# Scheduling time-triggered tasks in multicore real-time systems: a machine learning approach

Félicien Fiscus-Gay
*Computer Science*
*Auckland University of Technology*
Auckland, New Zealand
felicien.fgay@gmail.com

***Abstract*—Tasks on real-time systems are becoming more and more complex and those system are becoming more and more multicore, complexifying the scheduling of such time-triggered tasks. To tackle this growth, the Directed Acyclic Graph (DAG) task model has been introduced and research has been done on designing heuristics to address the scheduling of DAGs on a multicore platform, but very few papers used machine learning for this problem. In this paper, the real-time DAG task scheduling literature is analyzed and a supervised machine learning model is designed using a graph convolutional network and evaluated for scheduling a single DAG task with the goal of minimizing its makespan using non-preemptive, global fixed-priority scheduling. The Integer Linear Programming method has been used to compute the output labels for the supervised model to learn on and a non-preemptive global and fixed-priority scheduler has been implemented to compute the exact makespan of a DAG task. The results showed that the supervised model doesn't learn and that the supervised learning method seems unfit for this kind of problem. The performance problem is mainly caused by the computation of the output label that could be and should be investigated further in the future.**

***Index Terms*—real-time system, scheduling, time-triggered tasks, DAG, multicore, machine learning, supervised learning**

## I. Introduction

Real-time systems are utilized in various domains such as air traffic control, public transportation, and automated vehicles. Unlike non-real-time systems, tasks in real-time systems must be both functionally correct and meet strict execution time constraints, known as deadlines. Failure to meet these deadlines can lead to severe consequences. The critical nature of these systems necessitates the design of a system architecture that focuses on time and incorporating fault tolerance to ensure high reliability.

One example of such architecture is the time-triggered architecture (TTA)[1][2], which offers a fault-tolerant communication protocol and a precise timing system to synchronize different electronic control units. Increasingly, these real-time system architectures are enhancing their computational resources by transitioning to multiprocessor systems. This shift from uniprocessor to multiprocessor systems addresses the growing complexity and computational demands of tasks executed on these systems (e.g., autonomous cars, computer vision systems), aiming to reduce both the execution time of these tasks and the required resources to run them[3].

Hence, an increasing number of real-time systems are utilizing multi-core hardware to parallelize their tasks and convert sequential programs into parallelized ones using frameworks such as OpenMP [1] to do so. Unfortunately, in most real-life scenarios, the number of available processors/cores is fewer than the number of tasks/subtasks that can be executed in parallel (i.e., independent tasks). This means that not all independent tasks can be executed simultaneously on the system, raising the question: which task should be executed first?

This question is particularly important in a real-time context because having the wrong execution order, or schedule, could lead to, at best, a slow system, and at worst, deadline misses, which can have fatal repercussions[4]. In the case of a self-driving car system, for instance, a slight delay of 500 ms in detecting a pedestrian crossing the road can, in some cases, be enough to drive over the pedestrian or cause a car accident. Note that the resources of real-time systems are scarce and limited, which is why using as little processing power as possible while ensuring that tasks meet their deadlines is of crucial importance.

The extreme case of this scheduling problem arises when only one processor is available to execute tasks. This is known as task scheduling on a uniprocessor, and [5] provided two major priority policies: Rate Monotonic (RM) and Earliest Deadline First (EDF) for scheduling independent periodic tasks. However, when considering multiple processors, the scheduling problem becomes much more complex, and different task models must be considered.

A prevalent task model when considering periodic tasks that can be parallelized is the Directed Acyclic Graph (DAG) task model which arises when a time-triggered task can be parallelized into subtasks which are the nodes of the graph, thus reducing the worst-case execution time of the reccurent task when executed on a designated set of cores. Those nodes have dependency constraints which are modeled by the directed edges between the nodes. The DAG task model is used to model tasks that are parallelizable[6], fitting the ever-increasing multicore architectures found in today's real-time system architectures.

[1]OpenMP (2011) OpenMP Application Program Interface v3.1. http://www.openmp.org/mp-documents/OpenMP3.1.pdf

Given that the problem of scheduling independent tasks or dependency-constrained groups of jobs (i.e., DAGs) is NP-hard[2][7] [8], people have resorted to either heuristics to partially solve the problem, or the optimal but not scalable Integer Linear Programming (ILP) method.

Consequently, machine learning will be considered here as it can better approximate the unattainable perfect solution while being scalable in terms of computing time after the training phase[9][10]. More specifically, the research questions are the following:

RQ1 What is the current state-of-the-Art for DAG tasks scheduling ?

RQ2 What machine learning techniques are used for DAG task scheduling ?

RQ3 Can machine learning be a better solution to schedule DAG tasks ?

- RQ3.1 Can a machine learning solution compare to state-of-the art heuristics for scheduling Directed Acyclic Graph tasks ?
- RQ3.2 Can a machine learning solution compare to an ILP solution while being more scalable ?

To achieve this, the background section will introduce various technical terms, concepts, and fundamental algorithms. Following this, a systematic literature review will be conducted to address RQ1 and RQ2, and finally, the artifact and experimental design, results, and conclusion will be presented to answer RQ3.

*The solution we propose has the following features..*

*The primary contributions of this paper are:*

## II. Background

DAG task scheduling introduces several fundamental concepts.

### A. DAG task model

A Directed Acyclic Graph (DAG) task $\tau = (G, D, T)$ is comprised of a directed acyclic graph $G = (V, E)$, a relative deadline $D > 0$ and a period $T > 0$. The graph $G$ has a set of vertices $V$ and a set of edges $E \subseteq V \times V$ that models the precedence constraints of the vertices, i.e., for all $(v_i, v_j) \in V^2$, with $i \neq j$, $(v_i, v_j) \in E$ if and only if $v_j$ must wait the end of $v_i$'s execution to start executing(i.e., $v_i$ is a precedence constraint for $v_j$). The nodes of the graph $G$ are characterized by their own worst-case execution time $C_{v_i}$ and $vol(G) = \sum_{v_i \in V} C_{v_i}$ is the worst-case execution time of the DAG task $\tau$, also called the total workload of the DAG task. In this paper, the words nodes, vertices and subtasks will be used to address the same thing, that is the vertices of the graph of a DAG task.

An example of a DAG task is shown in Figure 1, where the DAG task has a wcet of 24 time units.

[2]If a problem is NP-hard, it means that it is impossible to find a solution in polynomial time complexity, i.e., solutions are not scalable

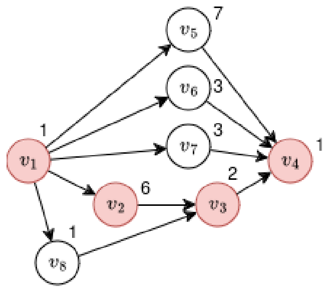


Fig. 1. a) DAG task $\tau$. The worst-case execution time (wcet) of each subtask written as an exponent and the nodes highlighted in red are the nodes in the critical path, the path of maximum length in terms of wcets.

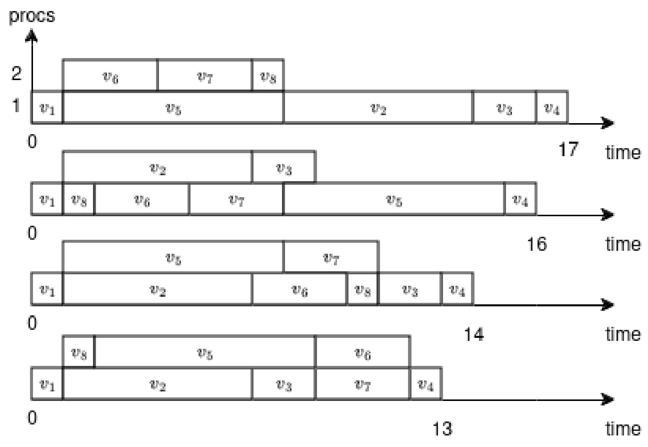


Fig. 2. b) Execution schedules for the DAG task above. Heavily inspired from Zhao et al. [11].

This concept will be the task model used in to conduct part of the systematic literature review (see Section III) and it also will be the task model used for designing the machine learning model (see Section IV).

### B. Makespan

The makespan or end-to-end response time of a DAG task is the amount of time it takes for all the subtasks in the DAG task to finish executing when given a schedule. For instance, for the task $\tau$ shown in Figure 1, the makespan of $\tau$ for the first schedule (from top to bottom) shown in Figure 2 is 17. Notice that Figure 2 shows multiple ways of scheduling the DAG task, the bottom one yielding a makespan of 13.

This is a key measurement when scheduling the nodes of a single DAG task, also called intra-task scheduling (see Section III), and it will be the main efficacy criteria when comparing the machine learning model with state-of-the-art heuristics and ILP (see Section IV).

### C. Intra-task and inter-task scheduling

When considering the DAG task model, two concepts arise : intra-task scheduling and inter-task scheduling.

Intra-task scheduling of DAG tasks is when only the execution order of the nodes in a single DAG task is considered.

The goal of such a problem being to minimize the makespan of a single DAG task.

Inter-task scheduling, on the other hand, is when multiple DAG tasks are considered and the goal then becomes to maximize the acceptance ratio (see below) of the scheduling algorithm.

### *D. NP-hardness intuition and problem motivation*

The problem of intra-task scheduling of DAG tasks, that is, scheduling the nodes of a single DAG task so that the makespan is minimized, is a NP-hard problem[8][7] but can also be intuitively seen as a NP-difficult problem when looking at the schedules shown in Figure 2. Indeed, when looking at a single DAG task, a simple yet efficient way of scheduling the nodes is looking at a critical-path-first execution order(CPFE). The critical path, highlighted in red in Figure 1, is the longest path of the graph in terms of wcets. The CPFE principle is applied in the third schedule (from top to bottom) in Figure 2 and achieves a makespan of 14. But this is not the minimum makespan as the fourth schedule yields a makespan of 13.

This is due to the fact that, although executing the critical path first is the best strategy[11], one needs to take into account the dependency constraints and thus the global structure of the graph. The fourth schedule in Figure 2 achieves a makespan of 13 by prioritizing the execution of node $v_8$ before node $v_5$ which is crucial to enable $v_3$ to execute sooner than in the third schedule. Typically in graph theory, when the global structure needs to be known in advance in order to find the best solution, it often means that the problem is NP-hard (e.g., the problem of coloring a graph[12] or coloring a path of a graph[13]). Therefore, researchers have resorted to heuristics to solve the problem(see Section III).

### *E. Utilization factor*

The utilization factor represents the percentage of processing time that a taskset $(\tau_1, \cdots, \tau_n)$ will utilize. Formally, it is defined as

$$U = \sum_{k=1}^{n} \frac{vol(G_k)}{T_k} \tag{1}$$

where $U$ is the utilization factor. This concept is significant because, when evaluating a scheduling algorithm $S$, we desire $S$ to effectively schedule tasksets that maximize the utilization factor $U$. Consequently, the higher the utilization factor bound for $S$, the more efficient the scheduling algorithm. Additionally, this concept is valuable in real-time systems where processing resources are often limited and expensive, making it crucial to maximize their usage.

This concept is also used either as a measurement when comparing two scheduling algorithms and considering their utilization bound(see Section III), or used as a parameter to generate tasksets or DAG tasks with a fixed utilization (see Section IV).

### *F. Capacity augmentation bound*

Another measurement used when scheduling event-chains, or DAG, tasks is the capacity bounds, or capacity augmentation bound, which compares resource use to a theoretically optimal scheduling algorithm. It can also be used as a simple schedulability test. It is often refer to as a $\beta$ coefficient and the lower $\beta$ is, the better the scheduling algorithm.
This metric is used for DAG tasks or event-chains of tasks.

### *G. Acceptance ratio*

When dealing with several independent DAG tasks or tasksets, the acceptance ratio is often used to measure the performance of a scheduling algorithm (see Section III). It consists of looking at a number of generated tasksets (or DAG tasks) and calculating the amount of schedulable (i.e., the schedule produced doesn't lead to a deadline miss) tasksets compared to the total amount of taskets. The resulting percentage is the acceptance ratio and the closer it gets to 100% for a scheduling algorithm, the better the scheduling algorithm.

This concept is also used as a metric, to assert the efficiency of scheduling algorithms when considering inter-task scheduling (see Section III).

While the acceptance ratio, also called system schedulability, is used to measure the performance of scheduling algorithms for inter-DAG task scheduling (i.e., scheduling multiple DAG tasks), the makespan and the capacity bound are only used for DAG tasks and tasksets representing chain of events.

## III. Related Works

### *A. Systematic Literature Review process*

This SLR aims at tackling RQ1 and RQ2. More precisely, the following research questions will be answered:

RQ1 What is the current state-of-the-Art for DAG tasks scheduling ?

RQ2 What machine learning techniques are used for DAG task scheduling ?

It will also be shown how the literature doesn't provide a complete answer to RQ3, hence the contributions of this paper.

From these research questions, several concepts have been isolated, namely, time-triggered or DAG tasks, the nature of the system (real-time multicore system), the scheduling of tasks, and machine learning. The recording of the search results were done using the BibTeX LateX plugin combine with the google scholar "cite" feature.

Searching was conducted using the IEEE and ACM databases. According to the concepts identified above, the keyword chain used for searching was "("real-time" OR "real time") AND "system" AND ("time-triggered" OR "time triggered" OR "DAG" OR "Directed Acyclic Graph" OR "event chain" OR "event-chain") AND "task" AND ("scheduling" OR "scheduler" OR "schedule") AND ("multi-processors" OR "multi-cores" OR "multi processors" OR "multi cores" OR

”multi-processor” OR ”multi processor” OR ”multi-core” OR ”multi core”)”. The search produced 1,259 results on the IEEE database which was reduced tp 515 papers when considering only articles published in the past 5 years.

Then the following exclusion criterias were used to filter out the rest of the articles, bringing the number of papers down to 19 (see Figure 3).

EC1 Not focusing on homogeneous multicores and hard RTS
  ”Heterogeneous” not in the title nor the abstract.
  ”mixed critical*” not in the title nor the abstract

EC2 Not focusing on scheduling
  ”scheduling” or ”scheduler” or ”schedule” in the title
  ”energy” not in the title
  Focus on conference and journal papers

EC3 Not focusing on real-time systems, not proposing a scheduling algorithm, not using DAG or event-chain tasks.

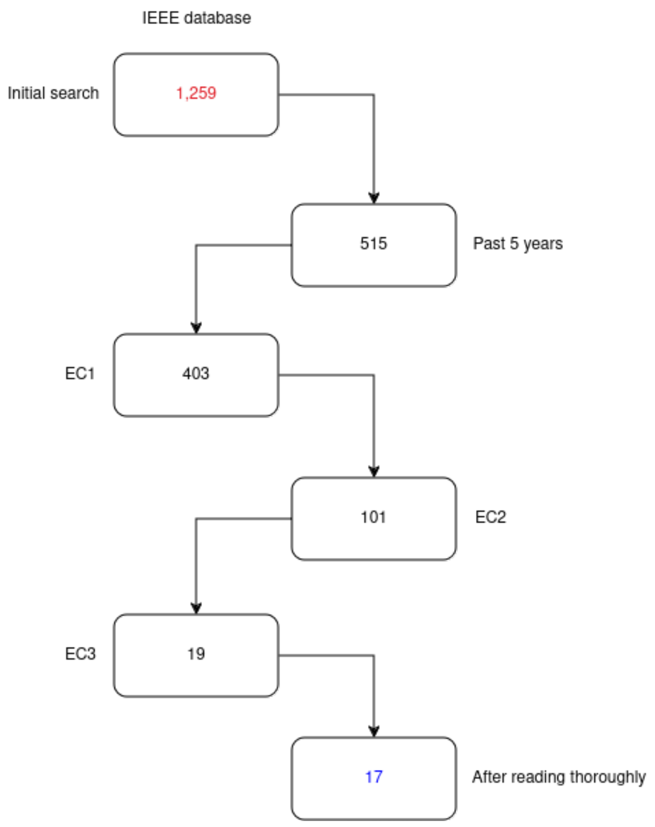


Fig. 3. SLR process diagram. EC stands for Exclusion Criteria, which are listed above.

### B. *Findings of the Literature Review*

#### *1) Non-Machine Learning techniques:*

Guan et al. [14] use fluid scheduling to schedule multiple DAG tasks on a multicore system. Fluid scheduling has been used in previous work for independent time-triggered task scheduling[15][16] but very few consider DAG tasks. Fluid-scheduling is known for producing optimal scheduling algorithms. Their method decomposes a DAG task into several sequential segments in which the subtasks will execute according to the fluid scheduling model. Although their algorithm significantly outperforms existing algorithms, the main limitation that is common to all fluid-based scheduling algorithm is the runtime overhead induced by the fluid-scheduling model. Although authors in [14] briefly explain how to transform their scheduling algorithm to a non-fluid one for practical implementation, they do not evaluate the overhead caused by the frequent task migrations and preemptions. This overhead can lead to deadline misses on systems where the migration cost is high which is why it is important, especially with fluid-based algorithms, to consider this overhead[17]. Also, their algorithm only considers DAG task with implicit deadlines (D = T) which makes the response-time analysis simpler but to the cost of not covering all types of tasks that can execute on real-time systems.

As a follow up, Guan et al. [18] extend the fluid scheduling algorithm in [14] to constrained and arbitrary deadline tasks, especially focusing on DAG tasks with a deadline greater than their period, thus generalizing their previous fluid algorithm to all types of periodic tasks. Their main contributions are their new scheduling algorithm that performs better than existing methods in terms of acceptance ratio, and producing the first theoretical capacity bound for DAG tasks with deadlines greater than their periods. However, the authors still don’t provide any evaluation on the amount of runtime overhead their scheduling algorithm implementation produces which generally lowers the actual acceptance ratio of the algorithm.

Instead of considering fluid-scheduling, a popular scheduling method is federated scheduling. Federated scheduling is based on the idea of assigning heavy tasks ($U > 1$) to multiple cores for the whole duration of the tasks’ executions, and assigning light tasks ($U \leq 1$) to execute on cores that have not been assigned a heavy task. Although it is popular, it suffers from a resource wasting problem, especially when the difference between the critical path’s length and the deadline is small, which many papers aim at solving [19] [20] [21] [22] [23] [24].

Jiang et al. [20], for instance, consider federated scheduling and GEDF and introduces a better metric called the util-tensity bound that extends the concept of capacity bound to have a better schedulability test. Based on this newly derived bound, the authors propose an extension to the classic federated algorithm, with very low tensity tasks being scheduled with GEDF, tasks with high-utilization and relatively high tensities are scheduled using the classic federated scheduling and low utilization tasks with relatively high tensities are scheduled using partitioned-EDF. Their algorithm, based on their newly derived bound, effectively improves the system schedulability of DAG tasks and reduces the resource wasting problem of federated scheduling. The main limitation of this paper is that they only consider GEDF for their util-tensity bound and also only consider implicit deadline DAG tasks which doesn’t represent tasks that need to have a deadline lower than their period, for example.

This problem of resource wasting in federated scheduling is also tackled by Kobayashi and Azumi [23] where they propose a federated and bundled-based scheduling algorithm which enhances the schedulability of DAG tasks compared to existing federated scheduling algorithms. Their method consists of using federated scheduling for tasks with high critical path to deadline ratio and bundled scheduling for tasks with low critical path to deadline ratio. Unfortunately, this paper only looks at 3 DAG tasks to evaluate their algorithm which is a really small amount and is not representative of the different DAG tasks that can exist and leaves space for observational bias.

Jiang et al. [21] take another approach by proposing a virtually-federated scheduling algorithm that leverages the advantages of federated scheduling while improving the acceptance ratio for DAG tasks, outperforming existing algorithms. Their approach consists of adding a virtual layer of processors, on top of physical processors, and apply their federated-based scheduling algorithm on those virtual processors, thus enabling tasks to share a physical processor even though they are assigned to different virtual processors. The main drawback in [21] is that the authors only consider the heavy tasks (i.e., $U > 1$) and do not take the light tasks into account, meaning it doesn't consider tasks that only need one processor to execute.

To fix this limitation, Jiang et al. [22] extend their previous work[21] so that it considers both heavy and light tasks. The resulting virtually federated scheduling algorithm clearly outperforms any other federated-based scheduling algorithms in terms of acceptance ratio. However, they still only consider implicit or constrained deadline tasks and they don't provide any evaluation of the run time overhead their algorithm might induce, to compare with algorithms currently used in real-time systems, such as GEDF.

Gua n et al. [19] consider arbitrary deadline tasks and especially DAG tasks that have a deadline that is greater than their period. They introduce a new federated scheduling algorithm that takes those type of tasks into account and compare it to existing global or federated scheduling approaches for arbitrary deadline tasks, significantly outperforming most of them in terms of acceptance ratio. Their approach consists of using this new proposed aglgorithm for heavy tasks that have a deadline bigger than their period, then using classic federated scheduling for the heavy tasks with a constrained deadline, and finally using EDF-First-Fit (EDF-FF) for the light tasks. Although the fluid-based method in [18] outperforms this new federated scheduling algorithm, the impracticality of fluid-based algorithm makes this algorithm the current best, in terms of acceptance ratio, for dealing with arbitrary deadlines. The main limitation of this work is that it doesn't tackle the resource wasting problem that classical federated scheduling, or their new algorithm, has or can potentially have, but only focuses on providing an algorithm for arbitrary deadline tasks.

For constrained deadline DAG tasks, He et al. [24] propose a federated-based scheduling algorithm that outperforms on average by more than 18% previous SOTA[22] in terms of acceptance ratio, making this work the current SOTA for constrained deadline multi-DAG scheduling. Their approach uses the notion of degree of parallelism, which they define rigorously, to improve the classic federated scheduling way of choosing the number of cores to assign each heavy tasks. They also propose a new response-time bound for constrained deadline DAG tasks based on this defined notion. Although their method clearly stands out, they don't consider intra-task scheduling at all when their motivation came from the notion of degree of parallelism being used but wrongly defined in previous intra-task scheduling work[25][11].

Federated scheduling isn't the only method used, Jiang et al. [26], for instance, propose a decomposition-based approach to schedule multi-DAG tasks as well as a metric for testing the schedulability of tasks. Their decomposition strategy proves to be the most efficient , according to the defined metric, and the scheduling algorithm derived from it shows promising results in terms of acceptance ratio. Their decomposition strategy basically works by first definin execution segments and then then assigning subtasks to those segments using the laxity[3] of those subtasks so that no segments are overloaded with workload. The main limitation of this work is that they only look at GEDF variants for priority assignment and do not evaluate their decomposition method using other scheduling heuristics for multi-DAGs. Most of the articles presented up to now tackle inter-task scheduling, not considering the intra-task execution schedule.

Indeed, intra-task scheduling[27][28] [29][9][10] [30] is often tackled as a separate problem due to the dependency constraints.

Xiao et al. [28], for instance, introduce a scheduling algorithm , 'MAS', that shortens the makespan of periodic DAG tasks compared to the classic EDF dynamic priorty scheduling technique. Their algorithm is based on a clustering approach, combined with a technique called task duplication, and evaluate their results on an actual simulation object for real-time scheduling. Unfortunately, their evaluation is only based on a single DAG task and they only compare their algorithm with EDF. Their algorithm also shows a scalability problem compared to other existing algorithm with comparable results.

A scalable way of scheduling sub-tasks of a DAG is looking at priority-list scheduling which He et al. [27] use to propose an algorithm that effectively outperforms other intra-task DAG scheduling algorithms in terms of makespan. Their priority assignment algorithm is based on the length of the paths passing through each vertex. The longer the maximum length, the higher the priority. This effectively takes advantage of the inner graph structure to optimize the intra-task execution order, which is something that hasn't been done before this work. Although the authors compare their results with existing scheduling heuristics, they do not consider the, mathematically optimal, ILP method to compare their makespan results with the mathematically minimum makespan.

[3]Laxity is the gap between the total execution time of a task (potentially comprising the I/O delays) and its deadline.

This priority-list scheduling approach is also used by [25], extending their previous work[11] which develop a priority assignment based on the critical-path execution first (CPEF) concept, effectively outperforming He et al. [27] in terms of makespan and providing a federated-based multi-DAG scheduling algorithm, compatible with this new priority-list scheduling algorithm. Their multi-DAG scheduling algorithms also outpeforms the multi-DAG algorithm used in [27]. Their method uses the vertices in the critical path as producers of workload for the parallelizable vertices to consume. For multi-DAG they look at assigning processors to DAG tasks, like in federated scheduling, using a parallelism-aware workload distribution model that uses their Concurrent Producer Consumer (CPC) model to assign cores while minimizing the inter-task workload interference. One limitation of their work is that, although they consider constrained deadlines for the response-time analysis, they only use implicit deadline DAG tasks for evaluating the system schedulability of their multi-DAG scheduling algorithm.

Those intra-task scheduling papers only consider the theoretical makespan when task migration and preemption is instantanious, but they don't take the runtime overhead into account. One major factor in the runtime overhead is the communication delays between each subtasks of a DAG task, which are rarely considered. To that end, Shi et al. [29] propose an extension of the DAG task model to add execution groups that bind groups of subtasks to a single physical core, thus reducing inter-core communication which can cause major communication delays depending on the topolgy of the system. They also introduce a scheduling algorithm and compare the makespan of their approach to existing methods such as federated scheduling and critical path-based scheduling. Their method shows comparable results while minimizing the communication overheads. However, the evaluation is only done on 100 generated DAG tasks which is a small sample size compared to the other papers[25][27], exposing the experiment to obersvational biases. Also, although they introduce a way to extend their approach to schedule multiple DAGs, they do not provide any evaluation of that.

*2) Machine Learning techniques:*

Only a few papers consider machine learning to solve the scheduling problem.

Lee et al. [10], for instance, design a deep reinforcement learning (DRL) model called GoSu which takes a DAG task as input and outputs a priority list of the DAG's subtasks. The makespan resulting from this priority-list is then compared to the results in [25] and [27] and the DRL model proposed in [10] outperforms them by up to 3%. The model is comprised of a graph convolutional network layer to encode the graph structure information, and a sequential decoder layer based on the attention-mechanism, which produces a priority list of the vertices. The reinforcement learning uses the makespan as the reward to minimize and the REINFORCE algorithm is used to find the best policy. Although the time it takes for the model to run is measured, no comparison with the ILP method is done and there is no evaluation of the scalability of the model when increasing the amount of cores in the system or the amount of subtasks in a DAG task. But Lee et al. [10] isn't the only work considering DRL as a method for DAG intra-task scheduling.

Indeed, Zhao et al. [9] also use the DRL approach to tackle the intra-task scheduling problem and compare their results to the makespan obtained by solving the equivalent ILP problem. Their model achieves up to 75% of makespan reduction compared to ILP, that is, the makespan produced by the ILP method is 25% smaller than the makespan produced by their DRL model, which is a relatively good performance as the ILP approach gives the mathematically minimum makespan. The more important result, however, is that when you increase the subtasks in the DAG tasks, the ILP method explodes in terms of computing time when the DRL approach gives a result in a relativley short time, making it scalable, unlike ILP. The model uses a combination of a graph neural network with attention layers to better capture the structure and dependency information of the DAG task. The makespan is also used for the reward function but Soft Actor Critic algorithm is used for training the model. Unfortunately, the paper doesn't provide an evaluation when increasing the number of cores in the system and also doesn't compare their model to SOTA heuristics, which also aim to approximate the optimal solution given by the ILP method.

Guan et al. [30] focus more on optimizing communication delays by looking at the real-time simulation system FRTDS and using the I/O usage and ram allocation to construct a cost function, which is then used as a reward for their proposed DRL model to schedule DAG's subtasks. The cost is divided into a current cost, what we know, and the future cost which is predicted using the subtask's successors. The model performs better, in terms of makespan, than existing scheduling algorithms implemented on the FRTDS platform but because of the RL process accumulating the previous subtask's execution as experience to learn, the method uses a lot of memory which affects the speed of execution.

Table I gives a summary of the findings.

According to these findings, the current state-of-the-Art for DAG task scheduling (RQ1) seems to be a federated-based scheduling approach for inter-task scheduling and a global priority-list approach for intra-task scheduling [24][25].

In terms of machine learning techniques (RQ2), every article found uses deep reinforcement learning for scheduling DAG tasks. More specifically, the model proposed is often a combination of a kind of graph neural network and attention layers to produce a priority-list of the subtasks of the DAG task.

Multiple limitations were found for each paper. Notably, the type of task that is considered which mostly is constrained or implicit deadline tasks and arbitrary is rarely considered due to the complexity it adds to the scheduling analysis. Also, there hasn't been any use of supervise machine learning for real-time task scheduling and only one paper looked at comparing a machine learning solution to the optimal schedule provided by the ILP methodology.

TABLE I
SLR SUMMARY TABLE

| Ref. | Motivation | Contribution(s) | Limitation(s) | Methodology Summary |
|---|---|---|---|---|
| [14] | DAG tasks scheduling is getting more popular and fluid scheduling performs great theoretically | Provide a DAG-fluid scheduling algorithm that performs way better in terms of acceptance ratio then previous algorithms | Fluid scheduling is unpractical and introduces a lot of overhead and task migrations, also only for implicit deadlines tasks | fluid-based algorithm where it decomposes a DAG's subtasks into multiple sequential segments |
| [27] | DAGs are popular but no one looked at the intra-task execution order to leverage the graph structure | proposes a priority list scheduling algorithm for a single DAG task which performs better than SOTA in terms of makespan | no comparison with optimal priority assignment algorithms / optimal schedules. | uses the length (in terms of wcets) of each paths passing through the current vertex to assign the priority to the current vertex, the higher the length, the higher priority |
| [23] | Federated scheduling for DAG tasks is has proved efficient but for tasks where the difference between the critical path and the deadline is small, it can lead to over-allocating cores. | proposed a fedrated and bundled-based scheduling algorithm to avoid this problem and enhanced the schedulability of DAG tasks using their algorithms | They only compare their method with an example of a dag task set comprised of 3 dag tasks. | Uses federated scheduling for tasks with high critical path to deadline ratio and bundled scheduling for tasks with low critical path to deadline ratio. |
| [28] | DAG task scheduling is NP-hard so one can only approximate the optimal algorithm (when considering polynomial timed algorithms) and not a lot has been done on scheduling parrallel reccuring tasks | Introduces a scheduling algorithm 'MAS' that shortens the makespan of recurring DAG tasks compared to EDF | only compares EDF and MAS using one example of a DAG task for makespans and also only compares with EDF. Even though MAS shortens the makespan, it is less scalable than comparable algorithms. | The MAS algorithm integrates clustering, scheduling, and task duplication techniques and is evaluated using a TMS320C6678 simulation for measurements closely aligned with real-world results. |
| [20] | The capacity-bound measures performance and tests schedulability for DAG scheduling, but it may exclude schedulable tasks by using the same bound for normalized utilization and tensity. | Introduces a new bound called the util-tensity bound which proves to be a better schedulability test for GEDF with federated scheduling. | only looks at GEDF with federated scheduling and not other scheduling algorithms. | Uses GEDF for low-tensity tasks, federated scheduling for high-utilization, high-tensity tasks, and partitioned EDF for low-utilization, high-tensity tasks. |
| [26] | Decomposition-based scheduling can improve schedulability for DAG task scheduling but can also worsen it. It is, along with global scheduling, one of the main method to schedule DAG tasks. | Developed an efficient decomposition strategy and schedulability test. The resulting GEDF-based scheduling algorithm shows promising acceptance ratios. | Only looks at GEDF variants which is based on the EDF heuristics for priority assignments. | The decomposition works by first defining execution segments and then assigning subtasks to those segments based on their laxity so that there are no segments overloaded with workload. |
| [24] | The notion of degree of parallelism has been used for DAG task scheduling but lacks a clear definition in the research community. | Proposes a new response-time bound for DAG tasks as well as a new scheduling algorithm based on federated scheduling that outperforms the SOTA by more than 18% on average | They don't say which intra-task scheduling algorithm is used (just that it's work-conserving) and they don't consider intra-task scheduling. | They modify federated scheduling by optimizing core allocation for heavy tasks based on the degree of parallelism. |
| [29] | Many DAG intra-task scheduling algorithms overlook inter-core communication delays. In robotics and automotive applications, grouping subtasks on a single processor can reduce these delays by using the L1 cache. | Extend DAG to EG-DAG to bind subtasks to a single core, reducing communication delays. Their new scheduling algorithm shows similar performance to existing methods with lower overhead. | The evaluation has been done using only 100 DAG tasks which is quite low to cover all different types of DAG tasks. They propose a way to schedule multiple DAGs but do not offer evaluation results for that. | They use list scheduling with one list per execution group and use worst-fit heuristic to map the execution groups to the processors. |
| [19] | Federated scheduling works well for constrained deadlines but struggles with arbitrary deadline DAG tasks, especially with long WCET. Allowing job migration leads to pessimistic schedulability analysis. | Propose a new federated scheduling algorithm for arbitrary deadline DAG tasks with long WCETs. It outperforms other algorithms in acceptance ratio. | Doesn't tackle the problem of resource wasting when using federated scheduling or their new version of it. | The new federated algorithm addresses tasks with deadlines longer than periods and high densities, using standard federated scheduling for tasks with shorter deadlines and EDF-FF for low-density tasks. |
| [9] | The NP-hard nature of DAG multicore scheduling makes ILP solutions time-consuming, leading researchers to explore scalable heuristics. | Uses Deep Reinforcement Learning to learn an optimal scheduling policy for DAG tasks, comparing it to the ILP method. | Doesn't compare the machine learning method with SOTA heuristics, but only compares with ILP. | They use a Graph Neural Network with attention layers to capture structure and dependencies, maximizing the negative makespan as the reward function. |
| [31] | Current methods for allocating shared resources in multi-core real-time systems use static analysis or heuristics, which may not cover all scenarios, leading to higher WCETs and reduced schedulability. | They use Deep Reinforcement learning to propose a holistic scheduling and allocation framework and their model shows better schedulability than existing methods. | Only considers independent periodic tasks and also only considers even-EDF and even-RM when comparing schedulability performance. | The platform has an LLC architecture with a shared memory bus. The DRL model uses MLP and proximal policy optimization to generate time-triggered schedules and memory allocations. |
| [25] | A previous work done by Zhao et al. [11] introduced a fixed-priority scheduling algorithm for DAG intra-task scheduling which performed better than SOTA but didn't extend the method to multi-DAG scheduling. | "Extends the CPC model from Zhao et al. [11] to multi-DAG scheduling with a Parallelism-aware workload distribution algorithm, improving system schedulability and outperforming existing methods. | Considers constrained deadlines DAG tasks for the analysis but only consider implicit deadlines DAG tasks for the experiment evaluation. | Uses a critical path first model, prioritizing nodes on the critical path and allocating parallel execution time to subtasks. For multi-DAG scheduling, they use a federated-like approach based on the degree of parallelism of the DAG tasks. |

| | | | | |
|---|---|---|---|---|
| [10] | Several heuristics for DAG intra-task scheduling have been used but no scalable optimal scheduling algorithm exists. | Propose a DRL-based model for computing intra-task priorities in DAG scheduling, outperforming SOTA by up to 3% in makespan. | No ILP method comparison for minimum makespan is made, and scalability of the GoSu DRL model with more cores/subtasks isn't shown. | The network uses a Graph Convolutional Network and an attention-based decoder to generate a priority list, trained with REINFORCE using negative makespan as the reward. |
| [22] | Virtual scheduling, using threads as virtual processors, has been considered in the past but never for DAG task scheduling. The similar federaded scheduling method suffers from resource wasting. | Use the concept of virtual processors to provide a virtually-federated scheduling algorithm which significantly outperforms other federated scheduling methods in terms of acceptance ratio. | Only considers implicit and constrained deadline DAG tasks and doesn't consider the running time overhead that the proposed method induces. | Introduce active and passive virtual processors (VPs) per core. The active VP executes high-priority tasks, while unused active VP time is treated as a passive VP for low or high-priority tasks. |
| [18] | Most DAG studies use implicit deadlines, with few focusing on arbitrary deadlines, especially when deadlines exceed the task period. Fluid scheduling shows promise but is only applied to implicit deadlines. | Propose a fluid scheduling algorithm for constrained and arbitrary deadline DAG tasks, introducing the first capacity bound for deadlines longer than periods. It outperforms SOTA in acceptance ratio (as of 2022). | As for every fluid scheduling based algorithm the issue of runtime overhead is not entirely considered as they don't evaluate this metric. | They first decompose each DAG task into segments of sequential tasks and then assign execution rates to each tasks or threads. Those two steps aim at producing a fluid schedule so that it appears as though each DAG task is continuously running on the cores. |
| [30] | Most scheduling algorithm that consider resource use consider it as constraints rather than considering them as part of the scheduling decision process. | Propose a DRL-based algorithm for task scheduling on the FRTDS simulation system, outperforming existing algorithms in makespan for single DAG tasks. | The reinforcement learning algorithm uses the previous tasks' execution as experience which implies a lot of memory usage, thus affecting the speed of execution. | It uses I/O and RAM allocation to create a cost function as the RL reward, combining current costs with future costs predicted from successor subtasks. |
| [21] | Federated scheduling has shown great potential for scheduling DAG tasks but suffers from a resource wasting problem which has been addressed but to a limited extent. | Propose a virtually-federated scheduling algorithm that reduces resource wastage while maintaining federated scheduling's benefits, and outperforms existing algorithms in acceptance ratio. | Only considers heavy tasks and constrained deadline DAG tasks. | Introduce Active and Passive VPs per core. Active VPs handle primary tasks and use excess capacity for Passive VPs. Allocation is based on deadlines, critical path length, and task usefulness. |

## IV. Research Methodology

To answer the research questions defined in section I, the right research methodology needs to be selected to conduct systematic research. Multiple such methodologies exists but the following four will be evaluated and the ones which fit best to this research will be selected.

### *A. Systematic Literature Review*

The systematic literature review (SLR) methodology aims at looking at the current state-of-the art in a specific domain by selecting a range of articles in the scientific literature[32]. The main difference between a simple literature review is that the results need to be reproducable, that is, one can get the exact same results by simply following the different steps described by the one writing the SLR. This is done by first defining a set of keywords to search for, i.e., a search string, and then identifying the databases to search on. The resulting articles are then filtered out using exclusion and/or inclusion criteria to narrow down the number of papers to review. Examples of such criterias are restricting the publication year-range, excluding certain types of articles (i.e., conferences, early-access, etc.), etc. The remaining articles are then screened by first reading their abstract and then, from the resulting filtered articles, their entire content, which further filters the articles found through the initial search. This process gives a final list of papers to review and compare against each other to correctly answer the research questions. It is especially useful to answer research questions such as RQ1 and RQ2 and to have a good representation of the state-of-the-Art but often will show the research gaps that exists in the literature, thus not fully answering a specific research question. To address these gaps, other research methods need to be considered.

### *B. Design Science*

The design science methodology is focused on the creation and evaluation of artifacts intended to solve practical problems. It emphasizes the development of knowledge through the design and implementation of innovative solutions, such as algorithms, methods, tools, and frameworks, particularly in fields like software engineering[33]. For instance, one might design a new machine learning architecture or a new algorithm to better answer a specific problem, while rigorously detailing the design and evaluation aspect of this new technology or system. Indeed, in this methodology, the designed artifact(s) need(s) to answer a precise problem, defined beforehand, and choices in the design must be clearly justified. The evaluation of the artifact(s) needs not only to answer research questions, but also demonstrate practical applications, thus balancing methodological rigor and practical utility. In our case, the problem of scheduling DAG tasks has important practical applications and according to the SLR, no one looked at supervised learning as a solution to generating priority-lists for global fixed-priority scheduling. The non-existence of a supervised learning model implies it needs to be created, or designed, which is exactly the purpose of the design science methodology. Hence, RQ3 will be answered by designing a supervised learning model and evaluating this potential solution to the fixed-priority DAG task scheduling problem.

### C. Case study

A case study is useful at providing a great understanding of a real-life scenario that happened or that is currently happening, illustrating a specific issue with a real-life context. In a case study, you look closely at real-life observations to evaluate or investigate a design in the context of a real-life scenario[34]. It is especially used in the medical field to study and investigate a group of people and their eating habits or look at the effect of placebo in a control group compared to a medical subtsance to be tested on a test group. One can also use a case study methodology to study a specific phenomena and how an application react to it which can lead, in the case of a software application, to finding limitations or missing elements. Another use of the case study methodology is to address one or more real-life issues by analyzing a system using multiple datasets. For instance, a case study might design a benchmark for testing certain types of applications. In this research, no real-life scenario will be considered as the data collected will be via a DAG task generator because real-life data collection is very time-consuming and isn't the focus of this study. Indeed, the focus will be on designing an artifact, rather than investigating an existing one, simply because there aren't any supervised learning model that has been found in the literature (see Section III). Therefore, the case study approach will not be used in this research.

### D. Experiment

The experiment, or controlled experiment, methodology aims at identifying cause-effect chains by defining hypotheses and control variables on a specific system, and running multiple tests or experiments while having fixed and varying variables to assert their role in the system's behaviour[35]. The controlled experiment approach can provide insights on relationships between the controlled variables and the system which contributes to the body of knowledge of the specific system. This can be used, for instance, to observe and explain the impact of hyperparameters of a machine learning model on its performance. This approach has its drawbacks, the main one being that the experiment always has biases and doesn't fully reflect the reality of the matter. This drawback can be exacerbated when considering a small sample size or forgetting to take into account controlled variables that impact the results of the experiment, which is why great care needs to be taken for the design phase of the experiment(s). Although, the controlled experiment approach could be used in this research to validate or invalidate hypotheses about the designed AI model performances, especially when compared to the ILP method by experimenting with different values and parameters for the model and the dataset generation, the time constraints as well as evaluation problems made possible only the model's evaluation. Hence, the controlled experiment will not be used in this paper.

The SLR can answer research questions 1 and 2, by first collecting and comparing the different papers in the literature and then identifying not only the state-of-the-Art for scheduling DAG tasks on real-time systems, but also the machine learning techniques that are currently used in the literature as well as their drawbacks and limitations. The fact that only very few papers use machine learning to tackle the problem of real-time DAG task scheduling motivates the design of a new machine learning model to better answer RQ3 which implies applying the design science methodology to provide a more complete answer as to how a supervised machine learning model can compare to reinforcement learning models, heuristics, and also the ILP method. In addition, the controlled experiment approach can help interpret the evaluation results of the designed model by playing with different hyperparameter sets and also providing an answer to the scalability aspect in RQ3.2. The time limitations of this study as well as the fact that the identified SOTA papers use a dag task generation script and not real-life dag tasks datasets, makes the use of the case study methology not relevant in this research's context and not able to correctly compare the results with the SOTA papers.

## V. Problem definition

The problem to be tackled is the problem of scheduling a single DAG task with a non-preemptive, global and fixed-priority scheduler (GFPS), on a multicore system. That is, the problem of computing a priority for each node for a DAG task so that it minimizes, using a non-preemptive GFPS, the makespan of the DAG.

More precisely, given a DAG task $\tau = G = (V, E)$ and $n = |V|$, the designed machine learning model idealy needs to compute a list of priorities $p^* \in [\![0, n-1]\!]^n$ so that

$$\mathrm{AFT}(v_{\text{sink}}, p^*) = \min_{p \in [\![0, n-1]\!]^n} \mathrm{AFT}(v_{\text{sink}}, p)$$

with $\mathrm{AFT}(v_{\text{sink}}, p)$ being the actual finish time of the sink node of $\tau$, i.e., the makespan of the DAG task $\tau$, calculated using a non-preemptive GFPS. Note that $p^*$ is not necessarily unique. In this paper, the priorities will be higher when having a lower value, that is, the highest priority given to a node is 0 and the lowest is $n - 1$.

## VI. System design

No supervised learning model has been found in the literature to tackle the non-preemptive, global fixed-priority scheduling problem of a single DAG task. That is because supervised learning is most often used for problems that we, humans, know how to solve, but don't have an algorithm for it, such as image classification or text classification. In this case, there are ways to compute solutions to the problem using linear programming, but it is costly and doesn't scale well when inreasing the number of nodes per DAG. Hence, in the literature, only reinforcement learning has been used to tackle this problem, a machine learning method that doesn't need to know the true solution but can instead approximate the solution by maximizing a reward function. Still, supervised learning is a widely used machine learning method and can provide good interpretations when the model is not too deep. Also,

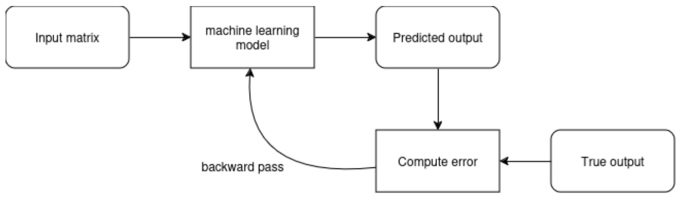


Fig. 4. The supervised learning method.

once trained, the supervised machine learning model can be fast and scalable in terms of computing time, unlike the ILP method. Therefore, the supervised learning model's purpose will be to approximate the computation of a priority-list for a single DAG task that leads to the minimum makespan.

## A. Supervised design

In supervised learning, the model predicts a value (forward pass) that is then compared to the known-to-be-true value using a specific error-function, often called the loss function, and the error is then propagated through the model to tweaks its parameter, minimizing the loss function (Figure 4). The idea is, then, to treat the problem as a classification problem, where each node of a DAG needs to be 'classified' in a specific priority. ILP will be used to compute the optimal[4] priority-lists, and use them as the known-to-be-true values to compare the model-predicted values with.

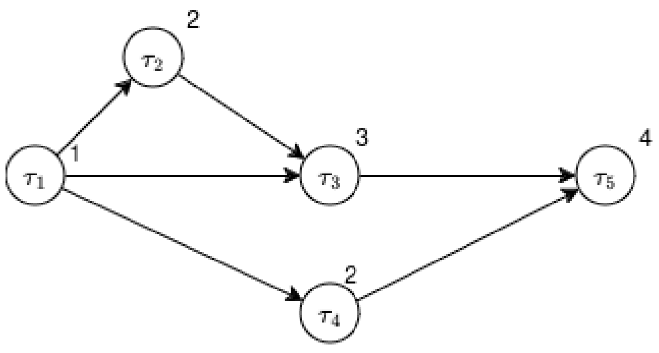
### 1) Input DAGs:

In machine learning, the input that we give to the model needs to be in a matrix representation so that the model can then process it. Therefore, every DAG task needs to have a matrix representation that encompasses information about its internal structure as well as its global attributes. Each input DAG task will, thus, be represented using a matrix of numbers with each row being a node and each column being a raw feature of a node. The list of raw features is similar to what is proposed by Lee et al. [10], that is :

- the normalized wcet of the node, i.e., $C_i/L$ for node $i$ with $L$ being the total workload ;
- the number of incoming neighbours ;
- the number of outgoing neighbours ;
- a boolean value of whether the node is the source or sink node ;
- a boolean value of whether the node is part of the critical path of the DAG.

The wcet is widely used as a criteria for priority-list scheduling algorithm (RM, EDF[36], etc.) but it needs to be normalized to have the context information of the whole graph. The number of incoming and outgoing neighbours makes it possible to take into account the inner structure of the graph to compute the execution order. The source and sink nodes are particular nodes in that their priority is not important because they will respectively execute first and last due to

[4]Here and in the following paragraphs, 'optimal' means 'yielding the minimum makespan'.



$$\begin{array}{c}\tau_1\\\tau_2\\\tau_3\\\tau_4\\\tau_5\end{array}\begin{pmatrix}0.083 & 0 & 3 & 1 & 1\\0.166 & 1 & 1 & 0 & 1\\0.250 & 2 & 1 & 0 & 1\\0.166 & 1 & 1 & 0 & 0\\0.333 & 2 & 0 & 1 & 1\end{pmatrix}$$

Fig. 5. Example dag task with 5 nodes a total workload of 12 and the matrix representation of each node with the column respectively being the above list of features.

their dependency constraints. Finally, the critical path plays a huge role in state-of-the-art heuristics[27][11], with nodes in the critical path often having higher priorities than those that aren't. An example of such a DAG task representation is shown in Figure 5.

### 2) Output priorities:

In order for the supervised model to learn, it needs to compare its predicted output against a known-to-be-true output, also called an output label. In this case, the problem is treated as a classification problem which means that for each node of a DAG, the model needs to output the probability of this node being classified as a specific priority. Therefore, for a DAG task with $n$ nodes, each node must have a list of $n$ probabilities, with each one being the probability of the node being assigned the priority corresponding to the position of the probability in the list of probabilities, starting from 0. For instance, if $n = 3$, a possible output for a node $\tau_i$ would be model$(\tau_i) = [0.2, 0.5, 0.3]$ which means that according to the model, the node $\tau_i$ has a probability to be assigned priority 0 of $0.2$, priority 1 of $0.5$ and priority 2 of $0.3$. As a consequence, the priority chosen for node $\tau_i$ will be the one that has the maximum probability a-posteriori, in the example above, it is priority 1.

This must be the case for every node and as the input is a matrix representing the whole graph, the output then also needs to be a matrix with each row corresponding to each node and the columns correspond to the priorities, e.g., Figure 6.

The model output needs to be compared to the known-to-be-true output or label, which means we need a way of

$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \tau_1 & 0.25 & 0.15 & 0.40 & 0.10 & 0.10 \\ \tau_2 & 0.04 & 0.06 & 0.01 & 0.09 & 0.80 \\ \tau_3 & 0.50 & 0.35 & 0.08 & 0.02 & 0.05 \\ \tau_4 & 0.10 & 0.23 & 0.07 & 0.55 & 0.05 \\ \tau_5 & 0.30 & 0.52 & 0.08 & 0.04 & 0.06 \end{array} \Longrightarrow \begin{array}{c} \tau_1 \tau_2\ \tau_3\ \tau_4 \tau_5 \\ [2, 4, 0, 3, 1] \end{array}$$

Fig. 6. Example of an matrix output from the model, the predicted list of priorities (on the right) is retrieved from the probabilities.

calculating the optimal priority list. The idea is to use an ILP solver to compute the optimal schedule of a dag task, and then use this optimal schedule to retrieve the optimal priorities for each node, according to their start time in the computed schedule. Each node will be sorted according to their start time in increasing order and the priority value will then be the position of each node in the sorted list. As a consequence, the nodes with the lowest start time will have the highest priority.

The makespan calculation is done using a non-preemptive GFPS, with a first fit core assigning strategy (see Section VI-D). Hence, the retrieved optimal priority list can sometimes lead to a different schedule than what the ILP solver computed. Fortunately, because we're on a homogeneous system, as long as the priorities are respected, the core assignment doesn't matter and would just lead to having a symmetric schedule compared to the ILP schedule, that is swapping cores in the ILP schedule, which doesn't change the makespan.

Therefore, an Integer Linear Programming (ILP) solver will be used to compute the optimal schedule for each DAG task (see Section VI-C) and then ordering the nodes according to their start time in the ILP schedule, thus having an optimal list of priorities for the nodes that can then be compared to the predicted model output list.

#### 3) *Loss function:*

Although the problem is being treated as a classification problem, the goal still is to compute a priority list that minimizes the makespan of the DAG. Therefore, the idea was, initially, to compare the makespan produced by the predicted priority-list with the makespan produced by the ILP schedule, using a simple difference operator. Unfortunately, the non-preemptive GFPS which is used to compute the makespan, isn't differentiable for the priority-list given to it, which lead to segfaults in the training phase. Thus, instead of looking at the makespans, the idea is to look at the priority matrix and compare the one predicted by the model to the ILP priority matrix that is retrieved from the priority-list. Indeed, an ILP output matrix to compare the predicted output to can be calculated using the previously retrieved priority-list. Like the model-predicted output matrix, the ILP matrix output will represent the probabilities of each node being assigned a specific priority, using the same process described in the previous section but in reverse, with a probability of 1 on the optimal priority value, and 0 otherwise. For instance, if the optimal priority list for a DAG task of 3 nodes is [$\tau_1$: 0, $\tau_2$: 2, $\tau_3$: 1], then the true label is the matrix :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

With each row representing $\tau_1$, $\tau_2$ and $\tau_3$ respectively, and the column represent the priorities 0, 1 and 2 respectively.

Once we have the two matrices, it is then possible to use the binary cross-entropy loss function, widely used for classification problem, to compare the predicted output to the optimal ILP matrix.

More specifically, the binary cross-entropy loss function is defined as :

$$loss(x, y) = \sum_{i=1}^{n} -y_i \log(x_i) \qquad (2)$$

where $x$ is the flattened matrix representing the predicted output, $y$ is the flattened matrix representing the true output (ILP output), and $n$ is the number of element in $x$ and $y$.

## B. *Model design*

The main idea for designing the architecture of the model is to leverage the structural information of the DAG. That is, it needs to learn the relationships and connections between the nodes to correctly classify the nodes into the right priorities. In machine learning a prevalent architecture when dealing with graphs is the graph neural network architecture. The idea is, instead of treating each node as independent vectors, like in a traditional feed-forward network, a graph neural network gathers information of each node and its neighbours and passes it to the next layer. One way to do that is to aggregate the neighbours' information of a node and use this aggregation to compute the next representation vector of the node.

Unfortunately, this kind of aggregation also means that a node will have similar attributes to its neighbours, when passing through multiple layers of a graph neural network (GNN). This problem can lead to groups of neighbours having very similar vector representation and can cause an over-smoothing problem[37]. The two main ways to fix this kind of problem is either to reduce the amount of GNN layers or/and use the attention-mechanism to give a notion of importance to a node's neighours, thus reducing impact a less important neighbour can have on the aggregation phase. Of course, reducing the number of GNN layers also means less complexity which can decrease the performance of the model, thus, a balance between having complexity and limiting the over-smoothing problem is often done to maximize performance. This is exactly what they do in the literature[10][9]. More specifically, Lee et al. [10] designed a Graph Convolutional Network (GCN) based on the attention-mechanism to encode the nodes' feature vectors in their reinforcement learning model, performing slightly better than the state-of-the-Art heuristics from Zhao et al. [11].

Therefore, the model's architecture will be based on the proposed encoder in Lee et al. [10], and it will be comprised of three modules : two feed forward networks and one attention-based graph convolutional network (GCN) (see Figure 7).

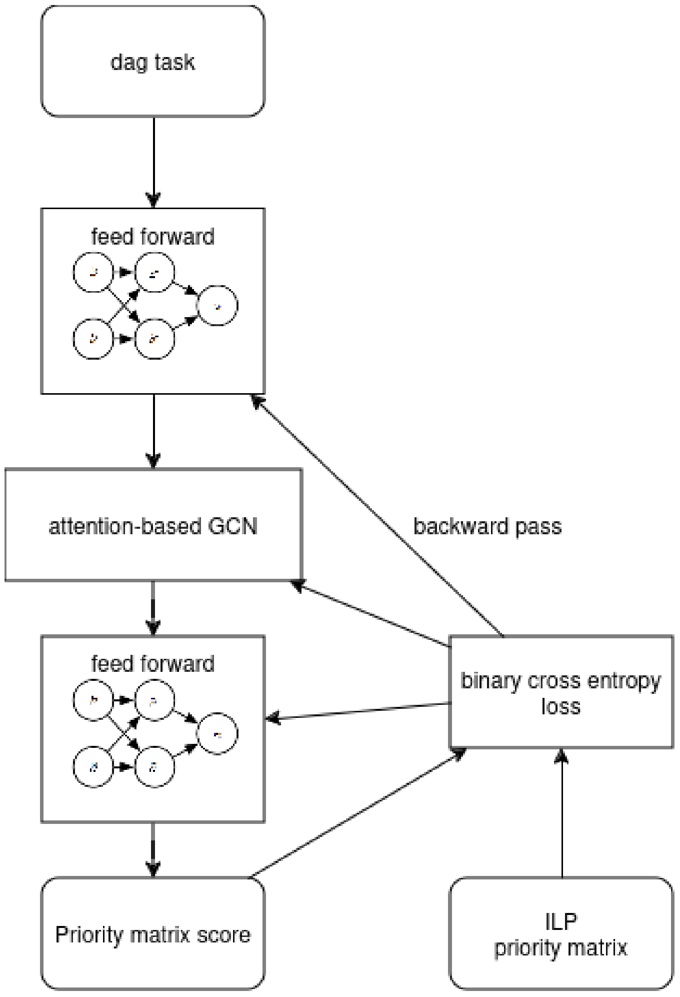


Fig. 7. The architecture diagram of the proposed supervised machine learning model.

*1) Feed forward networks:* Using a feed-forward network (FFN) before the GCN can help at pre-encoding the feature vectors of the nodes so that the GCN can better work on those feature vectors. Also, a feed-forward network (FFN) at the end of the model is needed to make the final classification decision about each node, once the feature vectors have all the graph structural information encoded into them. Of course, the more layers in the FFN, the more complexity is encoded in the feature vectors, but it can also potentially lead to an over-fitting problem with the model being too complex and taking the noise into account, decreasing its generalization power and its overall performance.

Therefore, the two feed forward networks will have 3 layers with each layer $n$ producing the following output,

$$o_n = ReLU(W_n X_n + b_n),\ n \in \{1, 2, 3\} \tag{3}$$

where $W_n \in \mathbb{R}^{5\times 5}$ is the matrix of trainable weights at layer $n$, $b_n \in \mathbb{R}^5$ is the bias and $X_n$ is either a node's vector representation or the output of the previous layer, and $ReLU(x) = max(0, x)$. There is an exception for the second feed forward network where the last layer's output is computed using the following equation:

$$o_3 = \tanh(W_3 X_3 + b_3) \tag{4}$$

where $W_3 \in \mathbb{R}^{nbPriorities\times 5}$, $b_3 \in \mathbb{R}^{nbPriorities}$ and $\tanh$ is the hyperbolic tangent activation function. Also, $nbPriorities$ corresponds to the number of different priorities that can be assigned to each node, which in this case is going to be the same as the number of nodes in the DAG.

*2) Graph Convolutional Network:*

The GCN layer needs to aggregate the information of the neighbours of a node and use that information to compute the next representation vector of the node. To do that, we utilize both the incoming and outgoing neighbours of a node and compute their attention scores to then aggregate that information to pass it through an activation function, thus updating the representation vector of the node (see Figure 8) Considering both in and out neighbours enables the encoding of more of the structural complexity of the graph, and also using the directed aspect of the DAGs. The Figure 8 represents one GCN layer.

To be more specific, for each GCN layer, the input vector $X_k$ goes through an aggregation phase where, given the set of incoming neighbours of node $v_i$, $\mathcal{N}_{in}(v_i)$, which includes $v_i$, and the set of outgoing neighbours $\mathcal{N}_{out}(v_i)$, which also includes $v_i$, the next vector representation $X_{k+1}$ is calculated by the following equation:

$$GCN(X) = AttentionModule(X, \mathcal{N}_{in}(v_i), \mathcal{N}_{out}(v_i)) \tag{5}$$

where $AttentionModule$ is defined by the Equations 6–9 and computes the importance scores of each neighbour, in each set of neighbours:

$$\begin{aligned} &AttentionModule(X, \mathcal{N}_{in}(v_i), \mathcal{N}_{out}(v_i)) = \\ &ELU\left(W\left(Att(X, \mathcal{N}_{in}(v_i)) \oplus Att(X, \mathcal{N}_{out}(v_i))\right) + b\right) \end{aligned} \tag{6}$$

$$Att(X, \mathcal{N}(v_i)) = W \sum_{j\in\mathcal{N}(v_i)} \alpha_{ij} X_j \tag{7}$$

Where

$$\alpha_{ij} = \frac{\exp\left(a^T W X_i + b^T W X_j\right)}{\sum_{k\in\mathcal{N}(v_i)} \exp\left(a^T W X_i + b^T W X_k\right)} \tag{8}$$

are the attention coefficients which evaluates how important is $v_j$ to $v_i$'s vector representation, similar to what is done in Lee et al. [10]. Equation 7 computes the attention scores for each neighbours of node $v_i$, scores that are then used in Equation 6 to aggregate the scores information and pass it through the ELU activation function. Every $W$, $a$, and $b$ are different matrices of trainable parameters, and ELU is the exponential linear unit activation function (Equation 9).

$$\text{ELU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \exp(x) - 1, & \text{if } x \le 0 \end{cases} \tag{9}$$

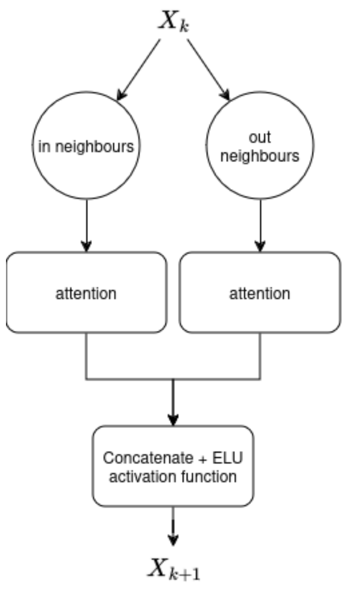


Fig. 8. Diagram of the graph convolution network layer. $X_k$ is the transformed node vector representation and ELU is the exponential linear unit function (see Equation 9).

The GCN layer can be repeated several times to improve the model's capacity to better capture the complexity of the graph structure. In this case, the number of GCN layers was set to 3.

## C. Computing the labels

*1) Computing the optimal schedule:* ILP will be used to compute the optimal schedule, hence the need to have an ILP solver. Fortunately, Yip and Kuo [38] proposed an ILP-based scheduling solver for event-chains of time-triggered tasks using the Logical Execution Time paradigm[39]. Their work aimed at minimizing the sum of dependency delays between the job instances of each task and they have recently added support for multicore architectures[5].

Although their minimizing problem is slightly different from ours, we can convert our minimizing problem and then use their LET solver by converting each DAG task to an event-chain of LET task. To do this, each node needs to be converted to an LET task which implies adding an initial offset, activation offset, LET interval duration and a period to every node. Hence, for each node, the initial offset and activation offset will be set to 0 and the LET interval duration will be set to the node's wcet. For the period, every node in a DAG will have the same period which will be equal to the total workload squared. The wcet of a node will be the wcet of its LET task version. Lastly, each path in the DAG will represent an even-chain of tasks.

Unfortunately, minimizing the sum of dependency delays (i.e., Equation 6 in Yip and Kuo [38]) is not equivalent to minimizing the makespan of a DAG (see Figure 9) but we can

[5]https://github.com/mkuo005/LET-LP-Scheduler

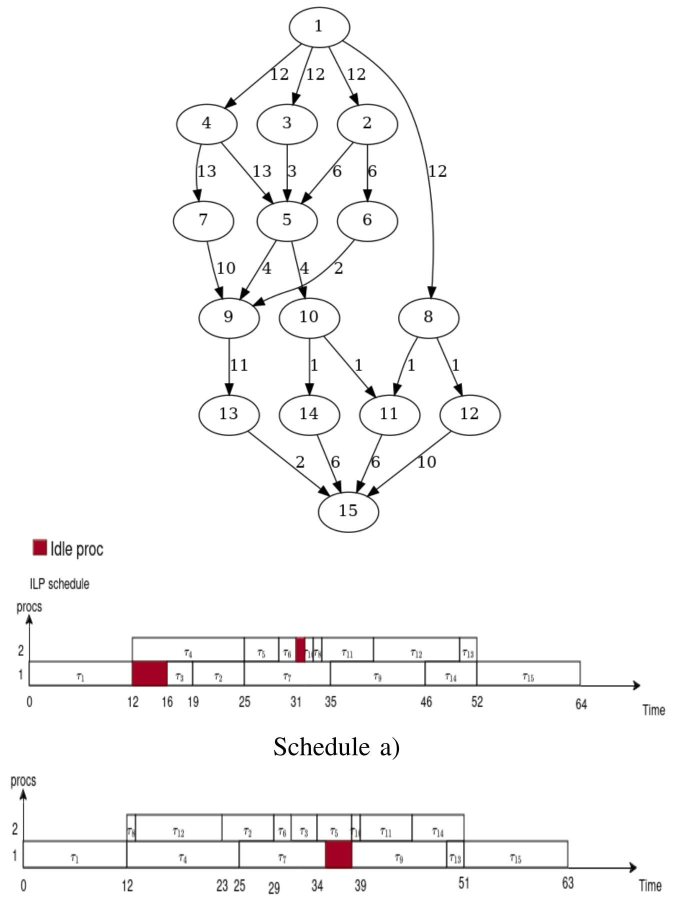


Fig. 9. Example of a DAG task (top image) where the schedule that the ILP solver outputs (the schedule a)) produces a longer makespan (64) but smaller sum of dependencies delays (400) than the schedule b) with a makespan of 63 and a sum of dependencies delays of 427.

change the objective function to the end execution time of the sink node, i.e., the node which doesn't have any successors, which is the definition of the makespan. This effectively makes the solver minimize the makespan of DAG task which is precisely the problem at stakes[6].

*2) Computing the optimal priority-list:*

Once the optimal schedule (i.e., the schedule yielding the minimum makespan) is computed, the optimal priority-list can be obtained from the schedule by looking at the start times of each node and increasing the priority according to the node's start time. The sooner the start time, the higher the priority.

For instance, for the non-optimal schedule shown in Figure 9 (schedule a)), the corresponding priority-list would be 0 for node $\tau_1$, 1 for node $\tau_4$, etc, i.e., [$\tau_1$: 0, $\tau_2$: 3, $\tau_3$: 2, $\tau_4$: 1, $\tau_5$: 4, $\tau_6$: 6, $\tau_7$: 5, $\tau_8$: 8, $\tau_9$: 10, $\tau_1$0: 7, $\tau_1$1: 9, $\tau_1$2: 11, $\tau_1$3: 13, $\tau_1$4: 12, $\tau_1$5: 14]. This priority list is then converted to a matrix by the process described at the end of Section VI-A2.

[6]See the github repository for more details https://github.com/FelicienFG/research-project-AUT/.

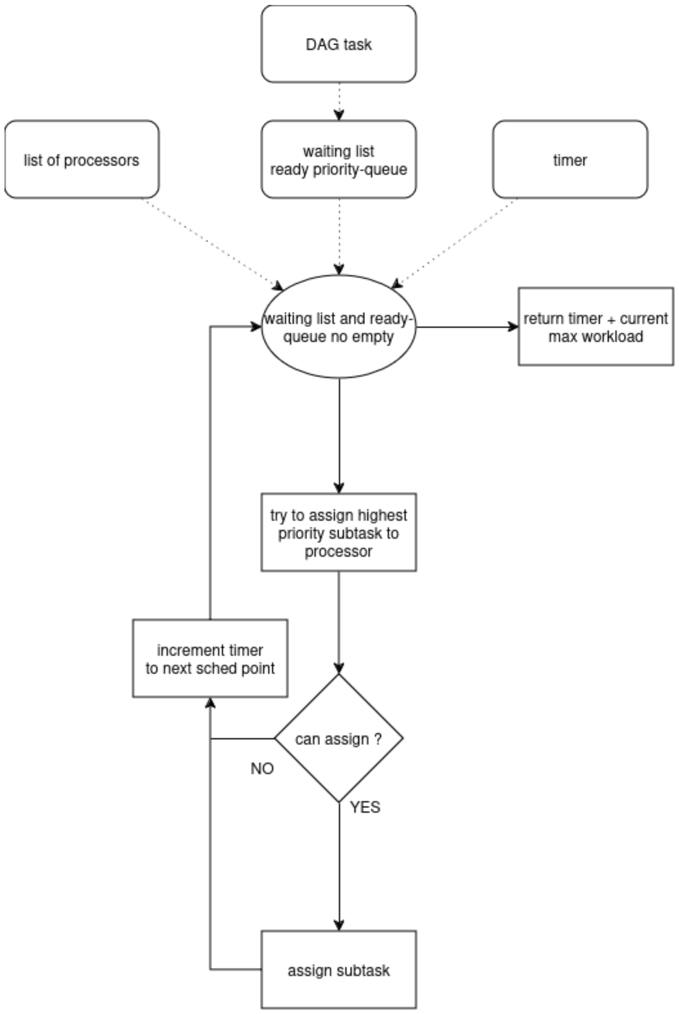

DAG task
list of processors
waiting list
ready priority-queue
timer
waiting list and ready-queue no empty
return timer + current max workload
try to assign highest priority subtask to processor
increment timer to next sched point
can assign ?
NO
YES
assign subtask

Fig. 10. Top view of the makespan computation algorithm.

*D. Makespan calculation*

The main performance metric that will be used in the evaluation is the makespan, i.e., the end execution time of the sink node of a DAG. The exact computation of the makespan is done by simulating a simple non-preemptive global fixed-priority scheduler, for which the algorithm is described in the Figure 10. The basic idea is to have a ready queue that contains the nodes that have their precedence constraints met and the nodes still waiting to have their dependencies met are in the waiting list. Each loop iteration, the waiting list and ready queue are updated and the first idle processor gets the node at the top of the ready priority queue (see Figure 10 and 11).

The implementation will be done in C++ with bindings to python to make the makespan computation faster.

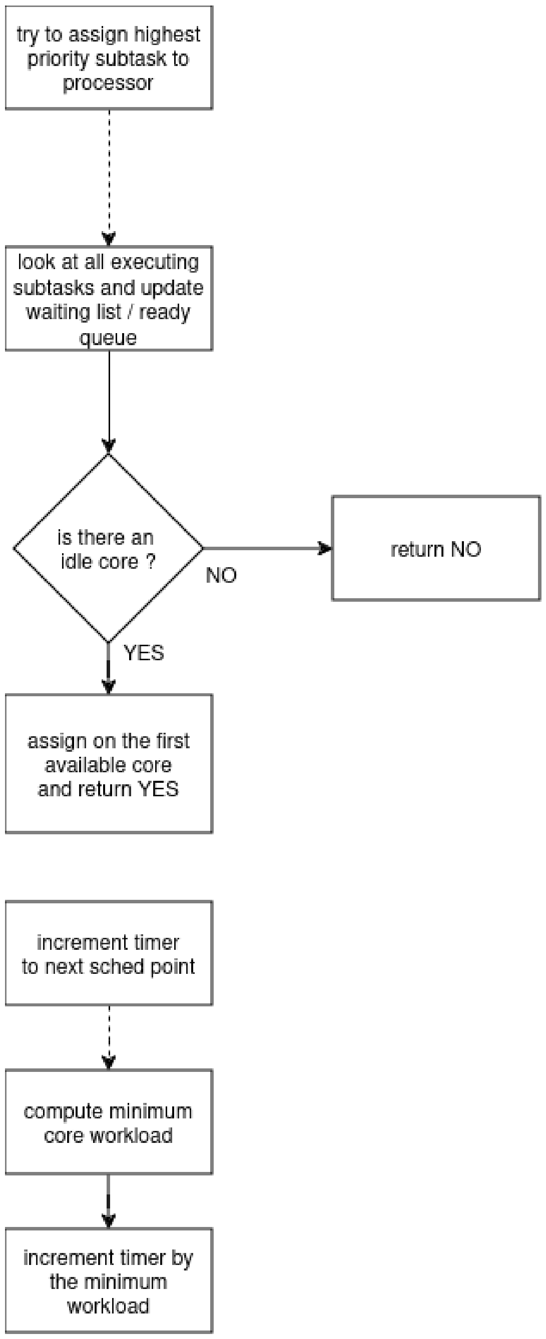

try to assign highest priority subtask to processor
look at all executing subtasks and update waiting list / ready queue
is there an idle core ?
NO
return NO
YES
assign on the first available core and return YES
increment timer to next sched point
compute minimum core workload
increment timer by the minimum workload

Fig. 11. Algorithms for assigning the highest priority node (or subtask) to a processor (top) and for incrementing the timer to the next scheduling point (bottom).

## VII. Evaluation

*A. Environment setting*

The evaluation of the model is focused on two metrics : the makespan and the computing time once trained. For the implementation of the model, python 3.10.12 with the PyTorch 2.4.1 library is used and runs on a google collab runtime with 90 TPU cores. The stochastic gradient descent is used for training with a learning rate of 0.001 and a batch size of 250.

During the initial evaluation, an over-smoothing problem[37] occured which lead to the model smoothing out the priorities and assigning the same priority to all the nodes. To try to solve this, reducing the number of GCN layers to 1 was done, which unfortunately didn't fix the problem, leading to the addition of a regularization term to the loss function. The idea is to tackle the over-smoothing probblem by forcing the model to differentiate between the nodes' representation. Hence, the regularization term is the squared inverse of the average euclidian distance between each of the nodes' latent representation, i.e.,

$$\text{regTerm}(X) = \frac{1}{\sum_i \sum_{j>i} \|X_i - X_j\|_2} \tag{10}$$

and the loss function then becomes

$$loss(X, y) = \sum_i -y_i \log(x_i) + \text{regTerm}(X) \tag{11}$$

with $x_i$ being the elements in $x$, the flattened matrix representation of $X$ and $y$ the true output (see Section VI-A3).

Although those modifications didn't seem to improve on the initial results, the evaluation of the model has been done with these modifications to establish why the over-smoothing problem was happening(see Section VII-E).

## B. Dataset generation

To generate the DAG tasks, the generator from Zhao et al. [11] is used, which is also used by Lee et al. [10] and [25], to generate random DAGs. Apart from it being used by state-of-the-Art papers, which my results will be compared to, and providing a qualitative and diverse dataset, this generator has great modularity and the open-source git repository[7] makes it easy-to-use. The random DAGs are generated using the following process : The generator starts at a source node and expands outward, creating nodes in successive layers. The total number of layers, or maximum depth, is randomly determined to be between two values $a$ and $b$. For each layer, the number of nodes generated is chosen uniformly, ranging from 2 up to the parallelism parameter, $p$ which in this case, is fixed at $p = 8$. Nodes that do not already have connections can randomly connect to other nodes in the previous layer with a probability of $p_c = 0.5$. After all layers are generated, any terminal nodes are linked to a final sink node. Both the source and sink nodes are used to structure the graph and have a fixed execution time of one unit each. Lastly, execution times are assigned randomly to all nodes while ensuring that the total workload sums up to $W = 1000$[11], that is, the fully sequential execution time of each DAG task is set to 1000 time units (i.e., the sum of each node's wcet).

[7]https://github.com/automaticdai/dag-gen-rnd

| n | a | b |
|---|---|---|
| 10 | 3 | 8 |
| {20, 30} | 5 | 8 |
| 40 | 7 | 10 |
| 50 | 10 | 15 |

TABLE II
DAG GENERATOR $a$, MINIMUM NUMBER OF LAYERS, AND $b$, MAXIMUM NUMBER OF LAYERS, PARAMETER VALUES FOR GENERATING RANDOM DAGS ACCORDING TO NUMBER OF FIXED NODES PER GRAPH WE NEED TO RETRIEVE AFTERWARDS.

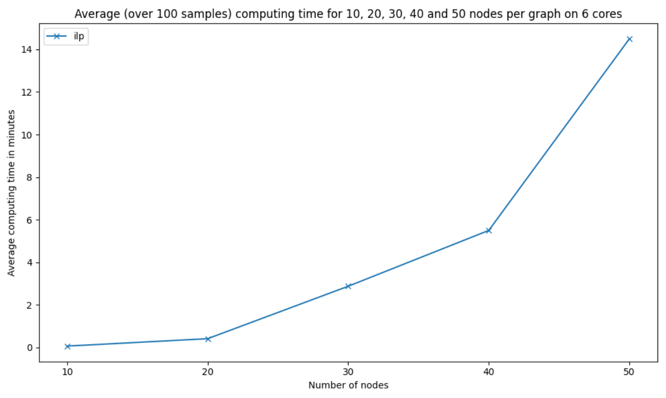


Fig. 12. Average computing time for computing the optimal schedule of a single DAG task using the ILP method, in minutes, according to the number of nodes in the DAG, on a system with 6 cores.

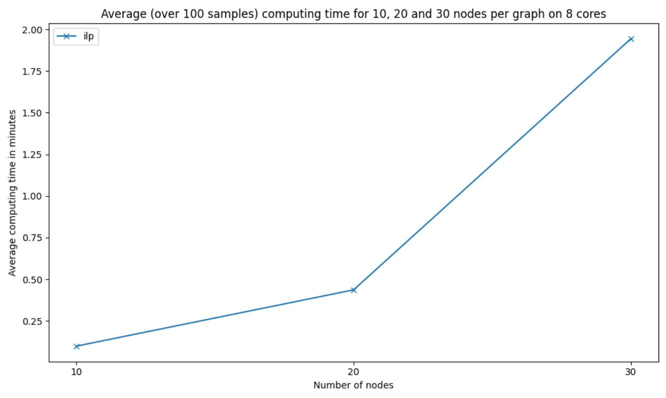
To generate DAGs with a fixed number of nodes $n$, the generator is first used to generate 50000 DAG tasks with different values for $a$ and $b$ depdending on what the value of $n$ is. Then, the generated DAGs with the specified number of nodes are retrieved from the dataset. Specifically, Table II shows the different $a$ and $b$ values according to $n$.

Using those values, 1400 DAG tasks were retrieved and used for evaluation, for each value of $n$. 1000 of them were used for training the model, 400 for testing and 100 of them were used to measure the computing time for both the ILP and the supervised ML methods.

## C. Computing time results

The computing time for the ILP method is shown in Figure 12. The ILP solver was timed out whenever the computing time exceeded 1 hour and it did time out when considering systems of 2 and 4 cores, when the number of nodes exceeded 20 nodes (i.e., 30, 40 and 50 nodes) per DAG task. Hence, in the next sections, only 6 and 8 cores will be considered, with the number of nodes not exceeding 30 nodes, to have enough ILP solved DAGs for training the model.

As you can see from Figure 12, the more nodes there are on the system, the more time it takes for the ILP solver to compute the optimal schedule, and it does so in, what it seems to be, an exponential growth. Figure 12 thus shows the non-



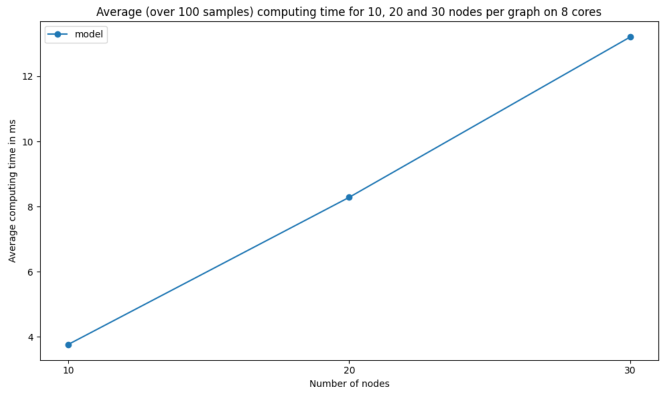


Fig. 13. Average computing time for computing the optimal schedule of a single DAG task using the ILP method (top), in minutes, and using the ML model (bottom) according to the number of nodes in the DAG, on a system with 8 cores.

scalability of the ILP method, with the number of nodes per DAG increasing.

When comparing to the ML model's computing time (Figure 13), you can see how, unlike ILP, the model seems to be a lot more scalable than ILP, as shown by the linearity of the computing time curve (bottom graph of Figure 13). The linearity is of course not a true linearity as the theoretical complexity, in terms of operations on the nodes' vector representation, of the model can be calculated to be $O(n^2)$ according to Section VI-B, but the curve is linear because of how short the X axis is, the polynomial curve is zoomed in which explains why the curve seems linear.

This difference in scalability is expected as the problem is NP-hard, which means that the ILP solution, calculating optimal solutions, has to be at least of exponential complexity, hence the non-scalability of the ILP method. The model's scalability is also expected as the theoritacl complexity is polynomial.

## D. Performance comparison

Two performance metrics will be used for the model. The makespan which will be compared to the ILP solution, the heuristic from Zhao et al. [11] and a random method where

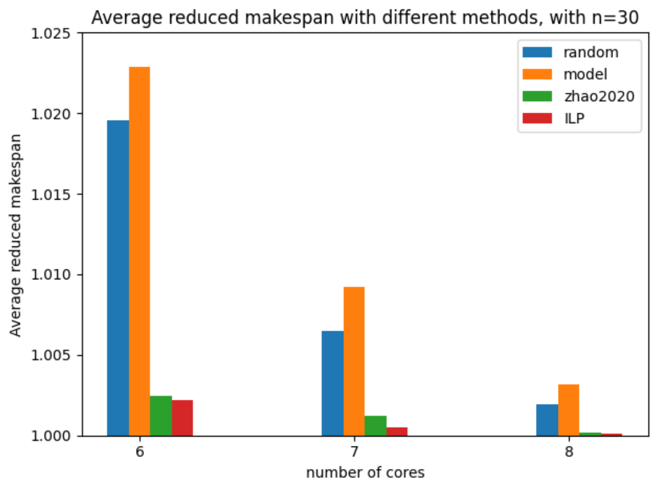


Fig. 14. Average reduced makespan (makespan divided by the critical path length) on the test set (400 DAGs) using the four different methods on 6, 7 and 8 cores with 30 nodes per DAG, with 'random' being the method of randomly assigning priorities to nodes of a DAG.

each priority is assigned at random, and the accuracy / loss of the model.

Figure 14 shows the average reduced makespan, which is the makespan divided by the length of the critical path, computed with each method on 400 DAGs from the test set. Therefore, the closer to 1 it is, the better it performs.

Not only does the model perform worse than the heuristic from Zhao et al. [11], the model performs generally worse than the random method. Also, the heuristic is very close to matching the ILP method, performing even greater than expected.

When not-trained (Figure 15), the model performs about the same or worse than randomly. This is because although the weights are initialized randomly, the similarity between the nodes' vector representation makes the model, even when untrained, compute the same priority for different nodes, which is exacerbated with training, leading to the over-smoothing issue (see Table III).

In terms of accuracy and loss, the results are shown in Figure 16. The accuracy is, in this case, how well the resulting priority list from the model matches the ILP priority list. Unfortunately, the accuracy is stagnant at less than 10% and doesn't get better when increasing the number of epochs to train on. Also, the accuracy doesn't change between the training and testing phase which further confirms that the model doesn't learn at all.

Furthermore, when looking at the output of the model (Table III), we can see that in most cases, the priority lists predicted by the model are, on average, more than 90% made out of the same value, i.e., for 10 nodes : [1, 1, 1, 1, 1, 1, 1, 1, 2, 1] for instance. This shows that the over-smoothing of the DAGs is still an issue for the model and explains the low accuracy.

Although the accuracy results shown here are only in the case of 8 cores and 30 nodes per DAG, the other cases, i.e.,

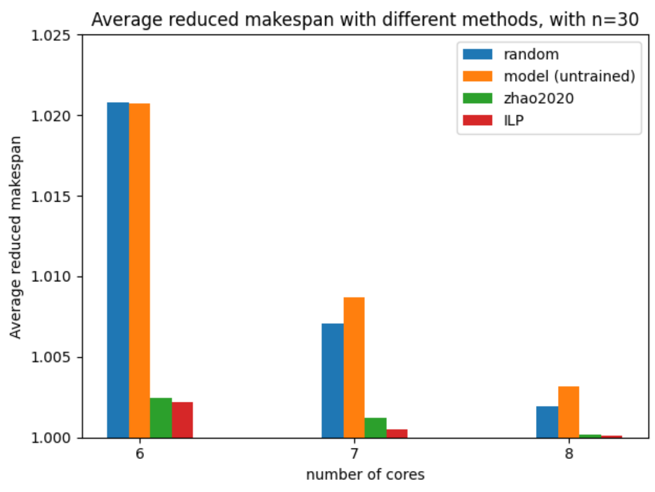


Fig. 15. Average reduced makespan (makespan divided by the critical path length) on the test set (400 DAGs) using the four different methods on 6, 7 and 8 cores with 30 nodes per DAG, with 'random' being the method of randomly assigning priorities to nodes of a DAG, and the model not being trained.

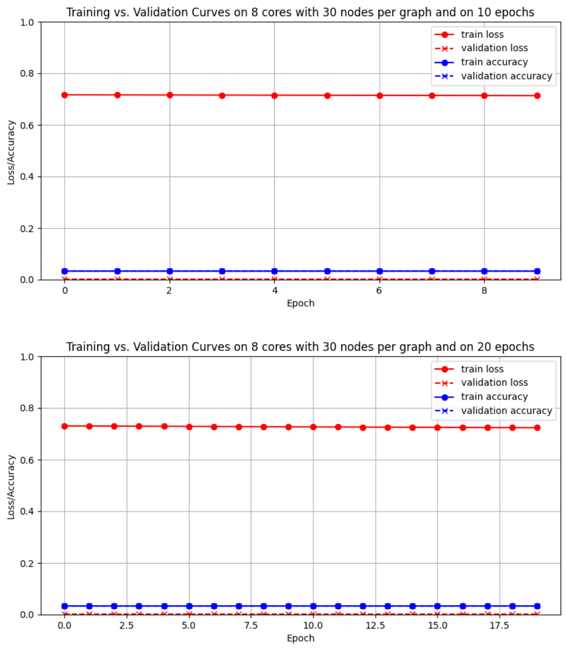


Fig. 16. Accuracy (in blue) and loss (in red) values for training (circles) and testing (crosses), accross the epochs. The top graph shows results for 10 epochs and the bottom one for 20 epochs. The results are shown on 8 cores with 30 nodes per DAGs, with a training batch size of 250 and a learning rate of 0.001.

| **number of nodes per DAG/number of cores** | **6** | **7** | **8** |
|---|---|---|---|
| **10** | 94% | 99% | 94% |
| **20** | 91% | 60% | 90% |
| **30** | 100% | 99% | 98% |

TABLE III
AVERAGE PERCENTAGE OF THE NUMBER OF PRIORITY SLOTS THAT HAVE THE SAME VALUE IN A PRIORITY-LIST OUTPUTED BY THE MODEL.

for 6, 7 cores and with 10,20 and 30 nodes per DAG, display very similar results and lead to the same conclusions as those shown here.

## *E. Discussions*

There are several reasons why a machine learning model sub-performs. The issue can be because of an overfitting problem, meaning that the model is too complex and learns the noise / bias that comes with the dataset. It can also be the opposite, under-fitting, which is due to a model that is too simple for the task at stake, lacking the architecture to capture the complex information and relationships in the data. It can be because of the nature of the dataset that is too biased or too noisy which makes it impossible for the model to learn from it. Finally, it can be because the model didn't train on enough data or on enough epochs, which would mean that the model didn't have the time to converge and need more data samples to do so.

Let's tackle these different problems and see why the problem is the fact that supervised-learning isn't fit for this kind of problem, that is the single DAG task scheduling problem.

### *1) Biased data and amount of data:*

Data is always bias and the amount of dataset bias[40] can have a huge impact on the performance of a model. Although the number of training samples (1000) is not a lot compared to what Lee et al. [10] have generated (8000), the dataset generation procedure is the same as the one in both Zhao et al. [11] and Lee et al. [10]. The latter specifically, got more than promising results with their machine learning model, outperforming the heuristic in Zhao et al. [11] by up to 3% in terms of makespan, when using the same data generation procedure as here. Furthermore, the loss is extremely stable and doesn't go down during training, even when training on 20 epochs (Figure 16). This means that even if there is bias in the data, the fact that others have used the same data generation procedure and got great results shows that the dataset bias cannot explain the low performance of the model. Also, the fact that the model doesn't seem to learn at all with its loss being stagnant accross the epochs, removes the idea of the model performing badly because there is not enough data to learn on.

### *2) Over-fitting or under-fitting:*

The reason why the model is performing this badly is probably not over-fitting. This is because over-fitting is characterized by the model learning the noise in the training set and performing[41]. Hence, if it were the case, there would be a noticable gap between the training and validation accuracy in Figure 16, which there isn't. Also, the over-smoothing problem, which is causing the model to sub-perform, doesn't come from the noise in the data but rather from the architecture of the model itself[37]. Hence the problem might rather be that the model is under-fit for this problem.

Indeed, as mentioned before, what the model is learning is to assign the same priority at every node, which also means that it doesn't learn to approximate the output given by the ILP priority-lists, as shown by the accuracy and loss values not notably decreasing during training (Figure 16). The issue here is that the model resembles very closely the encoder in Lee et al. [10] which didn't have the over-smoothing problem. The attention-mechanism used is also close to what is done in Zhao et al. [9] which also didn't have an over-smoothing problem.

The fact that The over-smoothing is very strong (see Table III) and that the model is close to what has been done in the reinforcement learning models[10][9], lead us to conclude that the problem might be the learning method, that is the supervised learning method.

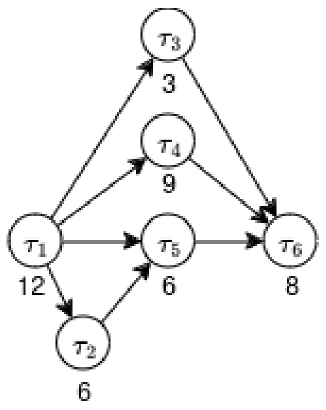
*3) the supervised learning problem:*

In supervised learning, the problem is either going to be a regression problem, when dealing with real numbers as the target variable such as predicting the price of a product or the rate of infection in an epidemic, etc. Or the problem is going to be interpreted as a classification problem. In our case, the numbers are discrete and the set of priorities is finite which is why the problem was treated as a classification problem. However, usually, for supervised learning, the predicted output aim at mathing a ground truth, known beforehand, that is unique. Although the ILP method permitted us to have a priority list that effectively yields the minimum makespan, that priority list is not unique.

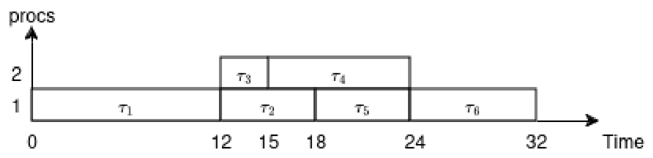
Indeed, not only is the minimum makespan schedule not necessarily unique (see Figure 17), which means that the optimal priority-list is not unique (in Figure 17, schedule a) gives [0, 1, 2, 3, 4, 5] but schedule b) gives [0, 1, 2, 4, 3, 5]). But even if the schedule might be unique, when the start time of nodes are the same, multiple priorities can be assigned to those nodes. For instance, in schedule a) of Figure 17, the two priority-lists [0, 1, 2, 3, 4, 5] and [0, 2, 1, 3, 4, 5] are valid for minimizing the makespan of the DAG.

This complete lack of uniqueness in the true labels renders the ILP priority-lists almost useless for the model to learn on, because those lists don't show the difference between a node having a specific priority because it lengthen the makespan otherwise (e.g., priority of $v_8$ in the last two schedule scenarios in Figure 2), and a node having a specific priority when it could have other potential priority values and it wouldn't change the makespan (e.g., the example in Figure 17). Thus, the model can't learn to assign different priorities to the nodes but rather to smooth out the priorities and give the same one to most of the nodes of a DAG, hence the over-smoothing problem. Furthermore, the non-uniqueness in the nodes' vector representations also contributes to the model struggling to learn. Indeed, if two nodes have the same number of in and out neighbours, are neither a sink nor a source node, their respective vector representation will be very similar. For instance, in the DAG shown in Figure 18, the vector representation of nodes $\tau_2$ and $\tau_3$ are the same, i.e., $(0.067, 1, 1, 0, 0)$, and although those nodes having different priorities doesn't impact the final makespan, they do have different priority values with the ILP priority-list, which contributes to the over-smoothing problem.

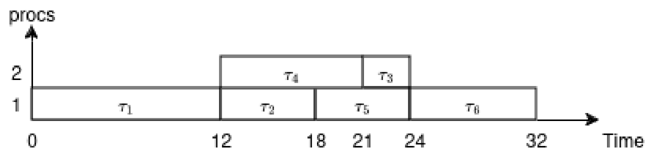
One potential solution to explore might be to generate , for each DAG, all the different schedules that lead to the minimum makespan and from them, retrieve a single, unique priority list not only containing the information about which nodes need to be executed first, but also having the information of which nodes have interchangeable execution



a)



b)



c)

Fig. 17. Example of DAG task (a) for which two schedules (b and c) leading to different priority lists are shown, both schedules giving the minimum makespan.

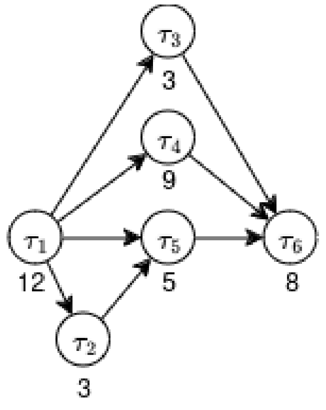


Fig. 18. Example of a DAG task where two different nodes have the exact same vector representation.

ordering (e.g., the two nodes in the example above, Figure 18).

The evaluation results show that the supervised learning model doesn't learn correctly, leading to an over-smoothing and under-fitting problem which makes the model perform very poorly (accuracy $< 10\%$) and having an even worse makespan performance than random priority assignment. Analysis of the results showed how the model performance problem is due to the learning method and supervised design being unfit for the single DAG task fixed-priority scheduling problem. Thus, relating to RQ3, although the reinforcement learning method has shown great performance compared to ILP[9] and state-of-the-Art heuristics[10], the supervised learning model presented here clearly cannot compare at all to either SOTA heuristics or ILP, even though it is more scalable than ILP, mostly because of the supervised learning method being unsuited for DAG task fixed-priority scheduling.

## VIII. Limitations

Although the DAG generator used is indeed used in the literature[10][11][25], the generator is not made for generating a number of DAG tasks with a fixed number of nodes. Hence, using the generator as it has been used might not lead to a dataset as diverse as what is done by Zhao et al. [9] for instance. Also, no evaluation of the model using multiple values of the parallelism parameter $p$ has been done, which could've been interesting to look at to see the impact of this parameter, and of how parallelizable a DAG task is, on the performance. The amount of DAG tasks used as a test set is 400 which is lower than what the other papers used, 600 in Zhao et al. [9] and 1000 in Lee et al. [10], which makes randomness have a bigger influence than in those papers.

For the evaluation, it has only been done using a train set and test set, no cross-validation such as K-fold has been done which could have enhanced the interpretation of the results, reducing the impact of dataset bias in the performance results.

Furthermore, the supervised learning method might still be an option for this kind of problem, as the main problem isn't the supervised learning method per se, but rather the computation of the output labels. Also, although the ILP method computes optimal schedules, it does take a long time to do so and is not scalable, making the model training not scalable which can if we want to train the model on DAGs with 100, 500 or 1000 nodes.

## IX. Conclusions and Future Works

In this paper, the DAG task scheduling on multicore systems problem has been addressed, especially looking at the machine learning techniques used to tackle this problem. It has been found that very few papers use machine learning for this kind of problem and no paper has been found using supervised learning for the single DAG multicore scheduling problem. Hence, a supervised learning model has been designed and evaluated. Unfortunately, the performance results showed how the supervised learning method seems unfit for this kind of problem, with the main problem being the computation of the output labels, and reinforcement learning appears to yield much better results, comparable to state-of-the-Art heuristics and the optimal Integer Linear Programming method.

Future works should be done on computing unique and more informative output labels, to expand on the results discussion, and, if successful, expand the model to treat multi-DAGs scheduling as well. One could also work on using reinforcement learning for multi-rate DAGs, or chain of time-triggered tasks, and potentially extend the task model to include other types of tasks, such as Logical Execution Time tasks.

## Acknowledgement

*For referencing in LaTeX, check out: https://texblog.org/2014/04/22/using-google-scholar-to-download-bibtex-citations/*

## References

[1] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[2] H. Kopetz, "The time-triggered model of computation," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE, 1998, pp. 168–177.

[3] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–38, 2019.

[4] D. Abbott, "Chapter 17 - linux and real-time," in *Linux for Embedded and Real-Time Applications (Fourth Edition)*, fourth edition ed., D. Abbott, Ed. Newnes, 2018, pp. 257–270.

[5] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[6] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *2012 IEEE 33rd Real-Time Systems Symposium*, 2012, pp. 63–72.

[7] J. Du and J. Y.-T. Leung, "Complexity of scheduling parallel task systems," *SIAM Journal on Discrete Mathematics*, vol. 2, no. 4, pp. 473–487, 1989.

[8] J. Ullman, "Np-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[9] M. Zhao, L. Mo, J. Liu, J. Han, and D. Niu, "Gat-based deep reinforcement learning algorithm for real-time task scheduling on multicore platform," in *2024 36th Chinese Control and Decision Conference (CCDC)*, 2024, pp. 5674–5679.

[10] H. Lee, S. Cho, Y. Jang, J. Lee, and H. Woo, "A global dag task scheduler using deep reinforcement learning and

graph convolution network," *IEEE Access*, vol. 9, pp. 158 548–158 561, 2021.

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

[12] R. V. Book, "Richard m. karp. reducibility among combinatorial problems. complexity of computer computations, proceedings of a symposium on the complexity of computer computations, held march 20-22, 1972, at the ibm thomas j. watson center, yorktown heights, new york, edited by raymond e. miller and james w. thatcher, plenum press, new york and london 1972, pp. 85–103." *The Journal of Symbolic Logic*, vol. 40, no. 4, pp. 618–619, 1975.

[13] T. Erlebach and K. Jansen, "The complexity of path coloring and call scheduling," *Theoretical Computer Science*, vol. 255, no. 1, pp. 33–50, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397599001528

[14] F. Guan, J. Qiao, and Y. Han, "Dag-fluid: A real-time scheduling algorithm for dags," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 471–482, 2021.

[15] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993, pp. 345–354.

[16] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 2006, pp. 101–110.

[17] A. Block and J. Anderson, "Accuracy versus migration overhead in real-time multiprocessor reweighting algorithms," in *12th International Conference on Parallel and Distributed Systems - (ICPADS'06)*, vol. 1, 2006, pp. 10 pp.–.

[18] F. Guan, L. Peng, and J. Qiao, "A fluid scheduling algorithm for dag tasks with constrained or arbitrary deadlines," *IEEE Transactions on Computers*, vol. 71, no. 8, pp. 1860–1873, 2022.

[19] F. Gua n, L. Peng, and J. Qiao, "A new federated scheduling algorithm for arbitrary-deadline dag tasks," *IEEE Transactions on Computers*, vol. 72, no. 8, pp. 2264–2277, 2023.

[20] X. Jiang, J. Sun, Y. Tang, and N. Guan, "Utilization-tensity bound for real-time dag tasks under global edf scheduling," *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 39–50, 2020.

[21] X. Jiang, N. Guan, H. Liang, Y. Tang, L. Qiao, and Y. Wang, "Virtually-federated scheduling of parallel real-time tasks," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 482–494.

[22] X. Jiang, H. Liang, N. Guan, Y. Tang, L. Qiao, and Y. Wang, "Scheduling parallel real-time tasks on virtual processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 1, pp. 33–47, 2023.

[23] T. Kobayashi and T. Azumi, "Work-in-progress: Federated and bundled-based dag scheduling," in *2023 IEEE Real-Time Systems Symposium (RTSS)*, 2023, pp. 443–446.

[24] Q. He, N. Guan, M. Lv, and Z. Gu, "On the degree of parallelism in real-time scheduling of dag tasks," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[25] S. Zhao, X. Dai, and I. Bate, "Dag scheduling and analysis on multi-core systems by modelling parallelism and dependency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4019–4038, 2022.

[26] X. Jiang, N. Guan, X. Long, and H. Wan, "Decomposition-based real-time scheduling of parallel tasks on multicores platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2319–2332, 2020.

[27] Q. He, x. jiang, N. Guan, and Z. Guo, "Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2283–2295, 2019.

[28] S. Xiao, D. Li, and S. Wang, "Periodic task scheduling algorithm for homogeneous multi-core parallel processing system," in *2019 IEEE International Conference on Unmanned Systems (ICUS)*, 2019, pp. 710–713.

[29] J. Shi, M. Gtinzel, N. Ueter, G. v. der Bruggen, and J.-J. Chen, "Dag scheduling with execution groups," in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024, pp. 149–160.

[30] Y. Guan, B. Zhang, and Z. Jin, "An frtds real-time simulation optimized task scheduling algorithm based on reinforcement learning," *IEEE Access*, vol. 8, pp. 155 797–155 810, 2020.

[31] Z. Xu, Y. Zhang, S. Zhao, G. Chen, H. Luo, and K. Huang, "Drl-based task scheduling and shared resource allocation for multi-core real-time systems," in *2023 IEEE 3rd International Conference on Intelligent Technology and Embedded Systems (ICITES)*, 2023, pp. 144–150.

[32] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering – a systematic literature review," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009, special Section - Most Cited Articles in 2002 and Regular Research Papers. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584908001390

[33] R. Wieringa, "Design science methodology: principles and practice." New York, NY, USA: Association for Computing Machinery, 2010.

[34] C. Sarah, C. Kathrin, R. Ann, H. Guro, A. Anthony, and S. Aziz, "The case study approach," *BMC Research Medical Methodology*, vol. 11, no. 1, p. 100, 2011.

[35] V. R. Basili, "The role of controlled experiments in software engineering research," in *Empirical Software Engineering Issues. Critical Assessment and Future Directions: International Workshop, Dagstuhl Castle, Germany, June 26-30, 2006. Revised Papers*. Springer, 2007, pp. 33–37.

[36] G. C. Buttazzo, "Rate monotonic vs. edf: Judgment day," *Real-Time Systems*, vol. 29, pp. 5–26, 2005.

[37] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun, "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 3438–3445.

[38] E. Yip and M. M. Kuo, "Letsynchronise: An open-source framework for analysing and optimising logical execution time systems," in *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*, 2023, pp. 349–354.

[39] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," *Advances in Real-Time Systems*, pp. 103–120, 2012.

[40] A. Torralba and A. A. Efros, "Unbiased look at dataset bias," in *CVPR 2011*. IEEE, 2011, pp. 1521–1528.

[41] H. Jabbar and R. Z. Khan, "Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study)," *Computer Science, Communication and Instrumentation Devices*, vol. 70, no. 10.3850, pp. 978–981, 2015.