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

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

***Abstract*—*Background*: Previous research and/or rationale for performing the study.**

***Aims*: Hypotheses/propositions to be tested, or goal of the study.**

***Method*: Description of the type of study, treatments, number and nature of experimental units (people, teams, algorithms, programs, tasks etc.), experimental design, outcome being measured.**

***Results*: Treatment outcome values, level of significance.**

***Conclusions*: Limitations of the study, implications of the results, and further work**

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

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 (or flexible) execution time constraints, known as deadlines. Failure to meet these deadlines can lead to severe consequences. The critical nature of these systems necessitates designing the system architecture with a focus 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. Developing and running tasks on these architectures require in-depth knowledge of the system and its architecture, which complicates code reusability and scalability when adding hardware resources or upgrading to a larger system.

To address these issues, the Automotive Open System Architecture (AUTOSAR[1]) was developed. AUTOSAR introduces layers of abstraction between hardware, firmware, and software, enhancing software reusability and hardware scalability across different systems while maintaining safety and security standards. It is now the most widely used architecture among car manufacturers, with notable core partners including BMW, Ford, and Toyota.

[1]https://www.autosar.org/

Scalability, in particular, plays a crucial role in modern real-time systems. Increasingly, real-time systems such as autonomous cars or computer vision systems 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, aiming to reduce both the execution time of these tasks and the required resources [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 [2]. 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. 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 [4] provided two major priority policies: Rate Monotonic (RM) and Earliest Deadline First (EDF) for scheduling 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 is the time-triggered task model,

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

which specifies tasks that execute periodically and is well-suited for time-triggered systems. Another type of task is the Logical Execution Time (LET) task. The LET paradigm is based on the time-triggered paradigm and was originally introduced by the Giotto real-time programming language [5] and later refined by [6] into the Hierarchical Timing Language (HTL). The main principle behind the LET paradigm is that each task's inputs and outputs are read and written in zero time, i.e., constant time.

The benefits of using LET are twofold. Firstly, the zero-time communication semantics greatly improve the predictability of the overall system and make I/O operations (i.e., memory access) on shared resources deterministic, which is crucial in real-time systems due to the highly negative impact that memory access contentions can have on the system [7]. Secondly, using LET in programming also provides a layer of abstraction that facilitates the direct translation from modeling to implementation, thus ensuring the implementation of timing requirements, enhancing code maintenance, and producing a less error-prone code base [8].

One drawback of LET is its implementation overhead, which increases the execution times of tasks due to the zero-time communication semantics [9]. Despite this drawback, the advantages of LET make it attractive for real-time systems [10], which is why the focus here will be on time-triggered and LET task scheduling on multi-core systems. Given that the problem of scheduling independent tasks is NP-hard[3] [11], no scalable optimal algorithm exists. Therefore, heuristics are used to partially solve the problem.

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. In other words, the research questions are:

- RQ1 What is the current state-of-the-Art in scheduling event-chains of tasks ?
  - RQ1.1 What is the current state-of-the-Art for DAG task scheduling with precedence constraints ?
  - RQ1.2 How has LET been used in scheduling event-chains ?
  - RQ1.3 What machine learning techniques are used for DAG task scheduling ?
- RQ2 Can machine learning be a better solution to schedule event-chains of tasks ?
  - RQ2.1 Can a machine learning solution compare to state-of-the art heuristics for scheduling Directed Acyclic Graph tasks ?
  - RQ2.2 Can a machine learning solution compare to ILP solutions 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 R1, and finally, the artifact and experimental design, results, and conclusion will be presented to answer R2.

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

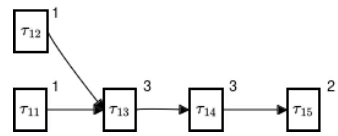

Fig. 1. DAG task τ_1. The nodes are the subtasks, the edges of the graph represent the precedence constraints between each subtasks and the worst-case execution time (wcet) of each subtask written as an exponent.

The solution we propose has the following features..
The primary contributions of this paper are:

II. Background

Task scheduling introduces several fundamental concepts.

A. Periodic task and schedule

Firstly, a periodic task $\tau_i(C_i, D_i, T_i)$ is characterized by its worst-case execution time (wcet) C_i, its deadline D_i, and its period T_i. This definition can be expanded by including an initial offset, which corresponds to the time of the task's first execution, and an activation offset, which is the time delay between the task being ready to execute (i.e., its execution period has begun) and the task actually starting to run. Secondly, a schedule S is a function that assigns a boolean value for each task τ and each time tick t, indicating whether the task τ is running at time t. Therefore, a scheduling algorithm is the method that, given a set of tasks, produces a schedule S for the task set.

This task model and schedule definition are widely adopted in the literature (see section III) and are the building blocks of all scheduling algorithms. The periodic task model, in particular, is used to define more complex tasks such as DAG tasks (see below) that will be used as input in the machine learning model (see section IV).

B. DAG task

A Directed Acyclic Graph (DAG) task is a task that models the multiple subtasks of an chain of tasks that have a precedence constraints. For example, when considering the task τ_1 that makes an aircraft keep its altitude, you usually have a number of subtasks to handle this task, namely : reading from the altitude sensor (τ_{11}), reading for the speed sensor (τ_{12}), computing the new speed for the aircraft to keep its altitude ($\tau_1 3$), computing the amount of thrust needed to achieve this new speed ($\tau_1 4$), and finally actuating the aircraft's jet engine ($\tau_1 5$). In this example, the DAG for τ_1 can be seen in Figure 1.

A DAG task τ_i also has a period T_i and a wcet C_i which is the sum of its subtasks wcets, and a deadline D_i. For instance, according to Figure 1, the wcet for τ_1 is 10 time units. You can also see how, for τ_1, the subtasks τ_{12} and τ_{11} can be parrallelized (i.e., executed in parallel) but the subtask τ_{13} needs to wait for both τ_{11} and τ_{12} to finish their

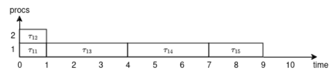

Fig. 2. Example of schedule for τ_1. The y axis represents the number of processors that are not idle. For the first two subtasks there are two processors active and for the rest there is only one active processor.

execution before it can start running.

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).

C. 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

$$U = \sum_{k=1}^{n} \frac{C_k}{T_k} \tag{1}$$

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 (see Section III), or used as a parameter to generate tasksets or DAG tasks with a fixed utilization (see Section IV).

D. 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 τ_1 shown in Figure 1, the makespan of τ_1 for the schedule shown in Figure 2 is 9. Notice that in Figure 2, if the subtasks τ_{11} and τ_{12} were executed sequentially instead of in parrallel, the makespan would be one time unit longer, in this case 10 instead of 9.

This is a key measurement when dealing with DAG tasks (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).

E. 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 measurement, to assert the efficiency of scheduling algorithms when considering independent tasks (see Section III).

F. Optimality

A scheduling algorithm S is said to be optimal when the following condition is true: for every taskset Ω, if there exists a scheduling algorithm S' so that Ω is feasible by S', then Ω is also feasible by S. Where *feasible*, means that, using the schedule generated by S, all the tasks in the taskset will finish executing before their deadlines.

This concept is used in the literature, mainly for independent tasks scheduling (see Section III).

G. Approximation ratio

The approximation ratio is the comparison between the average number of processors required by a scheduling algorithm to make a random taskset feasible and the average number of processors needed by the theoretically optimal scheduling algorithm for the same taskset.

It is a way of measuring scheduling algorithms, especially when considering independent tasks (see Section III).

While the acceptance and approximation ratio are used to measure the performance of scheduling alrogithms for independent tasks, the makespan is only used for DAG tasks and tasksets representing chain of events.

H. RM and EDF scheduling

When designing a scheduling algorithm, the key decision involves determining which task should execute first when two or more independent tasks are ready to execute. This requires assigning each task a priority. [4] introduced two heuristics for this purpose: Rate Monotonic (RM) and Earliest Deadline First (EDF).

The RM algorithm is a fixed-priority scheduling algorithm, meaning that the priority of each task is known before execution begins. RM assigns the highest priority to tasks with the minimum execution rate, i.e., $\frac{C_k}{T_k}$, and is considered optimal for assigning fixed priorities to tasks. In contrast, EDF assigns priorities dynamically by selecting tasks based on which one has the earliest absolute deadline.

Figure 3 illustrates the difference between the two algorithms by scheduling the same two tasks, τ_1 and τ_2. τ_1 has a worst-case execution time of 0.5 time units and a period of 2 time units, while τ_2 has a worst-case execution time of 2 time units and a period of 3 time units. These are examples of implicit deadline tasks, where the relative deadline equals the end of their execution period.

Although EDF calculates each priority at runtime, it is optimal for uniprocessor scheduling and has a theoretical utilization bound of 1, which is the maximum possible for

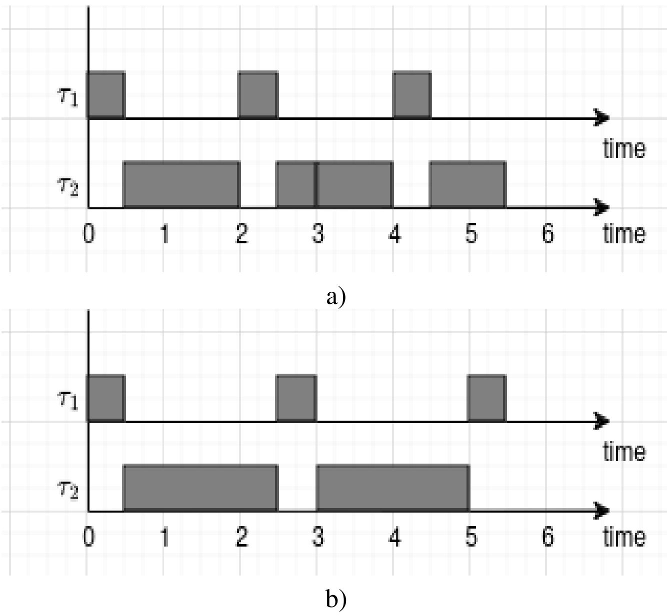

Fig. 3. Schedules of τ_1 and τ_2 using Rate Monotonic (a) and Earliest Deadline First (b) heuristics.

a feasible taskset on a single processor. RM, on the other hand, has a much lower utilization bound than EDF. While one might argue that RM introduces less runtime overhead and is therefore more practical, it has been shown that RM leads to more task preemptions (interrupting the execution of a task, as seen at times 2 and 4 for task τ_2 in Figure 3.a). This, combined with its lower utilization bound and non-optimality, makes EDF clearly superior to RM [12].

Although [4]'s work focused on uniprocessor systems, the proposed algorithms have also been applied to multi-processor scheduling.

III. Related Works

A. Systematic Literature Review process

Scoping

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

RQ1.1 What is the current state-of-the-Art for DAG task scheduling with precedence constraints ?

RQ1.2 How has LET been used in scheduling event-chains ?

It will also be shown how the literature doesn't provide a complete answer to RQ2, hence the contributions of this paper.

From these research questions, several concepts have been isolated, namely, time-triggered tasks, the nature of the system (real-time multicore system), the scheduling of tasks, DAG 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 "LET" OR "Logical Execution Time" 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 3,549 results on the IEEE database
exclusion : past 5 years : IEEE –¿ down to 1,171 heterogeneous not in title + abstract : IEEE –¿ down to 999 mixed critical* not in title + abstract : IEEE –¿ down to 952 scheduling or scheduler or schedule in title but not "energy" : IEEE –¿ down to 155 and 149 when just considering conference and journal papers (not early access)

removing those not about real-time system, not about proposing a scheduling algorithm, not about DAG nor LET tasks or event-chains : IEEE –¿ down to 21

a) :

B. Findings of the Literature Review

The works reviewed were compared on the following metrics.

- **Utilization Bound**: useful to see which algorithm is more efficient at using the available resources.
- **Acceptance Ratio**: it shows how optimal (see Section II) a scheduling algorithm is.
- **Makespan**: for DAG task scheduling, widely used in the literature.
- **Runtime Overhead**: some scheduling algorithms can show promising results theoretically but are practically very slow because of their complexity adding runtime overhead on the scheduler, this metric will not be a number but rather an amount such as minimal, practical, non-practical.

Every metric used here have also been chosen for their prevalence in the literature.

A comparison of the works was carried out and the overall results are illustrated in Table **??**.

1) DAG tasks:

Scheduling DAG tasks involves two steps: first, computing the intra-task schedule, and second, computing the inter-task schedule. For the inter-task schedule, various approaches can be employed.

[**?**], for instance, improve the worst-case makespan of GEDF under federated scheduling of multiple DAG tasks with arbitrary deadlines. The federated scheduling approach, similar to the partitioning approach, involves assigning clusters of processors to DAG tasks with the highest utilization, leaving the remainder for low-utilization tasks. This allows for intra-task parallelization instead of merely sequentializing the DAG task. Consequently, the paper also improves the acceptance ratio for single DAG task scheduling by providing a better schedulability test compared to previously used schedulability tests for GEDF.

Another method is the decomposition approach, which involves decomposing the DAG task into several independent sequential tasks. These tasks are then executed in parallel segments, with their release times and deadlines aligned to match the dependency constraints between the sequential tasks [?]. The paper [?] introduces a state-of-the-art stretching method for DAG task decomposition and employs the GEDF dynamic priority assignment algorithm to demonstrate improvements in the acceptance ratio over the previous state-of-the-art decomposition algorithm.

[?] explores a thread pool approach for parallelizing DAG tasks. Instead of assigning processors to DAG tasks, thread workers from a thread pool are assigned, and the number of thread workers each DAG can use is limited. Their algorithm, combined with a global fixed-point priority scheduling algorithm such as Rate Monotonic, is compared to other approaches, including the global approach and the semi-federated approach.

The global approach does not assign processors to DAG tasks but allows the tasks to utilize multiple processors dynamically. The semi-federated approach, similar to the federated approach, places as many heavy tasks (tasks with high utilization) in processor clusters as possible, while the remaining tasks, along with the light tasks, are allocated to the rest of the available processors.

It is found that the latest approach generally outperforms the method used in [?], although the thread pools approach still has a better acceptance ratio compared to the global approach.

The previously cited articles focus on the execution of multiple DAGs on a multi-processor system, utilizing existing priority scheduling algorithms such as GEDF or Global RM to evaluate their contributions. [?] considers recurrent DAG tasks and their inner graph structure to develop a priority assignment algorithm that minimizes the makespan of the DAG task. This algorithm is then extended to multi-DAG scheduling using a global scheduling approach such as G-EDF or G-RM. This dynamic scheduling approach performs best with G-RM in terms of acceptance ratio.

The results in terms of acceptance ratio are superior to those in [?], though in some cases they are surpassed by [?]. However, a drawback of [?] is that this type of task decomposition incurs significant runtime overhead, which diminishes task performance in real-life scenarios.

While [?] and [?] allow for task preemption, which especially in the case of [?] adds runtime overhead, [?] leverages the parallelism and dependency properties of DAG tasks, along with a 'critical path first' execution strategy, to develop a state-of-the-art non-preemptive and priority-based scheduling algorithm that completely outperforms [?] in terms of makespan. Their results are utilized by [?] to compare with a deep learning-based priority assignment algorithm for DAGs, which improves the makespan of DAG task execution by 2∼3%.

[?] extends their concurrent provider and consumer (CPC) model [?] to multi-DAG scheduling by minimizing inter-task DAG interference to zero and devising a processor-assigning scheduling algorithm where the priority of different DAG tasks is computed using the deadline-monotonic algorithm. Under non-preemptive scheduling, the proposed method significantly

Reference	**Motivation**	**Contribution(s)**	**Limitation(s)**	**Methodology Summary**
[13]	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 sub-tasks into multiple sequential segments
[14]	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 possibe through the current vertex to assign the priority to the current vertex, the higher the length, the higher priority
[15]	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.
[16]	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 combines techniques from clustering scheduling and task duplication algorithm and evaluates it on an actual simulation object (the TMS320C6678) so that the measurement are close to the real-life measurement you would get on the real system.
[17]	The use of multi-core systems can induce contentions because of shared memory / cache. This can lead to non-determinim and unpredictable behavior which violates the safety requirements of real-time systems, hence using LET tasks to fix those contentions	Proposes a DAG LET tasks scheduling algorithm based that avoids contentions while reducing the running time overhead due to LET implementation	They are using a multiple-clusters, multi-core architecture to evaluate their scheduling algorithm but only consider one cluster.	Uses a minimum-laxity-first priority assignment for intra-tasks and Earliest Finish time (EFT) for assigning tasks to cores. Considers multi-rate dags and reducing the LET interval to decrease the makespan.

outperforms the method used in [?] in terms of acceptance ratio, by up to 60%.

For multi-DAG scheduling, [?] employed the fluid scheduling strategy to manage DAG tasks with implicit deadlines. This fluid scheduling approach, also used in PFair and LLREF scheduling algorithms [33] [34], ensures that at every point in time, each task has utilized the amount of execution time dictated by its respective utilization factor, thereby approximating the perfect fluid execution of the task. The advantage of this approach is its exceptionally high acceptance ratio, outperforming other methods such as [?], [?], and [?]. However, it suffers from high runtime overhead, complicating practical implementation.

A more mathematical approach to the scheduling of DAGs is to model the scheduling problem as an Integer Linear Programming (ILP) optimization problem. In this model, precedence, deadline, and processor assignment constraints are represented mathematically, with the objective of minimizing the makespan. This method is utilized by [?] and compared to state-of-the-art priority assignment algorithms ([?] and [?]). The ILP method demonstrates a significant improvement in makespan, which is expected due to the optimality of the ILP approach. However, the drawback of this method is that as the number of tasks and subtasks increases, the number of constraints grows, causing the computation time to increase exponentially, rendering the method non-scalable.

Most studies do not consider the communication time between tasks, which can be significant in real-life systems. [?] addresses this by scheduling DAG tasks on a Network on Chip (NoC) system. The resulting schedule, DAG-Order, is non-preemptive and is based on ordering the tasks according to their communication delays and computation workloads.

Memory access contention, which occurs when two or more tasks attempt to access a shared memory location simultaneously, can also be crucial in real-life scenarios. Therefore, scheduling algorithms for implementations of the LET paradigm have been proposed to reduce or even eliminate contention problems with LET DAG tasks [?] [35].

The machine learning community has also explored DAG scheduling. For instance, [?] utilized reinforcement learning (RL), specifically Q-learning, to statically prioritize sub-tasks within DAG tasks and applied an earliest-start-first (EST) heuristic value to dispatch each sub-task to different processors. Similar to [?], [?] accounted for communication delays and the workload of subtasks when assigning priorities.

Another application of RL is demonstrated by [?], who designed a deep learning model based on RL that uses the spatial features of each DAG task as well as their temporal features, i.e., precedence constraints. They achieved this by combining a graph convolution network (for spatial information) with a sequential encoder (for temporal information), ultimately producing a prioritized list of the DAG's subtasks. This list can then be used to compute the makespan and optimize it via RL. The results in [?] were compared with state-of-the-art (SOTA) algorithms [?] [?], with the deep reinforcement learning method surpassing the SOTA by up to 3% in terms of makespan.

References	Category	Method
[?] [?]	Decomposition	task segmentation and processor exclusivity for critical path [?], and fluid scheduling [?]
[?] [?] [?] [?]	Partitioned/ Federated	Federated scheduling for inter-DAG scheduling and GEDF for intra-DAG [?], global preemptive fixed-priority scheduling using assigned thread-workers based on DAG workload [?], workload and no inter-task interference based processor assignment with CPC model for intra-task priority assignment [?], federated and order-based intra-task priority assignment based on workload and communication delays [?]
[?]	Global	G-RM and G-EDF for inter-task and priority assignment by maximizing intra-task parralelism for intra-task scheduling
[?]	ILP	Only interested in intra-task interference, uses Integer Linear Programming to minimize the makespan
[?] [?]	Reinforcement Learning	Q-learning with partitioning at the intra-task level using EST heuristic [?], GCN[4]and sequential encoding priority assignment for a single DAG task and then use a work-conserving GFPS[5]to assign tasks to processors [?]
[35] [?]	mixed global / partitioned	blocks access from main tasks to certain cores to avoid contention by interpreting I/O operations as tasks with precedence constraints and parallelizing the main tasks

TABLE II
SUMMARY TABLE FOR DAG TASK SCHEDULING.

As you can see, although dynamic priority algorithms outperform fixed-priority ones, the simplicity of implementation and low runtime overhead of fixed-priority algorithms make them attractive to the industry. This is especially true for the DAG task model, where there has been significant focus on fixed-priority scheduling. While some have attempted to produce an 'optimal' schedule using ILP [36] [37] [?], the primary issue with this method is its lack of scalability.

Regarding task migrations, the NP-hard nature of the bin-packing problem suggests that allowing tasks or subtasks to migrate between processors can improve utilization performance.

Also, only 3 articles used machine learning to tackle the task scheduling problem, from which two are from the same author. Furthermore, those articles only compare their results to heuristic-based methods and not ILP methods. If we focus on DAG tasks, then only 2 papers are left, one focusing on communication between the cores and applying the model on a specific architecture [**?**], and one more theoretical [**?**], comparing their model to SOTA [**?**] and [**?**]. The latter suffers from closed sourcing as their model is not open source which prohibits the research community to improve on their work.

Hence, there not only is a need to compare one such machine learning technique to the non-scalable but leading to the mathematically minimum makespan, ILP method, but there also is a need to have this model open source and open access. Therefore, in this paper, an attempt at replicating the model described in [**?**] will be done and a comparison with the SOTA heuristic-based algorithms and ILP will be conducted using the open-source software for LET task scheduling LETSyncronize [38].

IV. Research Methodology

This research could have been conducted using alternative methodologies. Provide a summary of the best fit methodologies, and their relative strengths and weaknesses (max 2-3 paras).

The "name of methodology" was chosen to conduct this research. Give details about how this methodology was adapted for your project (max 2-3 paras).

Include a clear plan with objectives and outcomes (gantt chart)

V. Contribution 1

Oh by the way, [**?**] did some great work.

Give an overall summary of the steps involved.

VI. Contribution 2

Give an overall summary of the steps involved.

VII. Experimental Results

Set up: what experiments/benchmarks were chosen?

Execution of results: how were the experiments conducted

Data: what was found to have happened?

Synthesis: what does the data mean?

Relevance: how does this work compare to others, and to what extent does it answer the RQs

Limitations:

VIII. Conclusions and Future Works

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] 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.

[5] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[6] T. A. Henzinger, C. M. Kirsch, E. R. Marques, and A. Sokolova, "Distributed, modular htl," in *2009 30th IEEE Real-Time Systems Symposium*. IEEE, 2009, pp. 171–180.

[7] K. Nagalakshmi and N. Gomathi, "The impact of interference due to resource contention in multicore platform for safety-critical avionics systems," *Int. J. Res. Eng. Appl. Manage.*, vol. 2, no. 8, pp. 39–48, 2016.

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

[9] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the let paradigm. in 2018 ieee real-time and embedded technology and applications symposium (rtas)," 2018.

[10] K.-B. Gemlau, L. Köhler, R. Ernst, and S. Quinton, "System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software," *ACM Transactions on Cyber-Physical Systems*, vol. 5, no. 2, pp. 1–27, 2021.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] S. Igarashi, T. Ishigooka, T. Horiguchi, R. Koike, and T. Azumi, "Heuristic contention-free scheduling algorithm for multi-core processor using let model," in *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 2020, pp. 1–10.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] F. Guan, 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.

[23] 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.

[24] 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.

[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] 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.

[27] 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.

[28] 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.

[29] 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.

[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] 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.

[32] P. Pazzaglia, D. Casini, A. Biondi, and M. D. Natale, "Optimal memory allocation and scheduling for dma data transfers under the let paradigm," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1171–1176.

[33] 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.

[34] 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.

[35] S. Igarashi, T. Ishigooka, T. Horiguchi, R. Koike, and T. Azumi, "Heuristic contention-free scheduling algorithm for multi-core processor using let model," in *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 2020, pp. 1–10.

[36] T. Wei, X. Chen, and S. Hu, "Reliability-driven energy-efficient task scheduling for multiprocessor real-time systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 10, pp. 1569–1573, 2011.

[37] E. Yip, M. M. Kuo, P. S. Roop, and D. Broman, "Relaxing the synchronous approach for mixed-criticality systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 89–100.

[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.