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*—*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.

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, 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

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

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

(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 ?
- 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.

The solution we propose has the following features..

The primary contributions of this paper are:

II. Background

Task scheduling introduces several fundamental concepts.

[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

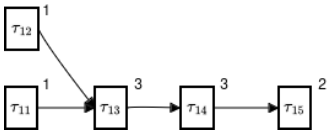

Fig. 1. DAG task τ_1. The nodes are the subtasks, the edges of the graph represent the precedence constraints between each subtasks and the worst-case execution time (wcet) of each subtask written as an exponent.

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 (τ_13), computing the amount of thrust needed to achieve this new speed (τ_14), and finally actuating the aircraft's jet engine (τ_15). 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 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).

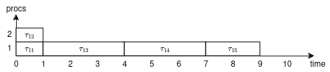

Fig. 2. Example of schedule for τ_1. The y axis represents the number of processors that are not idle. For the first two subtasks there are two processors active and for the rest there is only one active processor.

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 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,

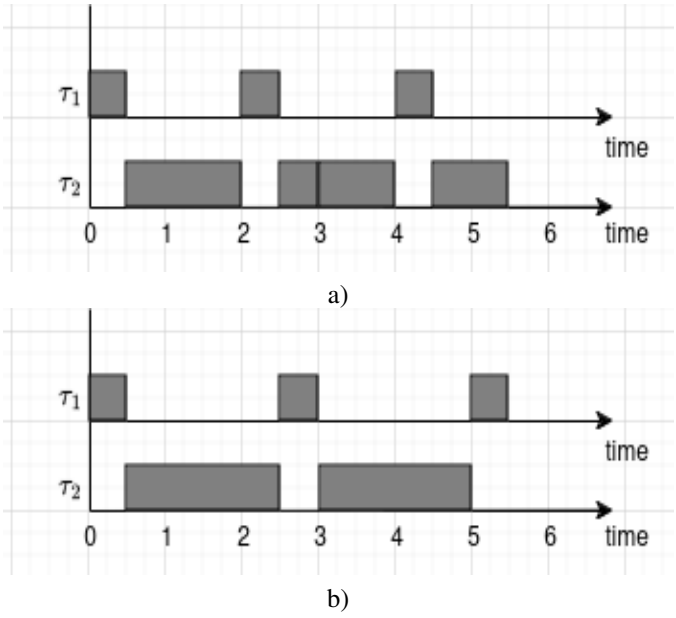

Fig. 3. Schedules of τ_1 and τ_2 using Rate Monotonic (a) and Earliest Deadline First (b) heuristics.

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 ?

RQ1.3 What machine learning techniques have been used for scheduling tasks on real-time systems ?

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 After reading the complete articles –¿ 19.

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 **??**.

In [13], authors 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 [14] [15] 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 [13] 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. Also, their algorithm only considers DAG task with implicit deadlines (D = T) which makes the response-time analysis simpler but to the cost of generalizability.

As a follow up, authors in [16] extend the fluid scheduling algorithm in [13] to constrained and arbitrary deadline tasks, especially focusing on DAG tasks with a deadline greater than

their period. 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 [17] [18] [19] [20] [21] [22] [23].

[18], 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.

This problem of resource wasting in federated scheduling is also tackled in [21] where the authors 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.

Authors in [19] take another approach by proposingg a virtually-federated scheduling algorithm that leverages the advantages of federated scheduling while improving the acceptance ratio for DAG tasks, outperforming existing algorithms.

Reference	**Scheduling technique**	**Task type**	**Scope (intra/inter/both)**
[13]	fluid	implicit deadline	inter
[24]	priority-list	constrained deadline	intra
[21]	federated and bundled-based	constrained deadline	inter
[25]	clustering	constrained deadline	intra
[26]	priority-list	LET constrained deadline	both
[18]	federated and GEDF and PEDF	implicit deadline	inter
[27]	Decomposition-based	implicit deadline	inter
[23]	federated-based	constrained deadlines	inter
[22]	partitioned / clustering	constrained deadlines	intra
[17]	federated	arbitrary deadline	inter
[28]	DRL	constrained deadline	intra
[29]	DRL	non-DAG implicit deadline	inter
[30]	priority-list and federated	constrained deadline	both
[31]	DRL	constrained deadline	intra
[20]	federated-based	constrained deadline	inter
[16]	fluid	constrained/arbitrary deadline	inter
[32]	DRL	constrained deadline	intra
[19]	federated-based	constrained deadline	inter
[33]	Mixed ILP	LET, constrained deadline	inter
Total: 19	**DRL: 4, Federated: 7, Fluid: 2, ILP: 1, Priority-List(intra): 3, Clustering: 2, Decomposition: 1**	**implicit: 4, constrained: 13, arbitrary: 2**	**inter: 11, intra: 6, both: 2**

TABLE I
SLR SUMMARY TABLE

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] 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.

[15] 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.

[16] 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.

[17] ——, "A new federated scheduling algorithm for arbitrary-deadline dag tasks," *IEEE Transactions on Computers*, vol. 72, no. 8, pp. 2264–2277, 2023.

[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, 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.