2 to 4, but the schedulability ratio decreases at $n = 10$ when m increases from 4 to 8. We do not believe that this is due to experimental error, as we observe the same results at [16] as well. We examined some of the job lists generated by DeEigbe, and in the case of $m = 2$, almost all jobs are assigned to the first processor. In the case of $m = 8$, there are often latter processors that are not assigned any jobs.

DeEigbe uses first-fit partitioning to assign tasks to processors, place the task in the first processors that can accommodate it. There is a phenomenon where the algorithm may place all tasks in the first one when traversing the processors. Then, when the algorithm traverses all processors, it may place all tasks in the first processor or in the first few processors, so there are processors that are not assigned any jobs.

In summary, we believe that DeEigbe performs better when $m = 4$, for the range of n we evaluated, because the first-fit partitioning coincidentally allows for a fairer allocation of tasks to different processors, thus making the best use of processor resources. But not such case when $m = 8$ and $m = 2$.

Why RS and P-RS obtain higher schedulability ratios than other global scheduling algorithms?

Regardless of the number of processors, RS and P-RS perform significantly better than the other global algorithms, *i.e.* except for DeEigbe. Compared to other priority assignment algorithms such as ALAP, the fine-grained tuning of release-time makes it easier to successfully schedule a instance, although we also use ALAP. As we demonstrate in subsection 5.2.3, our solution allows more instances to be not be interrupted by instances generated by other tasks. These instances then complete before their deadline, without affecting the execution of the next generated instance.

Why P-RS performs no better than RS in most cases?

An interesting observation is that P-RS performs no better than RS in most cases. The difference between them is that P-RS incorporates a partitioning process. This demonstrates that 'less is more',it is not always advantageous to add restrictions to reduce uncertainty. We believe there are two reasons for this. RS tries to simulate the execution of jobs, and the method we use allocates jobs to processors as fairly as possible. However, jobs have execution variation and we cannot fully predict the execution of jobs and find the optimal solution for partitioning. The second and most critical point is that

the restrictions provided by partitioning can lead to a waste of computation resources. Suppose a processor is busy and there is a job waiting in its ready queue, but another processor is free at the moment. The task that needs to run cannot be migrated to that free processor.

8

Conclusion

In this chapter, we summarize our contributions in section 8.1, provide responses in section 8.2 to the research questions described in subsection 1.2, and make recommendations for future research in section 8.3.

8.1. Summary of Contributions

In this thesis, we identify the limitations of current priority assignment scheduling methods by exploring an anomaly case in subsection 5.1.2 and then propose our solutions. Furthermore, we develop a complete experimental environment for evaluating various scheduling algorithms.

Stacks and Job batch placing algorithm We have designed **stacks** and develop some special rules to define the valid placement of a job within the stacks. In order to find a new release time for each job, we propose the **job batch placing algorithm**, in subsection 5.2.2, which finds the appropriate placement in stacks for all jobs in a job batch and is consistent with the precedence constraints.

Reassembly stacking algorithm We propose a new scheduling algorithm for parallel DAG tasks in subsection 5.2.1. We parse all the jobs in a hyperperiod and submit them to stacks in a special packing and ordering. The new release times are obtained and then combined with the existing priority assignment method ALAP for scheduling.

Partitioned reassembly stacking algorithm We propose a new partitioned scheduling algorithm for parallel DAG tasks in subsection 5.3. While previous partitioned scheduling algorithms have done partitioning at the task level, we have innovated by applying partitioning to the job level. We still apply stack to simulate the multiprocessor platform, and the job batch placing algorithm allows for a fair distribution of all jobs to all processors. This approach allows each processor to work as fully as possible.

Evaluation framework We have developed an evaluation framework for the efficient and automated evaluation of various scheduling algorithms in chapter 6. Our experimental environment supports the specification of parameters and includes many of the scheduling algorithms mentioned in this thesis. Furthermore, it supports the automatic analysis and summarizing of experimental results to produce tables and graphs.

8.2. Research Questions

RQ1: *Can a more granular level analysis compare with only one DAG structure analysis for recurrent DAG tasks assist priority-based scheduling policy to improve schedulability?*

We propose a new job-level analysis method for recurrent DAG tasks, the **job batch placing algorithm**, based on which we propose the scheduling algorithm **reassembly stacking algorithm**. From figure 7.1 and 7.2 we observe that our algorithm has a higher schedulability ratio in all cases compared to ALAP. There is no doubt that the **reassembly stacking algorithm** benefits from such a job-level analysis method. A more granular level of analysis for recurrent DAG tasks is able to assist priority-based scheduling policy to improve schedulability.

RQ2: *How to improve schedulability of priority-based scheduling policies for recurrent DAG tasks by using intra-DAG offset?*

The **reassembly stacking algorithm** we propose is essentially a **release-time tuning** technique. Instead of just intra-DAG offsets, which are configured for each node, we implement a fine-grained offset tuning technique, where offsets are configured for each job. From the evaluation results, it is clear that offset tuning indeed improves the schedulability of priority-based scheduling algorithms.

RQ3: *Does the partitioning method will improve schedulability of our priority-based scheduling policy?*

We combine the partitioning method with reassembly stacking to propose the **partitioned reassembly stacking algorithm**, which adds the additional restriction that all jobs can only run on the processor to which they belong. The answer to the research question is that reducing uncertainty does not improve the performance of the algorithm. We provide a discussion of this in last paragraph in subsection 7.2.3.

In addition, our combination of the partitioning method with the priority-based scheduling algorithm does not improve the schedulability ratio. This suggests that not all cases of imposing more restrictions on jobs, such as modifying the release time and specifying a processor for a job, will provide a higher schedulability ratio.

8.3. Future Work

In the following, we propose three future directions for research. So far, with regard to $Stacks$, we have only considered WCET, but the tasks we are scheduling have execution-time variations. So it would be a more fine-grained solution to include the BCET of jobs in the $Stacks$ analysis.

Moreover, in figure 5.19 we notice that there is a ”hole” between 77 and 109 in $stack_2$ that has not been assigned a job. In our solution we did not consider filling this ”hole” with a suitable job. If this could be done, it would result in better processor optimisation and the completion of an instance's jobs sooner.

Finally, our solution is still a priority-based scheduling algorithm, and we have only tried ALAP to generate priorities for all nodes, which means that our solution can be extended to all priority generation algorithms. In the future, it will be a mutually beneficial process to apply better priority generation algorithms to our solution.

References

[1] Ishfaq Ahmad, Yu Kwong Kwok, and Min You Wu. “Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors”. In: *International Symposium on Parallel Architectures, Algorithms and Networks, I-SPAN* (1996), pp. 207–213. DOI: 10.1109/ISPAN.1996.508983.

[2] Benny Akesson et al. “A comprehensive survey of industry practice in real-time systems”. In: *Real-Time Systems* (2021). DOI: 10.1007/s11241-021-09376-1. URL: https://doi.org/10.1007/s11241-021-09376-1.

[3] Abu Asaduzzaman, Manira Rani, and Fadi N. Sibai. “On the design of low-power cache memories for homogeneous multi-core processors”. In: *Proceedings of the International Conference on Microelectronics, ICM* (2010), pp. 387–390. DOI: 10.1109/ICM.2010.5696168.

[4] Sanjoy Baruah. “Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms”. In: *Real-Time Systems Symposium* (2004), pp. 37–46. ISSN: 10528725. DOI: 10.1109/REAL.2004.20.

[5] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor scheduling for real-time systems*. Springer, 2015.

[6] Sanjoy Baruah et al. “A generalized parallel task model for recurrent real-time processes”. In: *Real-Time Systems Symposium* (2012), pp. 63–72. ISSN: 10528725. DOI: 10.1109/RTSS.2012.59.

[7] Sanjoy K. Baruah. “The Non-Preemptive Scheduling of Periodic Tasks upon Multiprocessors”. In: *Real-Time Syst.* 32.1–2 (2006), pp. 9–20. ISSN: 0922-6443. DOI: 10.1007/s11241-006-4961-9. URL: https://doi.org/10.1007/s11241-006-4961-9.

[8] Niu Bin et al. “Asymmetric software architecture design of High performance control chip applied in industrial control field”. In: *International Conference on Advanced Electronic Materials, Computers and Software Engineering (AEMCSE)*. 2021, pp. 916–920. DOI: 10.1109/AEMCSE51986.2021.00186.

[9] Enrico Bini and Giorgio C Buttazzo. “Measuring the performance of schedulability tests”. In: *Real-Time Systems* 30.1 (2005), pp. 129–154. DOI: 10.1007/S11241-005-0507-9.

[10] Giorgio Buttazzo and Anton Cervin. “Comparative assessment and evaluation of jitter control methods”. In: *Conference on Real-Time and Network Systems*. 2007, pp. 163–172. DOI: 10.1109/TII.2011.2123902.

[11] Giorgio C Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 3rd Edition (Real-Time Systems Series)*. 2011. URL: http://www.springer.com/series/6941.

[12] Daniel Casini et al. “Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions”. In: *Proceedings - Real-Time Systems Symposium* 2018-December (Jan. 2019), pp. 421–433. ISSN: 10528725. DOI: 10.1109/RTSS.2018.00056.

[13] Edward G. Coffman et al. *Bin packing approximation algorithms: Survey and classification*. Vol. 1-5. 2013. DOI: 10.1007/978-1-4419-7997-1_35.

[14] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. The MIT Press, 2009.

[15] Zheng Dong and Cong Liu. “An efficient utilization-based test for scheduling hard real-time sporadic DAG task systems on multiprocessors”. In: *Real-Time Systems Symposium* (2019), pp. 181–193. DOI: 10.1109/RTSS46320.2019.00026.

[16] Eghonghon-aye Eigbe. *Low-Overhead Non-Preemptive Scheduling of Real-Time Tasks upon Multiprocessor Platforms | TU Delft Repositories*. 2019. URL: https://repository.tudelft.nl/islandora/object/uuid%5C%3A523dc2df-c8f2-4b97-b1fb-55b52522264d.

[17] Juan Fang et al. “Performance optimization by dynamically altering cache replacement algorithm in CPU-GPU heterogeneous multi-core architecture”. In: *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2017), pp. 723–727. DOI: 10.1109/CCGRID.2017.54.

[18] Vincent François-Lavet et al. “An Introduction to Deep Reinforcement Learning”. In: *Foundations and Trends in Machine Learning* 11 (3-4 Dec. 2018), pp. 219–354. ISSN: 1935-8237. DOI: 10.1561/2200000071. URL: http://dx.doi.org/10.1561/2200000071.

[19] R. L. Graham. “Bounds on the performance of scheduling algorithms”. In: *In Computer and Job Scheduling Theory* (1976), pp. 165–227.

[20] Nan Guan et al. “Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling”. In: *Journal of Systems Architecture* 57 (5 May 2011), pp. 536–546. ISSN: 1383-7621. DOI: 10.1016/J.SYSARC.2010.08.003.

[21] Qingqiang He et al. “Intra-Task Priority Assignment in Real-Time Scheduling of DAG Tasks on Multi-Cores”. In: *IEEE Transactions on Parallel and Distributed Systems* 30 (10 Oct. 2019), pp. 2283–2295. ISSN: 15582183. DOI: 10.1109/TPDS.2019.2910525.

[22] Zhigang Hua et al. *Learning to Schedule DAG Tasks*. 2021. DOI: 10.48550/ARXIV.2103.03412. URL: https://arxiv.org/abs/2103.03412.

[23] Xu Jiang et al. “On the Decomposition-Based Global EDF Scheduling of Parallel Real-Time Tasks”. In: *Real-Time Systems Symposium* 0 (July 2016), pp. 237–246. ISSN: 10528725. DOI: 10.1109/RTSS.2016.031.

[24] EG Co man Jr, MR Garey, and DS Johnson. “Approximation algorithms for bin packing: A survey”. In: *Approximation algorithms for NP-hard problems* (1996), pp. 46–93.

[25] Navid Khoshavi, Hamid R. Zarandi, and Mohammad Maghsoudloo. “Control-flow error detection using combining basic and program-level checking in commodity multi-core architectures”. In: *IEEE International Symposium on Industrial Embedded Systems* (2011), pp. 103–106. DOI: 10.1109/SIES.2011.5953691.

[26] Yu Kwong Kwok and Ishfaq Ahmad. “Static scheduling algorithms for allocating directed task graphs to multiprocessors”. In: *ACM Computing Surveys (CSUR)* 31 (4 Dec. 1999), pp. 406–471. ISSN: 03600300. DOI: 10.1145/344588.344618. URL: https://dl-acm-org.tudelft.idm.oclc.org/doi/abs/10.1145/344588.344618.

[27] Hyunsung Lee et al. “A Global DAG Task Scheduler Using Deep Reinforcement Learning and Graph Convolution Network”. In: *IEEE Access* 9 (2021), pp. 158548–158561. ISSN: 21693536. DOI: 10.1109/ACCESS.2021.3130407.

[28] Hyunsung Lee et al. “Panda: Reinforcement Learning-Based Priority Assignment for Multi-Processor Real-Time Scheduling”. In: *IEEE Access* 8 (2020), pp. 185570–185583. ISSN: 21693536. DOI: 10.1109/ACCESS.2020.3029040.

[29] Jinkyu Lee. “Improved schedulability analysis using carry-in limitation for non-preemptive fixed-priority multiprocessor scheduling”. In: *IEEE Transactions on Computers* 66 (10 Oct. 2017), pp. 1816–1823. ISSN: 00189340. DOI: 10.1109/TC.2017.2704083.

[30] Yuxi Li. *Deep Reinforcement Learning: An Overview*. 2017. DOI: 10.48550/ARXIV.1701.07274. URL: https://arxiv.org/abs/1701.07274.

[31] Hongzi Mao et al. “Learning Scheduling Algorithms for Data Processing Clusters”. In: *Conference of the ACM Special Interest Group on Data Communication* (Oct. 2018), pp. 270–288. DOI: 10.48550/arxiv.1810.01963. URL: https://arxiv.org/abs/1810.01963v4.

[32] Alessandra Melani et al. “Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems”. In: *Euromicro Conference on Real-Time Systems* 2015-August (Aug. 2015), pp. 211–221. ISSN: 10683070. DOI: 10.1109/ECRTS.2015.26.

[33] Mitra Nasri and Bjorn B. Brandenburg. “An Exact and Sustainable Analysis of Non-preemptive Scheduling”. In: *Proceedings - Real-Time Systems Symposium* 2018-January (Jan. 2018), pp. 12–23. ISSN: 10528725. DOI: 10.1109/RTSS.2017.00009.

[34] Mitra Nasri and Gerhard Fohler. “Non-work-conserving Non-preemptive Scheduling: Motivations, Challenges, and Potential Solutions”. In: *Euromicro Conference on Real-Time Systems* 2016-August (Aug. 2016), pp. 165–175. ISSN: 10683070. DOI: 10.1109/ECRTS.2016.11.

[35] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. “A response-time analysis for non-preemptive job sets under global scheduling”. English. In: *Euromicro Conference on Real-Time Systems*. June 2018. DOI: 10.4230/LIPIcs.ECRTS.2018.9.

[36] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. “Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling”. In: *Euromicro Conference on Real-Time Systems*. Vol. 133. 2019, 21:1–21:23. DOI: 10.4230/LIPIcs.ECRTS.2019.21. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10758.

[37] Marco Di Natale et al. “Optimizing end-to-end latencies by adaptation of the activation events in distributed automotive systems”. In: *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS* (2007), pp. 293–302. ISSN: 15453421. DOI: 10.1109/RTAS.2007.24.

[38] Frank Reichenbach and Alexander Wold. “Multi-core technology - Next evolution step in safety critical systems for industrial applications?” In: *Proceedings - 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD 2010* (2010), pp. 339–346. DOI: 10.1109/DSD.2010.50.

[39] Abusayeed Saifullah et al. “Parallel Real-Time Scheduling of DAGs”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.12 (2014), pp. 3242–3252. DOI: 10.1109/TPDS.2013.2297919.

[40] Maria A. Serrano et al. “Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions”. In: *Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2016, pp. 1066–1071.

[41] Micaela Verucchi. *A comprehensive analysis of DAG tasks: solutions for modern real-time embedded systems*. 2019.

[42] Micaela Verucchi et al. “Latency-Aware Generation of Single-Rate DAGs from Multi-Rate Task Sets”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium* 2020-April (Apr. 2020), pp. 226–238. ISSN: 15453421. DOI: 10.1109/RTAS48715.2020.000-4.

[43] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. “Pointer Networks”. In: *Advances in Neural Information Processing Systems* 2015-January (June 2015), pp. 2692–2700. ISSN: 10495258. DOI: 10.48550/arxiv.1506.03134. URL: https://arxiv.org/abs/1506.03134v2.

[44] Luping Wang et al. “Metis: Learning to schedule long-running applications in shared container clusters at scale”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* 2020-November (Nov. 2020). ISSN: 21674337. DOI: 10.1109/SC41405.2020.00072.

[45] Ronald J. Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning 1992 8:3* 8 (3 May 1992), pp. 229–256. ISSN: 1573-0565. DOI: 10.1007/BF00992696. URL: https://link.springer.com/article/10.1007/BF00992696.

[46] Shuai Zhao et al. “DAG Scheduling and Analysis on Multiprocessor Systems: Exploitation of Parallelism and Dependency”. In: vol. 2020-December. Institute of Electrical and Electronics Engineers Inc., Dec. 2020, pp. 128–140. ISBN: 9781728183244. DOI: 10.1109/RTSS49844.2020.00022.