Proceedings of the

15th Junior Researcher Workshop on Real-Time Computing 2022

(JRWRTC 2022)

Paris, France

07 - 08 June 2022

Table of contents

Junior Researcher Workshop on Real-Time Computing 2022

(JRWRTC 2022)

Message from the workshop chairs 5
Program committee 6

An automata-based method for interference analysis in multi-core processors ………………7
Thomas Beck, Frédéric Boniol, Jérôme Ermont, Luc Maillet, and Franck Wartel

Data Access Time Estimation in Automotive LET Scheduling with Multi-core CPU ……….11
Risheng Xu, Max J. Friese, Hermann von Hasseln, and Dirk Nowotka

Joint Scheduling, Routing and Gateway Designation in Real-Time TSCH Networks ………16
Miguel Gutiérrez Gaitán, Luís Almeida, Thomas Watteyne, Pedro M. d'Orey,
Pedro M. Santos, and Diego Dujovne

Toward Precise Real-Time Scheduling on NVidia GPUs ……………………………………….20
Nordine Feddal, Houssam-Eddine Zahaf, and Giuseppe Lipari

Shortening gate closing time to limit bandwidth waste when implementing Time-Triggered scheduling in TAS/TSN ……………………………………………………………………………25
Pierre-Julien Chaine and Marc Boyer

Message from the workshop chairs

JRWRTC 2022

It is our great pleasure to welcome you to the Junior Researcher Workshop on Real-Time Computing 2022, which is held conjointly with the 30th International Conference on Real-Time Networks and Systems (RTNS 2022). The main purpose of JRWRTC 2022 is to bring together junior researchers (Ph.D. students, postdocs, etc.) working on real-time systems. It is not only a good opportunity to present (ongoing) work and share ideas with other junior researchers, but also a chance to engage in discussions with and to receive feedback from the audience of the main conference.

This year, five peer-reviewed papers have been accepted, which were reviewed thoroughly by the international program committee. We hope that the program committee's detailed comments and remarks will help the authors to submit more mature long versions of their papers to the next edition of RTNS and would like to thank the members of the program committee for their effort.

JRWRTC 2022 would not be possible without the generous contribution of many volunteers and institutions which support RTNS 2022. Therefore, we would like to express our sincere gratitude to our sponsors for their financial support: INRIA and StatInf. We would like to thank the general chairs Liliana Cucu-Grosjean and Yasmina Abdeddaïm for giving us the opportunity to chair JRWRTC 2022. In particular, we thank Liliana Cucu-Grosjean for her valuable help during the organization of the workshop. Moreover, we are very grateful to the web chair Kevin Zagalo, the multimedia chair Mohamed Amine Khelassi, and the publicity chair Georg von der Brüggen for their support. Not least, we thank the local organizing committee and all others who helped to organize the conference – one of the first in our domain to return to an in-person/hybrid format. Let us all look forward to a successful continuation of the RTNS conference series!

Stéphan Plassart, EPFL, Switzerland
Lea Schönberger, TU Dortmund University, Germany

Workshop chairs

Program Committee

JRWRTC 2022

Abderaouf Nassim Amalou, University of Rennes, Inria, CNRS, IRISA, France

Pierre-Julien Chaine, Airbus, France

Anam Farrukh, Boston University, USA

Frédéric Fort, University of Lille, France

Anna Friebe, Mälardalen University, Sweden

Damien Guidolin, RealTime-at-Work, France

Alexandre Honorat, Inria Grenoble, France

Sena Houeto, University of Colorado Colorado Springs, USA

Matheus Ladeira, ENSMA, France

Reza Mirosanlou, University of Waterloo, Canada

Sims Osborne, University of North Carolina, Chapel Hill, USA

Junjie Shi, TU Dortmund University, Germany

Seyed Mohammadhossein Tabatabaee, EPFL, Switzerland

Aaron Willcock, Wayne State University, USA

Kevin Zagalo, Inria Paris, France

An automata-based method for interference analysis in multi-core processors

Thomas Beck
Airbus Defence and Space
Toulouse, France

Frédéric Boniol
ONERA
Toulouse, France

Jérôme Ermont
IRIT - INP - ENSEEIHT
Toulouse, France

Luc Maillet
Airbus Defence and Space
Toulouse, France

Franck Wartel
Airbus Defence and Space
Toulouse, France

1. INTRODUCTION

Multi-core processors (MCPs) provide huge gains that allow replacing several embedded single-core processors with a smaller number of computing platforms. However, such processors face important challenges to their integration into safety-critical systems. Due to resource sharing, unwanted interferences may exist between the tasks hosted by the different cores that can cause unexpected delays.

Certification authorities have published a set of recommendations [2] for designers who want to certify a MCP hosting a set of safety-critical tasks. Basically, this entails a two steps process: First, the designer must identify all the interferences that can occur between the tasks. Second, he must show that the severity of the interferences is compliant with the real-time requirements of the tasks. Many works have faced this interference challenge in MCPs. The reader can refer to [4] for a large and complete overview of the scientific work on timing verification for MCPs. Among this work, a recent one [1] has shown that it is possible to characterise tasks running on MCPs by a timed sequence of Time Interest Points (TIPS), i.e., a sequence of instructions on which the tasks can suffer from interference.

We focus on the issue of the identification of interferences. We refine the notion of interference with two finer definitions, related to the type of the components causing the interference. And, considering that tasks can be abstracted by timed sequences of TIPS, we develop an automata-based method, inspired from [3], to count the interferences that can occur between the tasks hosted by the processor.

2. METHOD OVERVIEW

The landscape in which the method is involved is shown in Figure 1. Within this landscape, the scope of the paper is highlighted by the gray box. Let us consider a set of tasks τ_i hosted by a MCP (on the left of the Figure). The objective of the method is to compute an upper bound of the extra timing penalty associated with each τ_i due to interference occurring in the architecture (on the right part of the Figure). For that purpose, we compute an upper bound of the maximal interference number (noted IN_i) from which each τ_i can suffer. The approach relies on the analysis method proposed by Carle *et al.* in [1] (on the left part of Figure 1). This method is based on the TIPS notion, which are load and store instructions that generate and suffer from interference. By means of static analysis, Carle *et al.* show that it is possible to extract a temporal segment sequence, as the one shown in Figure 2, from the binary code of a task . Each segment is characterised by a duration (noted $d_{i,j}$ for task τ_i and segment j), and the maximal number of memory requests leaving the core and sent to the bus (noted $\mu_{i,j}$). These requests correspond to load and store operations that are not *always hit* in the L1 cache of the core hosting the task. For instance in Figure 2, τ_1 makes zero memory request in segment one, and at most three requests in segment two. Such a sequence defines a *bus access profile* of the task under consideration. It characterises its worst-case memory activity that can lead to interference. As shown in Figure 1 we take these profiles as inputs, and we study an automata-based method to count, by model-checking, the interference that can occur for each τ_i .

3. DEFINITIONS

Let us consider an archetypal MCP architecture depicted Figure 3. This processor is composed of (1) two cores (C_0 and C_1) owning their private cache $L1$, (2) a shared cache $L2$, (3) a DDR memory composed of one bank B, and (4) a shared bus allowing the two cores to address the $L2$ cache and the DDR. Each core C_i hosts a task called τ_i.

Let us suppose that τ_0 and τ_1 are characterised by the profiles shown in Figure 4. They are divided into three segments. τ_0 and τ_1 can compete to access the memory in the first segment (from 0 to 10). In this segment, each task sends at most 2 memory requests. In the next two segments, either τ_0 or τ_1 does not send any request, leaving the memory path available for the other task.

Requests from τ_0 and τ_1 could collide in the bus. If they arrive at the same time, only one of them can pass, the second one must wait. The effect of the interference is a delay caused by a simultaneous collision.

According to the request path of τ_0 and τ_1, a second interference could arrive in $L2$. However, the intrinsic nature of this interference is different. Let us imagine for instance that τ_0 reads a data from the bank B and put it in the $L2$ cache. As long as the data remains in the cache, each time τ_0 accesses it, its request path ends with $L2$. Let us imagine now that τ_1 reads another data from the bank B. According to the cache policy, data of τ_1 could evict data of τ_0 from $L2$. Consequently, the next time τ_0 would try to access its data, it should have to lengthen its request path

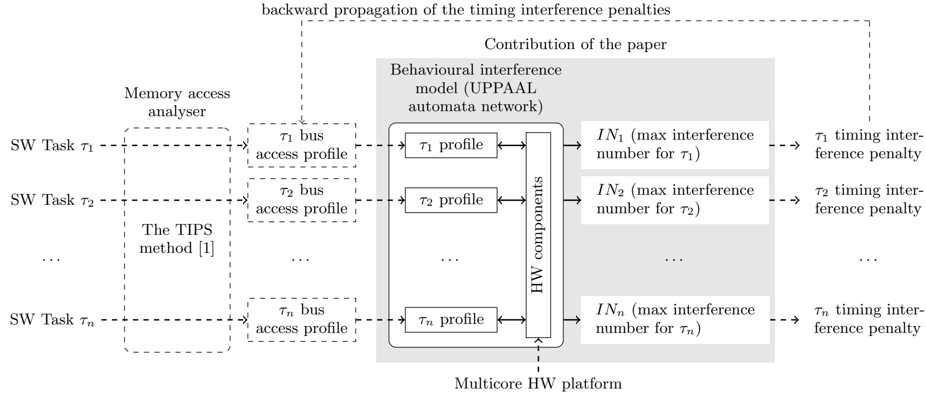


Figure 1: Approach overview (the gray box highlights the scope of the paper, while dashed lines are out of the scope)

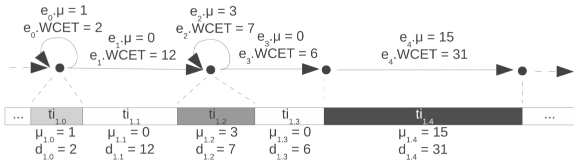


Figure 2: Example of bus access profile (excerpt from [1])

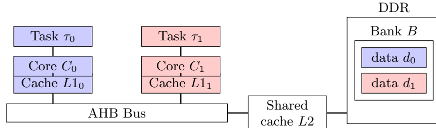


Figure 3: A simplified multi-core platform

Figure 4: Bus access profile of τ_0 and τ_1

to the memory. The effect is similar to the bus interference: τ_0 would suffer from a longer delay. However, the scenario of interference is different. There is no simultaneous collision. The two requests can occur at different time. In other words, τ_1 provokes a delayed interference on τ_0.

As shown in this example, in order to analyze the interferences that can occur, it is necessary to identify the types of HW components. We consider two types of components: transport and storage component.

Definition 1. A transport component is a component whose internal state only depends on the presence or absence of a request using it. If a request is using the component, then it is "occupied". Otherwise, it is "free".

Definition 2. A storage component is a component whose the internal state depends on previous requests (including the current one if any).

AHB Bus of Figure 3 is a transport component. Examples of "storage components" include caches and memory banks. The content of a cache depends on the previous memory requests that used it. It has a direct effect on the length of next memory request paths. Following the distinction, we refine the notion of interference.

Definition 3. An instantaneous interference occurs whenever at least two requests sent by two different tasks collide on the same transport component.

Definition 4. A delayed interference occurs whenever a request r sent by a task τ uses a storage component whose internal state has been made non-compliant with τ by another task τ'.

As explained below instantaneous interferences can occur in *AHB Bus* Figure 3. And a delayed interference can occur in $L2$. In the same way, a second delayed interference can occur in B. Indeed, a memory bank is composed of a row buffer acting as a local cache. It contains the last block of accessed data. Hence, requests to data in the row buffer (one talks about "row hit" requests) are faster than request to data not in the row ("row miss" requests). When a "row miss" occurs, the requested data has to be fetched in the row buffer, making the request time longer. As in cache, the content of the row buffer depends on previous requests. Hence, banks memory are storage components that can cause delayed interferences.

4. MODELING

To model the interferences, we define two classes of automata: automata for HW components, and automata for SW tasks.

4.1 HW components

The role of the HW component automata is to model the answers of the component to requests sent by the tasks. It determines if a request creates an interference in the component and notifies the software automaton of this result. As said in section 3, we consider two types of HW components: transport and storage components.

4.1.1 Transport component

According to definition 1 a transport component is modeled by the automaton Figure 5. It is composed of three locations (*Free*, *Occupied* and *Check*), and three internal data (*waiting_queue*, *nb_elmt*, and *bus_state*):

- *waiting_queue* is an internal list containing the identifiers of the tasks waiting for the component.
- *nb_elmt* is the number of tasks currently waiting for the component (including the task currently using it).

The three locations of this automaton are:

- *Free*: This location is the initial one. The component is free and waits for a request. When a request arrives the location changes to *Occupied*, and the function *update_bus* is called. This function updates the internal variables of the automaton to let it know which task is calling it.
- *Occupied*: The component is already occupied by a request. Once the request is completed the next location is *Check* if the internal waiting queue is not empty ($nb_elmt \neq 0$) and *Free* if the queue is empty ($nb_elmt = 0$). All outgoing transitions of the *Occupied* location are waiting for the synchronization event *end_req*. This event is sent by a task automaton when it releases the component after its request has been completed by the memory.
- *Check*: This location's purpose is to switch the task whose request is handled by the component. It is a transient location reached when the current task occupying the component is releasing it and when another one is waiting for the component. The role of location is to trigger the transition returning to *Occupied* to process the next request. Once again the *update_bus* function is called when returning to *Occupied* to change the internal variables of the component.

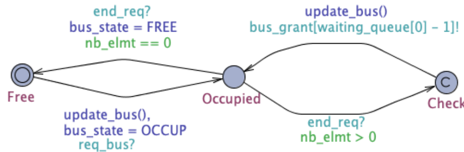


Figure 5: Automaton of a transport component

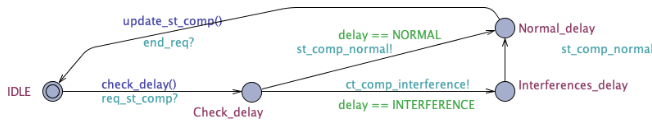


Figure 6: Automaton of a storage component

4.1.2 Storage component automata

Storage component generates delayed interferences, meaning that a task can interfere with another one by modifying the state of component. Storage components are modeled by the automaton Figure 6. The idea is to determine if the component reacts within a favorable delay (the normal case), or conversely within an unfavorable one (the interference case), when receiving a request. This notion of favorable delay is related to the task asking for the component. The automaton is composed of four locations:

- *IDLE* models the state where the component does not handle any request. It is waiting for a memory access request. When receiving it, it reaches *Check_delay* and it calls the function *check_delay* which determines whether there is an interference or not (i.e., whether the component is in an internal favorable state or not). *check_delay* compares the state of the component with the state of the component if the task is alone. Further explanations on this function are presented in the section 5.
- *Check_delay* is a transient location. Its role is to reach *Normal_delay* or *Interference_delay* according to the value of *delay*.
- *Normal_delay* models the normal response delay of request. Once the request is completed the component returns to *IDLE* and waits for another request. Before returning to the *IDLE*, the automaton is waiting for an occurrence of *end_req* sent by the task automaton processing the current request. When taking the transition to *IDLE*, the *update_st_comp* function is called to update the internal state variables of the component.
- *Interferences_delay*: When an interference occurs the response delay is extended. This location represents the added delay induced by the interference. Thereby the next location has to be *Normal_delay* because an interference is represented by an extra time added to the response delay.

4.2 Task automaton

A task automaton models the bus access profile of the task and the path of the requests through the HW components of the processor. Figure 7 gives the automaton of the task τ_0. It is composed of three parts. The first (locations *Request_bus*, *Result_bus*, and *Waiting_bus*) models the answer of the bus to the request. The second part (locations *Request_L2*, *Result_L2*, and *Waiting_L2*) models the answer of the L2 cache. And and the last part models the answer of the DDR memory.

The initial location of the automaton is an *Idle* location. It waits for a start (or a end) event from the scheduler (that is, the automaton implementing the sequence of the segment of the task profile). When receiving a start event, the local counter *nb_req* is set to zero. And the automaton reaches *Request_bus*. The three parts follow then the same pattern composed of three locations:

- *Request_cmp* (where *cmp* is *bus*, *L2*, or *DDR*) models the state in which the task has sent the request to the component *cmp* and is waiting to know if the request generates an interference. If there is an interference the next location is *Waiting_cmp*. If there isn't an interference the next location is *Results_cmp*.
- *Waiting_cmp* models the extra delay the task encounters due to the interference.
- *Results_cmp* models the case in which the request is normally completed (with or without interference). The

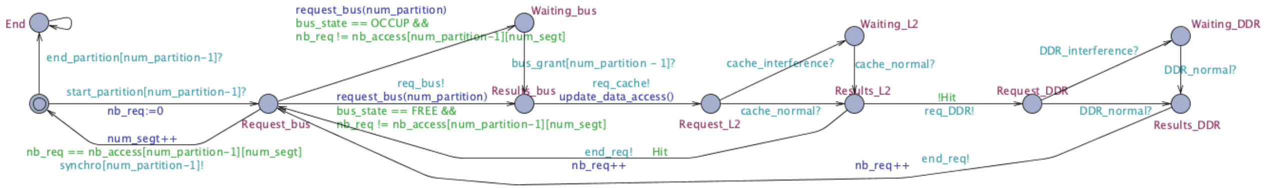


Figure 7: Automaton for task τ_0 of example Figure 3

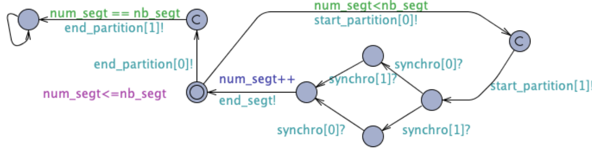


Figure 8: Scheduler automaton

next location is then the *Request_cmp* of the next component or *Idle* if the component is a storage component (L2 or DDR) and if the requested data is present in the component. When the outgoing transition of this location is taken, the task automaton sends an event to the component automaton to notify it of the end of the current request.

4.3 Scheduler automaton

The last automaton models the sequences of segments of the task profiles. For instance, the profiles of τ_0 and τ_1 shown in Figure 4 are modeled by the automaton in Figure 8. From the initial location, the Scheduler starts the first segment of τ_0 and τ_1. Then it waits for an occurrence of *synchro* sent by τ_0 and τ_1 at the end of their segment. When receiving them, the scheduler notifies the end of the first segment, and starts the second one. And so on until the last segment (when $num_segt == nb_segt$).

5. INTERFERENCE ANALYSIS

To determine when an interference occurs we instrument each component automaton with an algorithm to decide whether or not the task accessing the component is suffering from an interference. These algorithms are dependent on the type of the component under consideration.

Transport component. To count interferences in transport components is very simple: there is an interference *iff* the component is occupied and another request is waiting for it.

Storage component. For storage components, the decision algorithm is more complex. To determine if an access from a task X is affected by an access from a task Y, we not only maintain the "actual" state of the component but also the "virtual" state of the component for each task as if the task was isolated. Such a "virtual" view of the component is only paying attention to accesses from the task it is associated to. It means that an access from task X to the storage component affects both the "actual" view and its "virtual" view of the component, but not the "virtual" view associated to task Y. Thereby, for each access to a storage component we are able to determine if there is interference or not by comparing the virtual view of the task with the actual view of the storage component. If there is a difference between these two views it means that an interference occurred. This algorithm is implemented by the function *check_delay()* of each storage automaton. In the example of a L2 direct mapped cache with 16.384 lines of 32 bytes, our algorithm works on: (1) an array *cache_array* containing 16 384 lists of 32 bytes representing the current state of the L2 cache ; (2) one similar array *task_i_array* for each task i representing the virtual state of the cache if the task were alone. Let us consider two tasks. Imagine that task 1 requests a data stored in line 250. *cache_array[250]* and *task_1_array[250]* are updated. Then task 2 accesses the same line, *cache_array[250]* and *task_2_array[250]* are modified. When task 1 accesses again line 250, *task_1_array[250]* and *cache_array[250]* are different meaning that task 1 suffers from an interference.

Interference analysis by model-checking. To compute an upper bound of the number of interferences experienced by each task in each component for each segment, we use the UPPAAL model-checker. For instance, let us consider $nb_interf_L2[0,\tau_0]$ the number of interference in the first segment of τ_0 in cache L2 . UPPAAL shows that

$$\forall[](nb_interf_L2[0,\tau_0] \leq 3)$$

that is, 3 is an upper bound of $nb_interf_L2[0,\tau_0]$. This computation takes less than one second (UPPAAL 4.1.24 running on a 3,2 GHz Apple M1 Pro with 32 Go DDR5).

6. NEXT WORK

The next step is to conduct a set of experiments to validate the method. We plan to explore the GR740 processor: a quad-core processor based on SPARC V8 architecture. After this validation, the second work will consider finer placement and replacement policies in our cache model in order to capture more realistic configurations.

7. REFERENCES

[1] Thomas Carle and Hugues Cassé. Reducing timing interferences in real-time applications running on multicore architectures. In *18th International Workshop on Worst-Case Execution Time Analysis*, 2018.

[2] Certification Authorities Software Team. Multi-core Processors - Position Paper. Technical Report CAST 32-A, November 2016.

[3] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[4] Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019.

Data Access Time Estimation in Automotive LET Scheduling with Multi-core CPU

Risheng Xu
Kiel University
Mercedes-Benz AG
Kiel & Sindelfingen, Germany
risheng.xu@mercedes-benz.com
rxu@informatik.uni-kiel.de

Max J. Friese
Mercedes-Benz AG
Sindelfingen, Germany
max_jonas.friese@mercedes-benz.com

Hermann von Hasseln
Mercedes-Benz AG
Sindelfingen, Germany
hermann.v.hasseln@mercedes-benz.com

Dirk Nowotka
Kiel University
Kiel, Germany
dn@informatik.uni-kiel.de

ABSTRACT

Timing analysis is used to ensure that real-time software tasks are executed ahead of their deadlines. However, it continues to be a challenge due to the memory contentions in multi-core CPUs. We present a hybrid model that combines formal modelling and Monte-Carlo simulation to estimate the maximum observed execution time (MOET) of data access (read/write) for multi-core CPUs at the early stage of development. The model is applied to a genuine Mercedes-Benz powertrain software, and the derived results are compared to the actual measurements.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]; D.2 [**Software Engineering**]: Software Architectures; D.4 [**Operating System**]: Reliability

Keywords

Automotive, LET Scheduling, Multi-core, Timing Analysis

1. INTRODUCTION

The automotive industry has adopted the Logical-Execution-Time (LET) paradigm [12] as an approach to achieve better time and value determinism in multi-core CPUs [11]. For safe execution, the length of a LET time frame should at least cover the maximum execution time of its tasks. In the automotive V-model development process, we mainly rely on two measurements to get the software execution time:

- Processor-in-the-Loop (PiL) Measurement: The time measurement of a single function under the unit-test environment in one CPU core.
- Hardware-in-the-Loop (HiL) Measurement: The time measurement of the integrated binary file in multiple CPU cores.

However, HiL is not available until the late stages of the V-model (after software integration). Any failed time requirements here may result in costly task re-scheduling. Although the PiL is available before integration, the results may underestimate the data access time. Due to the fact that the PiL is performed on a single CPU core, the contentions between cores are omitted.

On the other hand, the task scheduling starts in the early stage of the development, even before the PiL. To determine the maximum task execution time, i.e. the shortest LET frame length, the current workaround is to use the HiL measurement from the previous version of the software, plus a margin factor, as the new maximum in the current version. However, there is no assurance that this upper bound is sufficient for the coming software updates. Thus, a timing model is needed, especially for data access time, to help the scheduler make sure that the LET frames are reliable.

In this paper, we focus on the Maximum-Observed-Execution-Time (MOET) of data accesses in an early stage (end-to-end requirement phase). We use MOET rather than Worst-Case-Execution-Time (WCET) for the following reasons: First, the early-stage WCETs are often over-pessimistic. With the given WCETs, it is extremely difficult to schedule the tasks. Second, we concede that the MOET is not the most secure time upper bound. However, this is sufficient if we require an intuitive evaluation in the early stage. The reliability of the complete software is still guaranteed by the HiL-based analysis after software integration. Thus, we employ the Monte-Carlo method and select the maximum execution time across multiple simulations as MOET, acting as the key performance indicator.

2. RELATED WORK

The authors in [19] give a comprehensive survey of timing techniques in multi-core CPU, and we concentrate on the most closely related works in this section.

The LET paradigm was first proposed in [12] as a programming model. The implementation of LET for automotive software is introduced by [18].

Many timing analyses are published for LET scheduling. In [3] the authors compare different LET variants in AUTOSAR environment. In [20] the authors analyse the end-to-end latency in LET scheduling. In [10] a formal analyses for timing compositionality in LET scheduling is given. However, the papers cited above have not compared the results of a WCET/MOET models with time measurement,

particularly in the context of a real-world project.

While the above LET analysis mainly focuses on software tasks, the hardware modelling offers a complementary perspective. In [5] the authors present a general WCET model for bus contention in multi-core CPUs. In [7], [15], [16] and [2] the authors analyse the memory and pipeline behavior of the Aurix platform. As we use the same platform, we refer to some hardware parameters from those preceding papers.

3. SYSTEM MODEL

3.1 LET Model

The LET model implemented in [18] and [23] aims to ensure the tasks are executed in a deterministic order. The implementation is shown in Figure 1, a LET task consists of three phases: read-in phase, execution phase and write-out phase. In the read-in and write-out phases, the local data copy is synchronized with the data in global memory.

The read-in and write-out phase includes not only the time required by data access but also other time parameters as below:

- r_b, w_b, r_e and w_e: the delay before and after signal access. We consider they are pre-measured constant in our model.
- r_s and w_s: the synchronization delay between read-in and write-out. The details are modelled in Section 3.2.
- r_i and w_i: the delay caused by the interrupt from the tasks with higher priorities. Note the starting point of r_i and w_i is flexible. They may happen before, after or during data access. The length ranges between 0 and a pre-measured upper bound.
- r_{cet} and w_{cet}: the core execution time (CET) of signal read-in and write-out. These two parameters are the main focus of this paper.

Depending on the position and the length of r_s and r_i, the r_{cet} shifts between the end of r_b and the start of r_e. We define this interval as possible range r_{pr} for r_{cet}. Similarly, w_{pr} is defined for w_{cet}. If we use the notation $max()$ to present the maximum length of a time interval, formally we can define

$$r_{pr} = max(r_s) + max(r_i) + max(r_{cet}) \quad (1)$$

$$w_{pr} = max(w_s) + max(w_i) + max(w_{cet}) \quad (2)$$

If the r_{pr} or w_{pr} overlaps with another r_{pr} or w_{pr}, in worst case their w_{cet} or r_{cet} overlap with each other as well. This leads to co-access requests to data and brings extra delays. According to the type of overlap, we define 4 kinds of neighbour for a given target job. First, `Read-Read Neighbour (RRN)`: r_{pr} of the target job overlaps with the r_{pr} of the neighbour; Second, `Write-Write Neighbour (WWN)`: w_{pr} of the target job overlaps with the w_{pr} of the neighbour; Third, `Read-Write Neighbour (RWN)`: r_{pr} of the target job overlaps with the w_{pr} of the neighbour; Fourth, `Write-Read Neighbour (WRN)`: w_{pr} of the target job overlaps with the r_{pr} of the neighbour.

The neighbour relationship defines all possible co-access jobs. The first two neighbours may introduce extra contention delays and affect the length of r_{cet} and w_{cet}, while the other two neighbours require extra synchronization delay r_s and w_s to guarantee the data consistency.

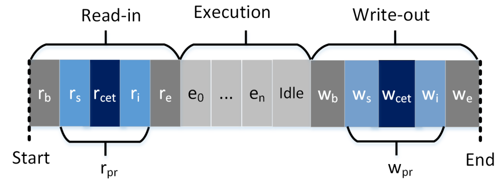


Figure 1: LET Model with 3 Phases

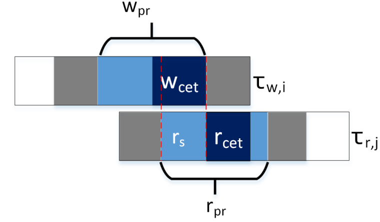


Figure 2: Example of Synchronization Delay

3.2 Synchronization Delay Model

To ensure data consistency, we require all the read-in operations to wait until all the executing write-out operations finish, and vice versa. The wait time is defined as synchronization delay r_s and w_s.

As shown in Figure 2, consider two LET job $\tau_{w,i}$ and $\tau_{r,j}$ while the w_{pr} from $\tau_{w,i}$ overlaps with the r_{pr} from $\tau_{r,j}$. The r_s of $\tau_{r,j}$ is determined by two factors: The length of the overlap area, and the length of w_{cet}. The maximum r_s of job $\tau_{r,j}$ is achieved when the w_{cet} interval of $\tau_{w,i}$ complete shifts to the overlap area, and the length of r_s is then the shorter one between the length of w_{cet} and the overlap area.

Now, we consider the job to have multiple neighbours. In practice, the length of a job is much longer than any of the possible range w_{pr} or r_{pr}, and the LET jobs in the same core are scheduled without overlapping. Thus, each w_{pr} overlap at maximum with one r_{pr} per core, and vice versa. In other words, each job has no more than one read-write and one write-read neighbour within one core. We can simply add up the maximum synchronization delay caused by each neighbour and use this sum as the upper bound of job's synchronization delay. Given a job $\tau_{m,n}$

$$r_s(\tau_{m,n}) \leq \sum_{o \in RWN(\tau_{m,n})} min(\, w_{pr}(o) \cap r_{pr}(\tau_{m,n}),\, w_{cet}(o)) \quad (3)$$

$$w_s(\tau_{m,n}) \leq \sum_{o \in WRN(\tau_{m,n})} min(\, r_{pr}(o) \cap w_{pr}(\tau_{m,n}),\, r_{cet}(o)) \quad (4)$$

3.3 Access Time Simulation

Data access is managed on the Aurix platform via the SRI bus. When a job attempts to access the bus while it is already serving another job, a memory contention occurs. In Section 3.2, we mandate read-write/write-read neighbour to synchronize with the target job. As a result, the contention occurs only between read-read/write-write neighbours and the target job. We consider the contention only happens in data access, the contention-free instruction fetch is achieved

by either banked program flash or local program memory. The data cache is forbidden because execution determinism is required.

First, the data access in the LET job is accomplished through a series of copy operations which are similar to `memcpy()` as in [21]. The instructions mainly consist of two parts: first is the load (`ld`) and store (`st`) instructions. Depending on the copy direction, one of these two instructions requires access to global memory. This instruction generates a SRI bus transaction [13], where the memory contention may occur. The remaining instructions are used to decode the memory address and are contention free. According to the statement in [4], the Aurix pipeline avoids the domino effect of time anomaly described in [22] and [17], and the core is time-compositional under a constant upper bound. Given the various pipeline states encountered during execution, it is reasonable to assume that the execution time d of these contention-free instructions is within a pre-defined time interval $d \in [d_{min},\ d_{max}]$. Due to the limit of measure, we are unable to obtain the distribution of d in the given interval. Therefore, we assume that d has a uniform distribution. This is the input random variable for our Monte-Carlo simulation.

Second, we generate two sequences for a job and one of its neighbours. As shown in Figure 3. The white block indicates that there are no contention in this cycle, whereas the black block may have contentions. More precisely

1. We insert a low-level bus transaction after the load and store instruction as black blocks. The required stall cycle is shown in [14].

2. One bus transaction contains maximum 256 bits data (block transfer mode) [14], Thus the non-atomic signals (e.g., array, structure) are split into multiple transactions.

3. A random number of white blocks $d \in [d_{min},\ d_{max}]$ are inserted between two bus transactions to indicates the execution time of those contention-free instructions. A constant number of white blocks d_{const} are appended to the end of the sequence as the constant cost for the data access function.

Thirdly, we compare the sequences for the target and neighbouring jobs. As illustrated in Figure 3, once two transaction appear in the same cycle, a memory contention is triggered. According to the hardware specification in [14], one must wait until the other finishes.[1]. We assume the target job always wait for its neighbour, to avoid underestimating its contention delay. Consequently, several black blocks are inserted to the target job as stall cycles. Repeat the comparison for the target job and all related neighbours, we now have the signal access cycles together with contention delay. The simulation from Step 1 to Step 3 is executed multiple times and the maximum value is saved as a Maximum-Observed-Execution-Time (MOET).

3.4 Algorithm

In previous sections, we have discussed the synchronization delay and the access time independently. However, these two time variables are highly correlated, any changes in one may effect the other. To address this issue, we propose a recursive algorithm that improves the precision of the data access time r_{cet} and w_{cet} for a given task τ_i during multiple recursions.

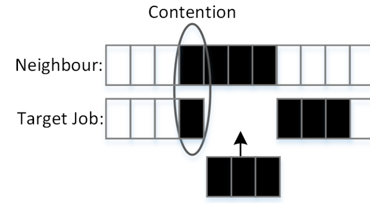


Figure 3: Example of the Memory Contention in Two Sequences

1. Generate all the task jobs within the specified time period (e.g, the hyper-period) of all tasks. Set the maximum value for the possible range r_{pr} and w_{pr}. In this case, we set the r_{pr} to the length of the job, and the w_{pr} to the length of the write-out phase with a pre-defined t^{w}_{init}.

2. Identify four kinds of neighbours for all task jobs by considering the overlaps of all r_{pr} and w_{pr}.

3. Simulate the maximum access time of w_{cet} and r_{cet} based on identified neighbours.

4. Calculate the maximum time of synchronization delay r_s and w_s based on r_{pr}, w_{pr}, and the maximum value of w_{cet} and r_{cet}.

5. Update the r_{pr} and w_{pr} using the maximum value of r_s, w_s, w_{cet} and r_{cet} from last two steps.

6. Repeat the Step 2 to Step 5 until either the result is convergent or an error occurs, e.g., the write-out phase exceeds the LET job.

4. CASE STUDY

4.1 Setup

We evaluate the data access time of a genuine Mercedes-Benz powertrain software in mass production. There are thirty periodic LET tasks evaluated (e.g., energy management, thermal management). The task periods vary from $2\,ms$ to $1000\,ms$ and the frame lengths are between $0.1\,ms$ to $4\,ms$. The detailed parameters and source code are available online.[2] The modeled hardware is Aurix TC397, and the software is allocated in domain 0 (CPU core 0-3). The measurements are collected from HiL station [8] with Gliwa tool [24]. We use CPU cycle as time unit in both model and measurement. As the CPU main frequency is $300\,MHz$, $1\,\mu s = 300\,cycle$.

The algorithm is iterated three times in two hours before a convergent result is observed. The Monte-Carlo simulation is repeated 100 times for all overlapped jobs in every iteration. In a schedule interval of 2 seconds, more than 2×10^5 times of simulations are executed. The repetition is based on two principles: First, the repetition must be sufficient for

[1]we configure all the cores to have the same SRI priority

[2]https://gitlab.com/xrisheng/DAA

a clear distribution as Figure 6. Second, the model execution time should not impede the agile software development process, eight hours or less is generally acceptable.

4.2 Result

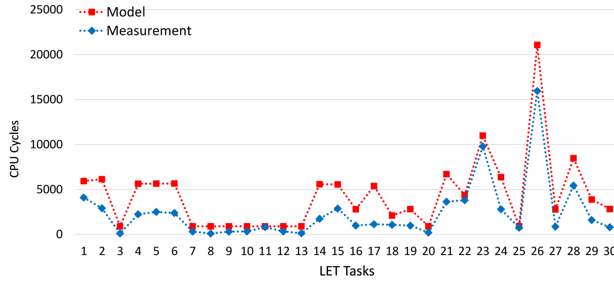


Figure 4: Maximum Read-in Time per Task

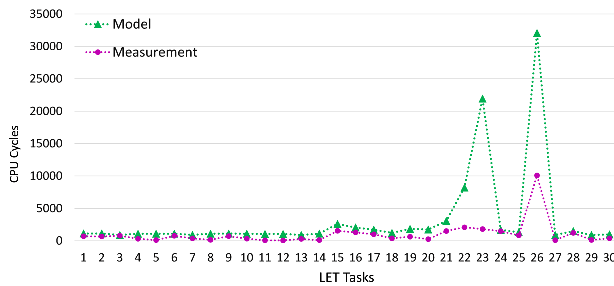


Figure 5: Maximum Write-out Time per Task

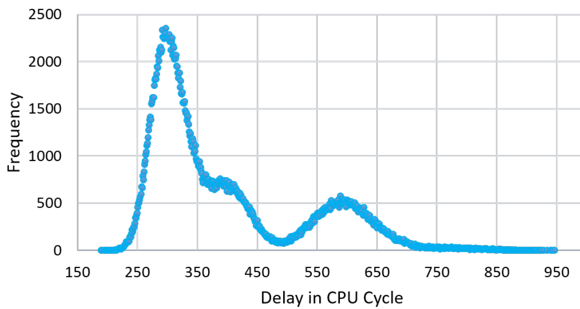


Figure 6: Contention Delay Distribution for LET Task 1 Read-in

As illustrated in Figure 4 and 5, we compare the maximum access time predicted by our model to the measurement for each task, the detailed data is listed in Appendix. In both Figures the modelled values are either equal to or higher than the measured values. The primary error (e.g., Task 22, 23 and 26 in Figure 5) is caused by an incomplete overview of signal flow and task scheduling, which are under the IP protection of the software supplier. In such case, we have to overestimate the number of signals accessed.

The contention delay distribution of the read phase from LET Task 1 is shown Figure 6 as an example. This distribution is created by superimposing multiple normal distributions. The reason for this is that the jobs of a LET task have different patterns of overlap in the simulated time period. Each pattern contributes a delay distribution based on the preconditions specified in Section 3.3, and the task's total distribution is then the sum of the pattern's distributions. The scheduler may use the task's distribution to calculate confidence intervals and adjust the length of the LET frame by an extra margin factor.

Additionally, we attempt to evaluate the powertrain software using existing WCET tools aiT [9] and OTAWA [1], both of which are based on static analysis. The OTAWA does not currently support the TC397 platform, so we gave up after evaluating the development effort. The aiT suffers from state explosion by default analysis. Therefore, extra annotation must be added to improve the performance (e.g., the number of loop cycles). The annotation requires a comprehensive understanding of the source code. However, our powertrain software is composed of the binary libraries delivered from the supplier and machine-generated C code. Both are difficult to parse. As a consequence, the existing WCET tools are inapplicable to our use case.

5. CONCLUSION AND FUTURE WORK

In this paper, we focus on the data access time of LET scheduling in Infineon Aurix platform. A hybrid model consists of formal modelling and Monte-Carlo simulation is used. We apply the presented model to a real Mercedes-Benz powertrain software and compare the results to those obtained using HiL measurements.

For the future work, we plan to use an address-based tracking tool (e.g., CEDAR [25]) to observe the instruction-level execution time distribution of data access operations (e.g., the variable d discussed in section 3.3). A measurement-based probabilistic model [6] may help to improve the precision.

6. REFERENCES

[1] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

[2] B. Binder, M. Asavoae, F. Brandner, B. Ben Hedia, and M. Jan. Formal modeling and verification for amplification timing anomalies in the superscalar tricore architecture. *International Journal on Software Tools for Technology Transfer*, pages 1–26, 2022.

[3] 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)*, pages 240–250. IEEE, 2018.

[4] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems. *Proceedings of Embedded Real Time Software and Systems*, 36:42, 2010.

[5] D. Dasari and V. Nelis. An analysis of the impact of bus contention on the wcet in multicores. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 1450–1457. IEEE, 2012.

[6] R. I. Davis and L. Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *LITES: Leibniz Transactions on Embedded Systems*, pages 1–60, 2019.

[7] E. Díaz, E. Mezzetti, L. Kosmidis, J. Abella, and F. J. Cazorla. Modelling multicore contention on the aurix tm tc27x. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[8] dSPACE GmbH. Scalexio system.

[9] C. Ferdinand and R. Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004.

[10] K.-B. Gemlau, J. Schlatow, M. Möstl, and R. Ernst. Compositional analysis of the waters industrial challenge 2017. 2017.

[11] J. Hennig, H. von Hasseln, H. Mohammad, S. Resmerita, S. Lukesch, and A. Naderlinger. Towards parallelizing legacy embedded control software using the let programming paradigm. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–1. IEEE Computer Soc., 2016.

[12] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *International Workshop on Embedded Software*, pages 166–184. Springer, 2001.

[13] "Infineon Technologies AG". *"Infineon AURIX TC27x User Manual D-Step"*, 12 2014. "Rev. 2.2".

[14] "Infineon Technologies AG". *"Infineon AURIX TC3xx User Manual Part 1"*, 08 2020. "Rev. 1.6".

[15] P. Jungklass and M. Berekovic. Effects of concurrent access to embedded multicore microcontrollers with hard real-time demands. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–9. IEEE, 2018.

[16] L. Kosmidis, D. Compagnin, D. Morales, E. Mezzetti, E. Quinones, J. Abella Ferrer, T. Vardanega, and F. J. Cazorla Almeida. Measurement-based timing analysis of the aurix caches. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, pages 9–1. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2016.

[17] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2002.

[18] R. Mader. Implementation of logical execution time in an autosar based embedded automotive multi-core application. In Dagstuhl Seminar 18092, 2018.

[19] 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)*, 52(3):1–38, 2019.

[20] J. Martinez, I. Sañudo, and M. Bertogna. Analytical characterization of end-to-end communication delays with logical execution time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2244–2254, 2018.

[21] A. S. Peter Gliwa, Dr. Nicholas Merriam. Best practice for timing optimization. Embedded Software Engineering Congress 2018, 2018.

[22] J. Reineke and R. Sen. Sound and efficient wcet analysis in the presence of timing anomalies. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[23] J. M. Rivas, J. J. Gutiérrez, J. L. Medina, and M. G. Harbour. Comparison of memory access strategies in multi-core platforms using mast. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2017.

[24] O. Scheickl, C. Ainhauser, and P. Gliwa. Tool support for seamless system development based on autosar timing extensions. In *Embedded Real Time Software and Systems (ERTS2012)*, 2012.

[25] A. Weiss, S. Gautham, A. V. Jayakumar, C. R. Elks, D. R. Kuhn, R. N. Kacker, and T. B. Preusser. Understanding and fixing complex faults in embedded cyberphysical systems. *Computer*, 54(1):49–60, 2021.

Joint Scheduling, Routing and Gateway Designation in Real-Time TSCH Networks

Miguel Gutiérrez Gaitán*
mgg@fe.up.pt
CISTER Research Centre
University of Porto
Porto, Portugal

Luís Almeida
lda@fe.up.pt
CISTER Research Centre
University of Porto
Porto, Portugal

Thomas Watteyne
thomas.watteyne@inria.fr
AIO Team
Inria
Paris, France

Pedro M. d'Orey
ore@isep.ipp.pt
CISTER Research Centre
University of Porto
Porto, Portugal

Pedro M. Santos
pss@isep.ipp.pt
CISTER Research Centre
Polytechnic Institute of Porto
Porto, Portugal

Diego Dujovne
diego.dujovne@mail.udp.cl
Esc. de Inf. y Tel.
Diego Portales University
Santiago, Chile

ABSTRACT

This research proposes a co-design framework for *scheduling*, *routing* and *gateway designation* to improve the real-time performance of low-power wireless mesh networks. We target time-synchronized channel hopping (TSCH) networks with centralized network management and a single gateway. The end goal is to exploit existing trade-offs between the three dimensions to enhance traffic schedulability at systems' design time. The framework we propose considers a global Earliest-Deadline-First (EDF) scheduler that operates in conjunction with the minimal-overlap (MO) shortest-path routing, after a *centrality-driven* gateway designation is concluded. Simulation results over varying settings suggest our approach can lead to optimal or near-optimal real-time network performance, with 3 times more schedulable flows than a naive real-time configuration.

Keywords

Centrality, Network design, Low-power wireless mesh networks, TSCH.

1. INTRODUCTION

Wireless networks are at the heart of Industry 4.0 and the Industrial Internet of Things (IIoT) [11], offering more flexibility and scalability than their wired counterparts. Time-synchronized channel-hopping (TSCH) is widely regarded as the de-facto low-power wireless networking approach for demanding industrial applications, achieving ultra low-power and wire-like reliability [2]. Its core features are time-division multiple-access (TDMA) and frequency diversity, making it ideal for real-time communication, and therefore often applied to real-time monitoring and process control [9].

Theoretical and empirical studies have analyzed the predictable and, thus, analyzable aspects of TSCH [6, 9]. These works typically focus on prioritized packet scheduling [10] and routing methods [4] for improved real-time network performance. Our previous work looks at gateway designation [3], the convenient positioning/selection of the gateway within a given topology, for enhanced traffic schedulability.

*Corresponding author.

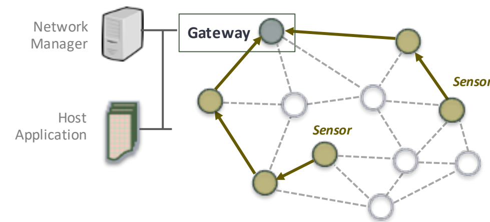


Figure 1: A low-power wireless mesh network.

In this work, we simultaneously deal with network design, routing and resource allocation to provide a joint configuration framework for improved real-time network performance in TSCH networks. We first look at joint minimal-overlap (MO) real-time routing and Earliest-Deadline-First (EDF) scheduling [4] then combine it with centrality-driven gateway designation [3]. The goal is to complement the benefits of the three approaches to further enhance the real-time properties of the network at the system design time. To the best of our knowledge, this paper proposes the first joint scheduling, routing and gateway designation framework for real-time TSCH-based networks.

2. PRELIMINARIES

We target low-power wireless sensor networks (WSNs) focused on industrial application in the broad sense (from smart farming to automotive). A typical network (see Fig. 1) is composed of several sensing nodes which have limited computation capabilities and energy. These nodes are interconnected wirelessly to more powerful networking equipment (e.g. access points) capable of hosting a gateway and/or a network manager. A gateway node is used to establish the connection between the sensor nodes and the host application. The specific application dictates the data collection process, i.e. the sampling rate at each sensor node.

2.1 Network Model

We assume TSCH as the underlying medium access control (MAC) layer adopted to build up highly reliable and low-power networks. TSCH supports multi-hop and multi-channel communication over a globally synchronized TDMA scheme. Transmissions take place inside time slots of fixed duration allocated over up to $m = 16$ channels. A time slot allows transmitting a single packet and receiving its corresponding acknowledgement. A channel-hopping mechanism is used to improve reliability against multi-path fading and external interference. A global scheduler defines the channel and the time slot used by the different links. In this paper, we use a global EDF scheduling policy [8].

We model the long-term connectivity between the nodes by a unidirected graph $G = (V, E)$, where V is the set of vertices (or nodes), E the set of edges (or links) between those nodes. The number of vertices in G is denoted by $|V_G|$; the number of edges, $|E_G|$.

2.2 Flow Model

The global traffic flow pattern is assumed as convergecast, directed toward a single gateway. Packets are generated by a subset of n sensor nodes $\in V$; the remaining $|V_G| - n - 1$ nodes act solely as relays. Sensor nodes also relay packets. Each sensor node transmits a periodic (and deadline-constrained) data flow over a single route. We call $F = \{f_1, f_2, ..., f_n\}$ the resulting set of n real-time network flows. Each flow is characterized by a 4-parameter tuple $f_i = (C_i, D_i, T_i, \phi_i)$. C_i is the effective transmission time between the source node s_i and the gateway, D_i is the deadline, T_i is the period, ϕ_i is the multi-hop routing path. Note C_i does not consider interference from other flows. We assume a flow never stops, i.e. new packets are always generated. The γth packet in flow i is denoted $f_{i,\gamma}$; it is generated at time $r_{i,\gamma}$ such that $r_{i,\gamma+1} - r_{i,\gamma} = T_i$. In accordance with EDF, $f_{i,\gamma}$ needs to reach the gateway before its absolute deadline $[d_{i,\gamma} = r_{i,\gamma} + D_i]$.

2.3 Performance Model

We consider the supply/demand-bound based schedulability test proposed in [7] to quantify the real-time performance of TSCH-based networks under EDF [5]. This method evaluates if the *supply-bound function* (sbf) – the minimal transmission capacity offered by a network with m channels – is equal or larger than the *forced-forward demand-bound function* [1] (FF-DBF-WSN) – the upper bound on the total network time demanded by a set of n time-sensitive flows in any interval of length ℓ.

Formally, the schedulability test can be presented by (1).

$$\text{FF-DBF-WSN}(\ell) \leq \text{sbf}(\ell),\ \forall \ell \geq 0. \qquad (1)$$

Where $\text{sbf}(\ell)$ is a piecewise-linear function in all intervals $[h, h + l]$ that satisfies (2).

$$\text{sbf}(0) = 0 \wedge \text{sbf}(\ell + h) - \text{sbf}(\ell) \leq m \times h, \forall \ell, h \geq 0; \qquad (2)$$

FF-DBF-WSN [7] is composed of two main terms: *(i)* **channel contention** due to mutually exclusive scheduling on multiple channels, equivalent to FF-DBF for multiprocessors [1], and *(ii)* **transmission conflicts** due to multiple flows encountering on a common half-duplex link.

$$\text{FF-DBF-WSN}(\ell) = \overbrace{\frac{1}{m}\sum_{i=1}^{n} \text{FF-DBF}(f_i, \ell)}^{\text{CHANNEL CONTENTION}} + \underbrace{\sum_{i,j=1}^{n}\left(\Delta_{i,j} \cdot \max\left\{\left\lceil \frac{\ell}{T_i}\right\rceil, \left\lceil \frac{\ell}{T_j}\right\rceil\right\}\right)}_{\text{TRANSMISSION CONFLICTS}} \qquad (3)$$

This results in (3), where $\Delta_{i,j}$ is a factor representing the path overlapping degree between any pair of flows f_i and f_j $\in F$ (with $i \neq j$) in a given network G, defined by (4).

$$\Delta_{i,j} = \sum_{q=1}^{\delta(ij)} Len_q(ij) - \sum_{q'=1}^{\delta'(ij)} (Len_{q'}(ij) - 3) \qquad (4)$$

$\delta(ij)$ is the total number of overlaps between f_i and f_j of which $\delta'(ij)$ are the ones larger than 3. The length of the qth and q'^{th} path overlap between f_i and f_j are called $Len_q(ij)$ and $Len_{q'}(ij)$, respectively, with $q \in [1, \delta(ij)]$ and $q' \in [1, \delta'(ij)]$. In convergecast, the factor expression is simpler since all paths are directed to the same root: only one path of arbitrary length is shared between any pair of flows. This implies $\Delta(ij) = 3$ for overlap paths larger than 3 hops.

3. A REAL-TIME TSCH FRAMEWORK

We consider the problem of co-designing the communication schedule, the routing topology and identifying the gateway to improve traffic schedulability. We build upon our prior research: the insights on joint EDF-MO scheduling and routing [4] and the centrality-driven network designation strategy [3]. While these works have already demonstrated – separately – their benefits, we show in this paper that combining the featured EDF-MO real-time scheduling and routing method with a judicious centrality-based gateway designation increases schedulability by up to 80% with respect to a naive real-time configuration.

3.1 Joint EDF-MO Scheduling and Routing

Minimal-overlap (MO) shortest-path routing is a greedy meta-heuristic search to find a suitable set of flow's paths that reduces the *overall path overlapping degree* in the network [4]. The *overlaps* – the set of nodes shared between two different flow paths – have a direct influence on the analysis of worst-case end-to-end delays for TSCH-based networks [10]. This, in turn, can be translated into an impact on network schedulability under a global EDF policy [12, 4]. The joint EDF-MO configuration takes advantage of this inbred network relationship to provide a set of disjoint paths which minimizes the number of overlaps among flow paths, regardless of the node designated as a gateway.

Algorithm 1 presents a pseudo-code of the MO routing based on its theoretical definitions in [4]. The algorithm consists of three major procedures: EdgeUpdate (lines 1-4), CalcOverlaps (lines 5-7) and the main method MoGH (lines 8-21). The latter determines a new set of flow paths Φ_k and its corresponding overall number of overlaps Ω_k at each kth iteration to find the set of paths that provides minimal overlapping. This procedure stops when $\Omega_k = 0$, or after a k_{max} number of iterations. EdgeUpdate updates the weights of the link for the input topology G_{in}, returning

Algorithm 1 Minimal-Overlap (MO) Routing

Input: G, F, k_{max}, ψ
Output: Φ_{opt}, Ω_{opt}

1: **procedure** EdgeUpdate(G_{in}, Φ_{in})
2: $W^k_{i,j}(u,v) = 1 + \sum_{e=1}^{\delta_{i,j}^{(k-1)}} \psi$
3: $G^{out} \leftarrow G_{in}(V, E^{weighted})$
4: **return** G^{out}
5: **procedure** CalcOverlaps(Φ_{in})
6: $\Omega_k = \sum_{i,j}^{n} \Delta_{ij}$
7: **return** Ω^{out}
8: **procedure** MoGH(G, F, k_{max})
9: $k = 0$ ▷ Initial Solution Start
10: $\Phi_0 \leftarrow$ ShortestPath(G, F) ▷ Hop-count-based
11: $G^0 \leftarrow G$
12: **while** $k \leq k_{max}$ and $\Omega_k^{min} > 0$ **do** ▷ Greedy Search
13: $G^k \leftarrow$ EdgeUpdate(G^{k-1}, Φ_{k-1})
14: $\Phi_k \leftarrow$ ShortestPath(G^k, F) ▷ Weight-based
15: $\Omega_k =$ CalcOverlaps(Φ_k)
16: **if** $\Omega_k < \Omega_{min}$ **then**
17: $\Omega_k^{min} = \Omega_k$
18: $\Phi_k^{min} = \Phi_k$
19: **else**
20: $\Omega_k^{min} = \Omega_k^{min}$
21: $k = k + 1$
22: $\Phi_{opt} = \Phi_k^{min}$, $\Omega_{opt} = \Omega_k^{min}$. ▷ Best Solution

a new weighted graph G_{out} over which new paths and overlaps are calculated. Cost function $W_{i,j}(u,v)$ determines the weight of an edge (u,v) in G_{out} as function of $\delta_{i,j}$ and ψ. The former is the number of overlaps between the paths of flows f_i and $f_j \in F$ at graph G_{in}; the latter a user-defined parameter used to control the speed of convergence of the algorithm. The ShortestPath procedure provides the shortest sequence of edges between two nodes in the graph, resorting to classical weighted or hop-count-based shortest-path mechanisms (e.g. Dijkstra). The CalcOverlaps procedure returns the total number of overlaps in the network by summing every $\Delta_{i,j}$ factor (as defined in Section 2), and which represents the overlapping degree experienced by the paths of any pair of flows f_i and $f_j \in F$.

3.2 Centrality-Driven Gateway Designation

To further enhance network schedulability, we consider the centrality-driven network designation strategy proposed in [3]. We use *network centrality* from graph theory to provide convenient graph-based positions to designate the gateway in order to improve real-time network performance, by design. Specifically, it uses the four most common centrality metrics in social network analysis: degree, betweenness, closeness and eigenvector centrality, considered as near optimally correlated for the purposes of benchmarking.

Table 1 [1] formally summarizes these four metrics.

[1]**Notation. DC**: $degree(v_q)$ is the number of edges of node v_q directly connected to any of the rest $N-1$ nodes in the graph G. **BC**: $sp_{r,s}$ is the number of shortest paths between any pair of vertices v_r and v_s, and $sp_{r,s}(v_q)$ is the number of those paths passing through node v_q; **CC**: $distance(v_p, v_q)$ is the shortest-path (hop-count) distance between vertices v_p and v_q, with $p \neq q$, $\forall\, v_p \in V$. **EC**: $\lambda_{max}(A)$ is the largest eigenvalue of the adjacency matrix $A = [a_{j,q}]_N$, where $a_{j,q}$ is the matrix element at row j and column q, and x_j is the jth value of the eigenvector x of graph G.

Table 1: Network Centrality Metrics.

Metric	Definition
Degree	$DC(v_q) = \frac{degree(v_q)}{N-1}$
Betweenness	$BC(v_q) = \sum_{q \neq r} \frac{sp_{r,s}(v_q)}{sp_{r,s}}$
Closeness	$CC(v_q) = \frac{1}{\sum_{p \neq q} distance(v_p, v_q)}$
Eigenvector	$EC(v_q) = \frac{1}{\lambda_{max}(A)} \cdot \sum_{j=1}^{N} a_{j,q} \cdot x_j$

4. PERFORMANCE EVALUATION

4.1 Simulation Setup

Wireless network. We consider a set of 100 mesh topologies generated from synthetic graphs. Each topology is created using a sparse uniformly distributed random matrix of $N \times N$ of zeros and ones, with target density d. N represents the total number of nodes, including the gateway; d is the portion of other nodes each vertex is linked to. We assume $N = 75$ and $d = 0.10$ for all topologies. We assume the network is TSCH-based with $m = 16$ channels available, and 10 ms time slots.

Network flows. A subset of $n \in [1, 25]$ nodes is chosen randomly as sensor nodes which transmit periodically deadline-constrained data toward a single gateway. The rest of nodes act as relay. Each period is randomly generated as 2^η, with $\eta \in \mathbb{N} \in [2, 7]$ slots. This implies a super-frame length of $H = 1280$ ms. C_i is computed directly from the number of hops in ϕ_i and $D_i = T_i$.

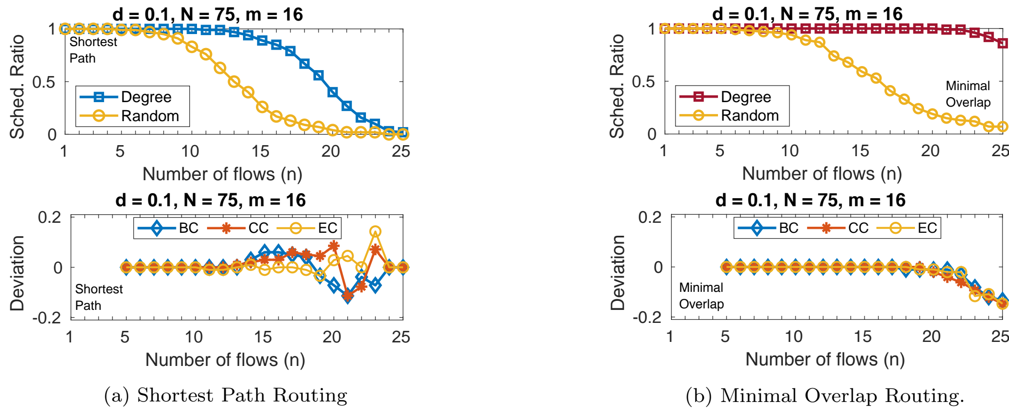
Real-time performance assessment. We consider the performance model described in Section 2. We evaluate the schedulability over an interval equal to the super-frame length, $\ell = H$, when all the channels are available. We further assume network management is centralized, scheduling uses a global EDF policy, and routing can be either a hop-count-based shortest-path routing (Dijkstra) or the featured MO routing described in Algorithm 1. For MO, we further consider the following: $\Psi = 0.1$ and $k_{max} = 100$.

4.2 Preliminary Results & Discussion

Fig. 2a (top) presents the schedulability ratio when a gateway is designated based on the degree centrality and a shortest path routing is assumed. Fig. 2b (top) shows equivalent results for a gateway designation based on the DC when the MO routing is considered. Both configurations also include the case when the gateway is designated randomly. The results show that a joint EDF-MO-DC framework can schedule up to 3 times more flows than a basic routing configuration, and up to twice more than the EDF-MO tuple. Notably, achieving up to 80% better schedulability than a naive real-time setting.

Figs. 2a (bottom) and 2b (bottom) presents the absolute deviation in terms of schedulability ratio of the other centrality metrics w.r.t. the DC. These results suggest none of the metrics dominates over the others, regardless of the routing used. We also observe the deviation among the metrics remains larger (up to $\sim 20\%$) for the shortest path routing, and almost marginal ($< 3\%$) when MO is used.

Overall, the reported results highlight the relevance of applying a judicious gateway designation in real-time TSCH



(a) Shortest Path Routing

(b) Minimal Overlap Routing.

Figure 2: **Top**: The schedulability ratio under varying number of network flows $n \in [1, 25]$ for both shortest path routing and minimal-overlap routing when using the *degree centrality* metric for gateway designation compared to a random benchmark. **Bottom**: The absolute deviation in terms of schedulability ratio of the other centrality metrics w.r.t. the degree centrality.

networks, even if a real-time routing scheme is considered.

5. CONCLUSIONS

This paper presents a novel framework towards joint scheduling, routing and gateway designation in real-time TSCH networks. By resorting to prior methods for joint routing and scheduling, and centrality-driven gateway designation, we show by simulation that a combined approach which takes into account all three dimensions can improve schedulability by $\sim 80\%$, scheduling up to three time more real-time flows than a basic configuration. We are working on further investigating the performance of the framework with a broader range of varying parameters (network density, number of channels, number of gateways) as well as to study its applicability in related real-time network domains (wireless TSN, 5G).

6. ACKNOWLEDGMENTS

This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UIDB/04234/2020); by the Operational Competitiveness Programme and Internationalization (COMPETE 2020) under the PT2020 Agreement, through the European Regional Development Fund (ERDF); by FCT and the ESF (European Social Fund) through the Regional Operational Programme (ROP) Norte 2020, under PhD grant 2020.06685.BD.

7. REFERENCES

[1] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. Improved multiprocessor global schedulability analysis. *Real-Time Systems*, 46(1):3–24, 2010.

[2] D. Dujovne, T. Watteyne, X. Vilajosana, and P. Thubert. 6TiSCH: deterministic IP-enabled industrial internet (of things). *IEEE Communications Magazine*, 52(12):36–41, 2014.

[3] M. G. Gaitán, L. Almeida, A. Figueroa, and D. Dujovne. Impact of network centrality on the gateway designation of real-time TSCH networks. In *2021 17th IEEE Int. Conf. on Factory Communication Systems (WFCS)*, pages 139–142. IEEE, 2021.

[4] M. G. Gaitán, L. Almeida, P. M. Santos, and P. Meumeu Yomsi. EDF scheduling and minimal-overlap shortest-path routing for real-time TSCH networks. In *Proceedings of the 2nd Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2021)*, volume 87, pages 2–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021.

[5] M. G. Gaitán, P. M. Yomsi, P. M. Santos, and L. Almeida. Work-in-progress: Assessing supply/demand-bound based schedulability tests for wireless sensor-actuator networks. In *2020 16th IEEE Int. Conf. on Factory Communication Systems (WFCS)*, pages 1–4. IEEE, 2020.

[6] M. G. Gaitán and P. Yomsi Meumeu. Multiprocessor scheduling meets the industrial wireless: A brief review. *U. Porto Journal of Eng.*, 5(1):59–76, 2019.

[7] M. G. Gaitán and P. M. Yomsi. FF-DBF-WIN: On the forced-forward demand-bound function analysis for wireless industrial networks. In *Work-in-Progress Session of the 30th Euromicro Conference on Real-Time System (ECRTS)*, pages 13–15, 2018.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[9] C. Lu, A. Saifullah, B. Li, M. Sha, H. Gonzalez, D. Gunatilaka, C. Wu, L. Nie, and Y. Chen. Real-time wireless sensor-actuator networks for industrial cyber-physical systems. *Proceedings of the IEEE*, 104(5):1013–1024, 2015.

[10] A. Saifullah, Y. Xu, C. Lu, and Y. Chen. End-to-end communication delay analysis in industrial wireless networks. *IEEE Transactions on Computers*, 64(5):1361–1374, 2014.

[11] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund. Industrial internet of things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial Informatics*, 14(11):4724–4734, 2018.

[12] C. Xia, X. Jin, and P. Zeng. Resource analysis for wireless industrial networks. In *2016 12th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pages 424–428. IEEE, 2016.

Toward Precise Real-Time Scheduling on NVidia GPUs

Nordine Feddal
Univ. Lille, CNRS, Inria,
Centrale Lille
UMR 9189 CRIStAL
F-59000 Lille, France
nordine.feddal.etu@univ-lille.fr

Houssam-Eddine Zahaf
Nantes Université, École
Centrale Nantes, IMT
Atlantique(1) , CNRS, INRIA
(1), LS2N, UMR 6004,
F-44000 Nantes, France
houssameddine.zahaf@univ-nantes.fr

Giuseppe Lipari
Univ. Lille, CNRS, Inria,
Centrale Lille
UMR 9189 CRIStAL
F-59000 Lille, France
giuseppe.lipari@univ-lille.fr

ABSTRACT

Scheduling a set of real-time tasks on modern GPUs is challenging due to the black-box nature of most GPUs and the lack of proper documentation. Moreover, with the rapid evolution of GPUs architectures, an important problem is to design a real-time scheduler for current and future GPU architectures. This paper describes on-going efforts to schedule real-time tasks on Nvidia GPUs. We propose the PRUDA (Predictable Real-time CUDA) library to manage GPU resources as multi-core systems.

Keywords

GPUs, CUDA, scheduling, multicore, real-time.

1. INTRODUCTION

Autonomous driving and Advanced Driver-Assistance Systems (ADAS) have received a particular attention from academic and industrial communities. These systems process a large amount of data, on which a series of AI inference kernels are applied. One of the main concerns to build these systems is to ensure correctness, while guaranteeing the respect of timing constraints, dictated by the evolution of the surrounding obstacles and path planning. Satisfying these constraints requires processing computer vision algorithms (image preprocessing, computer vision inferences, and actuation) within a predefined time window. Classical identical core platforms, composed of a set of identical CPUs, are not able to fulfill the required timing constraints, therefore automotive industries are using heterogeneous platforms featuring highly-parallel accelerators such as GPUs, along with a set of identical CPU cores.

An example of such platform is the NVIDIA Jetson Family. The Jetson AGX, for example, features 8 CPUs, along with an integrated Volta GPU and other accelerators such as Programmable Vision Accelerator (PVA) and Deep Learning Accelerator (DLA). A typical NVIDIA GPU is composed of hundreds of components able to achieve computations called CUDA cores, arranged into a set of streaming multiprocessors (SMs). The SMs share a complex memory hierarchy at different levels.

While real-time systems require that the run-time behavior can be analyzed to guarantee the respect of timing constraints, the GPU complex design make it difficult to predict the timing behavior of a real-time task. In addition, NVIDIA GPUs internals are closed-source, for intellectual property concerns, making it more difficult to control the tasks' execution within the GPU.

The real-time community has done a considerable effort to conciliate predictability and performances for NVIDIA GPUs. GPU scheduling has mainly been controlled by software schedulers on the top of NVIDIA internals (see Section 3). In the same effort, our previous work [6] enables preemption at block level, and a first attempt to enable predictable concurrent and parallel execution on GPUs. However, this work is restricted to NVIDIA GPUs with a limited number of SMs. When using many SMs, unpredictable behavior might be observed.

Contribution. In this paper, we present an evolution of PRUDA, a programming platform to manage GPU resources. PRUDA offers various strategies to control real-time execution within a GPU with CUDA. We use the GPU as a multicore platform, where each SM is considered as a separate processor, and a kernel is executed exclusively on a single processor. Each streaming multiprocessor is therefore scheduled using a single core real-time policy, and different kernels are either run on parallel on different SMs, or concurrently on the same SM.

Organization: The remainder of this paper paper is organized as follows. We provide a brief overview of NVIDIA GPUs and CUDA programming in Section 2. We briefly report related work in Section 3. We then explore various approaches to managing GPU execution using PRUDA in Section 4, and draw our conclusions in Section 6.

2. BACKGROUND ON NVIDIA GPUS AND THEIR PROGRAMMING

2.1 Overview on GPU architecture

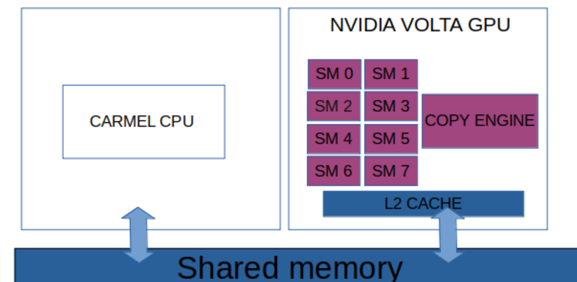


Figure 1: Jetson AGX Xavier Architecture

NVIDIA GPUs are composed of multiple computing elements, called CUDA cores. Multiple CUDA cores are grouped in a single abstract computation unit called *streaming multiprocessor* (SM). The different SMs and CUDA cores have a complex interconnect with multiple levels of memory hierarchy. For the sake of simplicity, we only report the L2 cache, which is shared between all SMs. A GPU features, as well, one or multiple copy engines (CEs): they are coprocessors responsible for achieving memory copy operations between different memory spaces (from CPU to GPU, from GPU to CPU).

In this work, we consider the NVIDIA Jetson AGX Xavier. The Jetson AGX Xavier is composed of 8 ARM (ISA v8) cores, two NVDIA DLAs (NVDLAs), a vision accelerator and a single VOLTA GPU. The Volta GPU contains 512 CUDA cores, arranged in 8 SMs. The different SMs share a cache of 512KB. CPUs and GPU share a common main memory. We report this architecture in Figure 1.

Listing 1 A typical CPU-GPU code

```
__global__
void vecMul(int * A, int *B, int *C){
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  C[i] = A[i] * B[i];
}

int main(int argc, char ** argv) {
  // ... some CPU side initialization
  int N = SIZE;

  int *h_a, *h_b, *h_c, *d_a, *d_b, *d_c;

  // ... GPU size allocation and initialization

  cudaMalloc(&d_a, N*sizeof(int));
  cudaMalloc(&d_b, N*sizeof(int));
  cudaMalloc(&d_c, N*sizeof(int));

  cudaMemcpy(d_a, h_a, N*sizeof(int),
                        cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, h_b, N*sizeof(int),
                        cudaMemcpyHostToDevice);

  int nbblock=8;
  int nbthreads=N/nbblock;
  vecMul<<<nbblock, nbthreads>>>(d_a, d_b,d_c);

  cudaMemcpy(h_c, d_c, N*sizeof(int),
                         cudaMemcpyDeviceToHost);

  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  free(h_a); free(h_b); free(h_c);
}
```

2.2 GPU Programming

A GPU can be programmed using generic programming platforms such as OpenCL or proprietary APIs such as CUDA for NVIDIA GPUs. CUDA allows having a tighter control over GPU resources, therefore it is the one used in this work. Our library is implemented in C/C++ and compiled using the NVIDIA NVCC compiler.

The CUDA API provides primitives and commands to copy data between the CPU (commonly called *host*) and the GPU (commonly called *device*), to allocate memory on the device, and to submit work (commonly called *kernels*) to the GPU.

CUDA programs have a common structure, similar to the one described in Figure 1. First, memory is allocated onto the GPU visible GPU memory (Lines 15-17). Further, a copy memory operation is achieved using the copy engine to copy data from host CPU visible memory space to device memory space (Lines 20-23). This operation is mandatory even for integrated GPUs where the CPU host and the GPU share the same physical memory. It allows an integrated GPU code to be portable to discrete GPU code, without any restrictions. This step can be transparent to the end user if the unified memory is used (*i.e.* memory is allocated using `cudaMallocManaged`). Further, the kernel is launched on the GPU (Line 27), by invoking the CUDA kernel. We further describe this operation, as it is a very important step in the design of PRUDA.

Once the CUDA kernel completes on the GPU, the result data is copied back to the host (Line 29). Finally, memory is freed for both host and device. CUDA malloc and free are costly operations. In the context of a real-time system, `cudaMalloc` is performed only once, before the real-time system starts, and the memory is reused during the periodic execution of the tasks.

CUDA code that starts with token `__global__` can be invoked from the host, while those that start with `__device__` can only be invoked by another CUDA kernel. A CUDA kernel represents the code that a single CUDA thread must execute. A thread is identified by its number within its block. Therefore, when a kernel is invoked, the CUDA developer must specify the parameters *numBlocks* and *threadsPerBlock* defining respectively the number of blocks and the number of threads.

All threads of the same block are executed on the same SM [1], however different blocks of the same kernel may be executed on different SMs. The NVIDIA scheduler schedules first blocks and then warps (a group of 32 threads). Threads within a warp execute the same instruction on different data following the Single Instruction, Multiple Thread (SIMT) model.

CUDA API uses a FIFO queue data structure, to submit work to the device, in a FIFO policy. By default, GPU commands are inserted into the default stream (i.e. NULL stream). The CUDA stream can be defined in the kernel invocation parameters. When the cuda stream is not defined, the default stream is used, and all kernels are therefore executed one after the other, allocating implicitly all the GPU resources to the under running kernel. CUDA API allows creating multiple streams to run concurrent and parallel kernels within the GPU.

3. RELATED WORK

Over the past few years, there have been many studies related to the implementation of a real-time GPU scheduler: [4, 1, 2] have attacked the problem of providing support to real-time systems onto GPUs from different perspectives. Kato et al. proposed different platforms (TimeGraph and RGEM) for non-preemptive scheduling for graphical tasks in GPU [4, 3]. Authors of [1] tried to study how a GPU takes scheduling decisions based on black-box experimentation on

the Jetson TX2 platform.

Capodieci et al. [5] modified the proprietary NVIDIA driver to implement an event-driven scheduler allowing to use fine grain preemption levels provided by recent GPUs (since Volta architecture) under different policies such as EDF and fixed priority. The authors used NVIDIA Drive PX platform that is commonly used in ADAS and autonomous vehicle application. However, some details were omitted, and their source code was not released due to NVIDIA non-disclosure agreements.

Elliott et al. [2] consider the GPU scheduling problem as a synchronization problem. GPUSYNC is a real-time GPU framework using k-exclusion locks to allow mutually exclusive access to a number of GPUs in a system. GPUSync source code can be found here [1]. The work in [5] has closed sources whereas GPUSync [2] platform does not support GPU preemption. In both cases, GPU is used as a single-core platform.

4. PRUDA

In this section, we present the design of PRUDA, our programming platform. It is a set of functions to program real-time tasks on NVIDIA GPUs and take scheduling decisions. As described in [6], PRUDA uses *CUDA streams* to enforce scheduling decision and concurrent execution.

A fixed priority can be assigned to every different *CUDA stream* using the *cudaCreateStreamWithPriority* primitive. All kernels in a high-priority *CUDA stream* will be executed prior to kernels in a low-priority *CUDA stream.* Moreover, if a kernel in a low-priority stream is running, and a kernel of a higher priority stream is submitted, the GPU can preempt the current kernel to execute the kernel in the higher priority stream. Fine-grained preemption capabilities are available in NVIDIA GPUs starting from the PASCAL architecture. In the PASCAL architecture a preemption is possible at the block level, i.e. preemption is achieved when all threads of a given block finish their execution.

PRUDA provides three strategies to implement scheduling decisions, according to the user needs and goals. In the first and the second strategy, the GPU is abstracted as a single core platform. In this paper, we developed a third strategy in which the GPU can be seen as a multi-core platform. The new strategy is described in the next Section 5.

The different strategies have a common design: all tasks scheduled using our platform are stored in a task-queue called `tq`. When a task is activated, its priority is computed and it is inserted accordingly to the active run-queue denoted by `rq`, by invoking the `subscribe()` function. Later, according to the selected strategy, tasks are consumed from `rq`, when the `resched()` function is invoked, which moves the task from the active run-queue to the corresponding CUDA stream.

4.1 Single-Stream strategy

The first strategy uses only one CUDA stream. Consequently, once a PRUDA task is executing, it cannot be preempted by another submitted PRUDA task before its completion. Only non-preemptive scheduling algorithms can be implemented with this strategy. The advantage of this strategy is that it is simple to implement, and it provides an implicit synchronization between PRUDA tasks due to the FIFO nature of the CUDA stream. However, as only one PRUDA task can run at a given time, this strategy involves reserving all GPU resources (all SMs) for the current running PRUDA task, even if the task is not using all GPUs cores. This leads to GPUs resources waste, which can be avoided by using another strategy to enable preemption and concurrent execution between different PRUDA tasks.

4.2 Multiple-Stream strategy

Using this second strategy, PRUDA creates as many priority levels as there are streams, allowing concurrent kernel execution and preemption. The Jetson AGX Xavier provides only two priority levels, one denoted high priority (`h-sq`) and one denoted low priority (`l-sq`). When a scheduling event occurs, the scheduler checks if:

- (1) `h-sq` $= \emptyset \wedge$ `l-sq` $= \emptyset$: the scheduler will allocate the task to the `l-sq` queue, therefore the task will be submitted immediately to the GPU.
- (2) `h-sq` $= \emptyset \wedge$ `l-sq` $\neq \emptyset$: the scheduler checks if the highest priority in `rq` is greater than the priority of the task in `l-sq`. If yes, the task is inserted into the high priority queue `h-sq`. Therefore, it preempts the task in `l-sq`, when possible.
- (3) `h-sq` $\neq \emptyset \wedge$ `l-sq` $\neq \emptyset$: No scheduling decision are taken.

This strategy solves the preemption limitations of the single-stream, however it still uses the platform as a single core.

5. PRUDA EXTENTION: SM AS CORE

The third strategy, called *SM as core* distinguishes two types of kernels: those executing in a single stream, denoted as $\mathcal{K} = \{k_1, k_2, \cdots\}$, and those that are free to execute on an arbitrary number of SMs, without any restriction, denoted as $\overline{\mathcal{K}}$. In this strategy, as many priority levels as streams are created, similarly to the *Multiple-Stream strategy.* In addition to the scheduling structures described for the previous strategy, the *SM as core* strategy creates a stream for *noise kernels* and a stream for every SM. The goal of the noise stream is to be able to asynchronously submit *noise kernels*, having a high priority with respect to the executing kernels. The other streams are defined to schedule kernels that are meant to be allocated to a predefined SM. `h-sq` and `l-sq` are kept to schedule the kernels in $\overline{\mathcal{K}}$, similarly to the previous strategy.

First, we explain the general idea behind our strategy. When the PRUDA scheduler selects kernel k in $\mathcal{K}$ to execute on SM x, it first launches a *noise kernel* (k_n) which occupies all GPU resources on all SMs with dummy code. Further, the threads of k_n executing on the SM x, where k is meant to be allocated, exit immediately by invoking primitive `asm("exit;")`. Therefore, the GPU internals will have only one SM at disposal to schedule blocks of kernel `k`. The other threads of the *noise kernel* k_n are maintained active, until kernel k completes.

Although this solution allows allocating a kernel to a predefined SM, all other SMs are occupied by dummy code, preventing therefore other kernels to execute. We avoid these scenarios by calibrating the number of threads and blocks of

[1]GPUSYNC Project github. https://github.com/GElliott/litmus-rt-gpusync/.

the noise kernel. The goal is to define the number of blocks and threads per block, such that the NVIDIA internals will find only a single SM available for executing kernel k, without occupying all the GPU resources, henceforth allowing other kernels to execute. However, our solution is architecture specific, *i.e.*, the noise kernel configuration will depend on the target GPU architecture (SM count, maximum number of threads per block, and maximum number of threads per SM). PRUDA is able to automatically be adapted to different GPUs, however the binary code can not be directly ported to different GPUs, without breaking PRUDA scheduler.

PRUDA starts by computing the number of threads of the noise kernel. It must allow all other kernels in $\overline{\mathcal{K}}$ to execute, therefore the number of threads of the noise kernel is set such that the sum of noise kernel threads and the maximum number of threads of any block of $\overline{\mathcal{K}}$ does not exceed the number of threads per SM, allowed by the GPU hardware design, as follows:

$\texttt{nbThreads} = \frac{\texttt{MaxThreadsPerSM} - \max_{\texttt{blockSize}}(\overline{\mathcal{K}})}{2}$, where:

- `MaxThreadsPerSM` is the maximum number of threads per SM (2048 for Jetson AGX Xavier)
- $\max_{\texttt{blockSize}}(\overline{\mathcal{K}})$ is the maximum number of threads assigned to kernel in $\overline{\mathcal{K}}$ (in the example of Table 1 it is equal to 512).

Our strategy modifies the number of blocks and threads defined by the CUDA developer, such that our restrictions are guaranteed to be respected at runtime. Therefore, PRUDA provides a simple and efficient thread indexing mechanism that the CUDA developer uses instead of CUDA classical indexing mechanisms. PRUDA uses a global flag and inter-block synchronization to its thread indexation mechanisms, and keeps track of the running user- and noise-kernels.

We illustrate our approach by the following example.

Example 1. *Let K_1, K_2, K_3 be 3 GPU kernels such that $\mathcal{K} = \{K_2\}$ and $\overline{\mathcal{K}} = \{K_1, K_3\}$. The kernel parameters are described in Table 1.*

	K_1	K_2	K_3
num blocks	6	6	6
threads per block	512	256	512

Table 1: Task set Γ.

The noise kernel size that allow K_1 and K_3 to execute without any restriction is 768 *threads per block.*

Figure 2 illustrates the execution trace of the task set of Table 1.

At time (a), Kernel K_2 arrives and, since it is in $\mathcal{K}$ and allocated to SM-1, PRUDA launches a noise kernel k_n with 768 *threads per block to fill all SMs except SM-1. PRUDA reassigns the thread per block of Kernel K_2 to 1024, so that Kernel 2 cannot be dispatched on any other SM but SM-1.*

At time (b), while k_n and K_2 are executing, Kernel K_1 arrives. As the noise kernel size is 768 threads per block, at least 1 block of K_1 can be executed on each SMs except SM-1.

At time (c), while k_n and K_2 are still running, Kernel K_3 arrives and, like K_1; it can be executed on every SMs except SM-1. At (d), K_2 has completed its execution; it modifies the global flag to notify the noise kernel k_n to stop execution. At time (f), the noise kernel has finished.

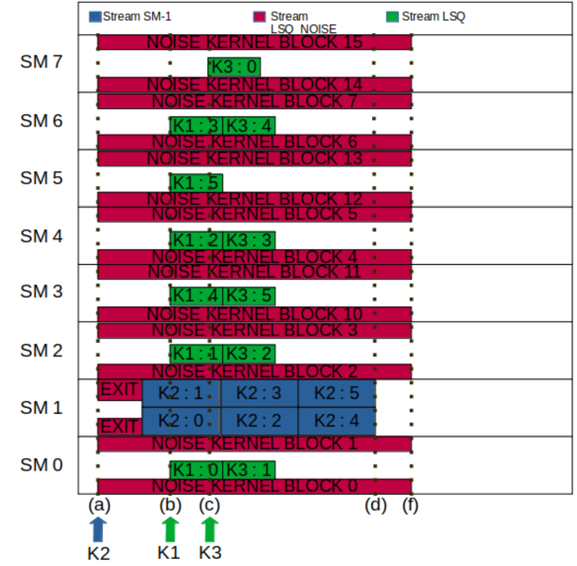


Figure 2: Concurrent Execution with One SM task.

6. CONCLUSIONS AND FUTURE WORKS

In this paper, we extended the PRUDA library with new strategy for real-time GPU management, which provides a way to isolate the execution of a real-time task on a single SM. This is achieved by means of *noise kernels* that fill any SM except the one allocated to the task, and thus forces the hardware scheduler to allocate the task on the selected SM.

At the moment, we influence the scheduling decision with the concept of noise kernel by manipulating the number of threads per block assigned to the noise kernel. In the future, we want to take into account the amount of shared memory and the number of register the block will use on the SM. We plan to trace PRUDA task execution, and estimate a possible GPU state at run time.

We also want to evaluate the performance of our library on computer vision applications. To do this we plan to integrate PRUDA into a CV library such as OPENCV.

References

[1] Tanya Amert et al. "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed". In: *2017 IEEE Real-Time Systems Symposium (RTSS)* (2017), pp. 104–115.

[2] Glenn A. Elliott, Bryan C. Ward, and James H. Anderson. "GPUSync: A Framework for Real-Time GPU Management". In: *2013 IEEE 34th Real-Time Systems Symposium.* 2013, pp. 33–44. DOI: 10.1109/RTSS.2013.12.

[3] Shinpei Kato et al. "RGEM: A Responsive GPGPU Execution Model for Runtime Engines". In: *2011 IEEE 32nd Real-Time Systems Symposium* (2011), pp. 57–66.

[4] Shinpei Kato et al. "TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments". In: *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. Portland, OR: USENIX Association, June 2011. URL: https://www.usenix.org/conference/usenixatc11/timegraph-gpu-scheduling-real-time-multi-tasking-environments.

[5] Ignacio Sañudo Olmedo et al. "Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective". In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 213–225. DOI: 10.1109/RTAS48715.2020.000-5.

[6] Houssam-Eddine Zahaf and Giuseppe Lipari. "Design and Analysis of Programming Platform for Accelerated GPU-Like Architectures". In: *Proceedings of the 28th International Conference on Real-Time Networks and Systems*. RTNS 2020. Paris, France: Association for Computing Machinery, 2020, pp. 1–10. ISBN: 9781450375931. URL: https://doi.org/10.1145/3394810.3394826.

Shortening gate closing time to limit bandwidth waste when implementing Time-Triggered scheduling in TAS/TSN

Pierre-Julien Chaine
Airbus
Toulouse – France

Marc Boyer
ONERA / DTIS, Université de Toulouse
F-31055 Toulouse – France

ABSTRACT

Time-Triggered (TT) communications consists in specifying the instants (or time windows) at which any frame will be transmitted on any network link. In TSN Ethernet, this principle can be implemented using dedicated queues and time-triggered gates. Since TSN can mix several kinds of flows, the gates of non-TT flows are commonly closed during a TT window (when TT flows are transmitted). This is called "*exclusive gating*". In this paper, we propose to use "*protective gating*", that sets TT queues at the highest priority levels and reduces this closing time to its minimal value in order to save bandwidth while guaranteeing correct TT behavior.

1. INTRODUCTION

Time-Triggered (TT) communications consists in specifying offline the instants (or time windows) at which any frame will be transmitted [1]. In a store and forward network, a time window must be reserved for each frame on each link, and a good schedule keeps the waiting time of each frame in each switch as small as possible. Nevertheless, in complex systems, TT applications may coexist with Event-Triggered (ET) applications (i.e. producing ET data flows). In general, the networking technologies allow ET flows to use the medium outside the time windows dedicated to TT flows [2]. This approach is called "*exclusive gating*". This time partitioning may lead to some waste of bandwidth. In fact, when the time window dedicated to a TT flow is not used, or partially used, the unused time cannot be used by ET flows.

In this paper, instead of protecting the whole TT time window, we propose to protect only its beginning, while setting TT queues at the highest priority levels. This approach is called "*protective gating*" and, in the context of a TSN (Time Sensitive Networking) network, can offer the same guarantees to TT traffic, while reducing the bandwidth waste.

2. MEDIUM ACCESS IN TSN

Let us first recall the architecture and frame selection rules in a TSN output port. A TSN output port is made of up to 8 queues. To each queue is associated a Transmission Selection Algorithm (TSA) and a gate. The port itself uses a single static priority arbiter (cf. Figure 1).

The TSA determines, at each instant, if the head of queue frame is "*available for transmission*" [3, § 8.6.8]. There are currently 4 possible TSA (Static Priority, Credit-based Shaper, Enhanced Transmission Selection and Asynchronous Traffic Shaper) [3, Table 8.6], [4]. Each TSA has its own rules to determine the availability for transmission.

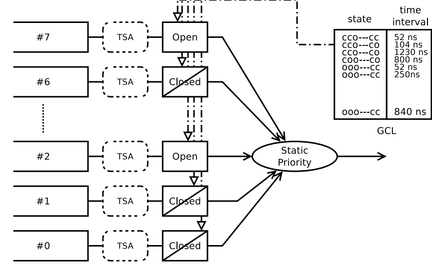


Figure 1: Architecture of a TSN output port

The gate is either open or closed. Each output port has it own Gate Control List (GCL). Each entry in the list has a duration (known as "*TimeInterval*") and a state for each queue gate (open or closed). A state machine crosses the list, opening and closing each gate during the specified duration, in a cyclic way, as defined in the Time Aware Shaper (TAS) addendum [5].

The output port arbiter considers the set of queues satisfying three conditions: first, the head of queue frame is set as available for transmission by the queue TSA, second, the gate is open, and third, there is "*sufficient time available to transmit the entirety of that frame before the next gate-close event*" [5, §8.6.8.4]. Then, the arbiter selects, in this set, the head of the queue frame with the highest priority (i.e. with the highest queue number).

Note that this implies that *when the gate of a queue is closed, it is not sending any frame.*

Also note that the condition on next closing event is often presented as a "*guard band*", but to highlight the fact that this condition is dynamic, depending on the frame size, we will call it "*next closing dynamic guard band*" or simply "*dynamic guard band*".

3. TT COMMUNICATIONS IN TSN

3.1 TT communications in store-and-forward networks

A TT schedule consists in considering a global time horizon (often the least common multiple - lcm - of the period of all data flows), and allocate, to each frame sent to the network during this period, a slot (aka time window) on ev-

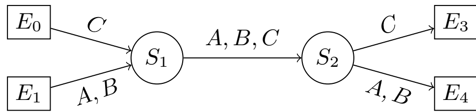


Figure 2: Simple topology

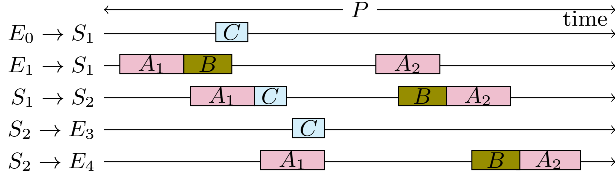


Figure 3: Example of TT schedule for network in Figure 2

ery link along its path from source to destination (this path being considered as fixed).

Consider the network depicted in Figure 2 with three data flows, A, B, C, with routing depicted on the Figure, and assume a period P for B, C and $P/2$ for A.

A simple example of TT schedule is presented in Figure 3. On each link, two slots, A_1 and A_2, are reserved for flow A, and only one slot, denoted B (resp. C) is reserved for flow B (resp. flow C).

Consider first flow C: it has a slot scheduled on link $E_0 \rightarrow S_1$, and right after that (as soon as the frame is fully received, plus some margin due to clock error and internal commutation delay), another slot is scheduled on link $S_1 \rightarrow S_2$, and the same on link $S_2 \rightarrow E_3$. As shown in Figure 3, the frames of flow C spend the least amount of time in intermediary buffers They thus benefits from the smallest end-to-end network latency. The flow A has the same kind of behavior, with slots A_1 and A_2. The situation is different for B: it is sent just after A_1 on $E_1 \rightarrow S_1$, but it stays longer in the buffers of S_1 since it is forwarded only just before A_2. It then experiences a latency larger than the other flows (but it may be sufficient w.r.t. its requirements).

This behavior has been chosen to illustrate several points. First, notice that it is not possible to ensure a minimal latency for all frames without changing the instants at which these frames are injected into the network (that may impose to change the instants when the embedded data are computed, or to induce some waiting time between computation and injection). Second, some frames are sent back-to-back, therefore, the sequence A_1B can be seen either as two adjacent slots, or as a single slot hosting two frames. Third, some trade-offs may exist on slot size w.r.t ET data flows: on the one hand, during a slot dedicated to TT frames, no ET frame can be forwarded, therefore a large TT slot increases the latency of ET flows, which favors small slots; but on the other hand, the time before a TT slot cannot always be used (i.e. a large ET frame cannot start its transmission if it cannot be completed before the TT slot), creating a per-slot penalty, which favors a small number of slots. Fourth, in this schedule, frame B is fully received by S_1 before frame C, but is forwarded after.

In summary, building a TT shedule consists in defining slots (without any encroachment between two slots) and assigning frames to slots while optimizing criteria such as number of slots, slot length, end-to-end frame latency, while satisfying that each frames will be received before its departure slot on any hops.

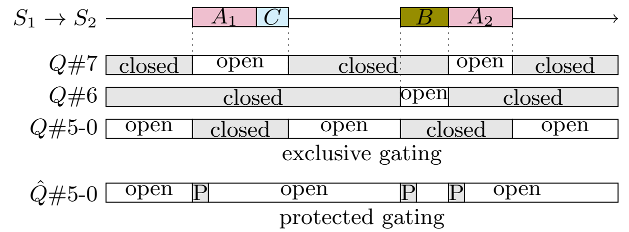


Figure 4: Example of GCL schedule implementing TT schedule for link $S_1 \rightarrow S_2$ in Figure 3, with A, C at priority 7 and B at priority 6.
The time-lines $Q\#7$ and $Q\#6$ represent the state of the gates for these TT queues, and $Q\#5\text{-}0$, represents the state of the gates for all other queues, in the "*exclusive gating*" mode. The "*exclusive gating*" mode uses the same gate scheduling for queues $Q\#7$ and $Q\#6$ but uses $\hat{Q}\#5\text{-}0$ for the others.

3.2 Implementing TT communications in TSN

As presented in Section 2, the TT behavior of TSN is based on queues, not on flows. Then, to send a frame during a predefined time window, this frame has to be written in a queue before the time window, when the queue gate is closed. The frame will be transmitted during the time window, when the gate is open (if the slot is large enough).

Also note that the order of frames in a queue is in general the arrival order[1]. Then, it is not possible to implement the scheduling of Figure 3 using a single queue for TT frames in S_1, since B is received before C but forwarded after[2]. In this example, one solution would be to shift to the right the slot dedicated to B on link $E_1 \rightarrow S_1$ (i.e. postpone B transmission). Then, it would be received after C and the output order will be the same as the input order. But for illustration purposes, let us keep this scheduling.

To implement such a schedule in TSN, the common practice consists in assigning one or several queues to TT flows (in the example depicted in Figure 4, queues #7 and #6), and opening the gate only during the slots of frames associated to this queue. All non-TT queues keep their gate closed when there is at least one TT-queue open, and open it when all TT-queues are closed. The TT queues have no TSA. This is known as "*exclusive gating*".

Note that exclusive gating does not require that the TT queues have the highest priority: during a slot, a TT queue has exclusive access to the output port, and the static priority arbiter is useless. Nevertheless, it is a common practice to do so.

4. STATE OF THE ART

Building a TT schedule is a hard and old problem, and a complete overview can be found in [6].

The opportunity to support both TT and ET flows in an Ethernet context is a feature of TTEthernet. It relies on a per-frame time slot allocation, with dedicated hardware support, and not on a per queue gate opening and closing. Like

[1]The exact requirements are specified in [3, § 8.6.6].
[2]In TTEthernet, such a scheduling is not a problem since there is no order requirement between TT frames.

TSN, it has to handle the case of ET frames trying to access the medium just before a TT slot, and supports 3 integration methods: timely block, shuffling and preemption [2]. A lot of work have been done on the efficient computation of a global schedule, see for example [7, 8, 9].

Once the IEEE have defined an ET real-time extension of Ethernet, known as AVB, which ensures guaranteed latency and controlled jitters, the need of "*temporal isolation*" for "*scheduled traffic*" (ST) appeared, and it was proposed to use a "*separate class*" (i.e. one dedicated queue) "*in the highest priority*" [10].

As presented in Section 3.2, the queue-based storage of frames requires to adapt the algorithms developed for TTEthernet.

In order to cope with the potential non-determinism induced by the loss of a frame, [11] adapts the constraints of [7] and introduces *Flow Isolation* and *Frame Isolation*. In order to take into account non-TT traffic while building the TT schedule, [12], [13] and [14] introduce strategies to modify the TT frame schedule by either spacing the frame offsets or gathering them.

Most recently, a second approach with configurations based on schedule per group of frames instead of per frame, has appeared. [15] applies the TTEthernet schedule generation methodology [16] to TSN networks. The authors introduce new sets of constraints adapted for group of frames schedules as well as *Stream Isolation*, a fusion of *Frame isolation and Flow isolation* to again cover the loss of a frame. In [17], the same authors use their new constraints to implement a configuration generator and compare their two approaches. More recently [18] proposes a group of frames configuration but chooses not to use exclusive gating like all other configuration generators. Moreover, it considers non-TSN end-stations (i.e. Ethernet) in their system.

The first use of the expression "*exclusive gating*" in the context of TSN seems to appear in [19].

5. PROPOSAL: PROTECTIVE GATING

Our proposal consists in relaxing the "*exclusive gating*" by only closing the gate of each non-TT queue during a time interval as small as possible (depending on the implementation[3]) just when the gate of a TT queue opens, and setting all TT queues to the highest priority levels (i.e. if there are 3 TT queues, they will be queues #7, #6, #5), eventually by reassigning queues, but maintaining their relative order. This reduction of the closing time is called ' '*protective gating*", and is illustrated in Figure 4, where the P in gray boxes stands for "*Protective*".

Note that it does not mean that each closed window in the non-TT scheduling is replaced by a single short "*protective*" closing event: it may be replaced by several ones. Consider Figure 4: if the third protective closing event was absent, if frame B was lost or shorter than its slot, a non-TT frame would be able to start its transmission before the A_2 slot and prevent/delay the transmission of A_2 frame.

We claim that the small modification introduced by protective gating offers a small benefit without any cost.

5.1 Condition for no impact on TT traffic

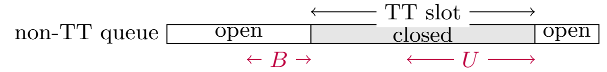


Figure 5: Bandwidth possible loss due to gate closing.

If all the frames that must be sent during a slot are in the output queue before the slot opening, and if the TT queues have highest priorities, then "protective gating" offers the same behavior to these frames than "exclusive gating".

Consider one TT queue, and one of its slot. Assume that there are n frames to be sent during the slot and that all are in the output queue before the slot opening.

In an "*exclusive gating*" configuration, at the slot beginning, the gate of this queue opens and all other gates are closed (they are just closing or already closed). During the whole slot, only this queue has its gate open, and only this queue has access to the output port. Since all frames are already in the queue, they are sent back-to-back (there is no TSA to block a frame).

In a "*protective gating*" configuration, at the beginning of the slot, the gate of this queue opens and all other gates are closed (they are just closing or already closed). Then, the head of the TT queue can be sent. At the end of transmission of this first frame, the second frame of the slot is ready for transmission. It may exist frames ready for transmission in others queues, but these queues are non-TT. Since the TT queue has the highest priority, this second frame is sent just after the first one. The same argument holds at the end of this frame, and the next ones. Then, all frames of the slot are sent back-to-back, a behavior equivalent to the "*exclusive gating*" configuration.

5.2 Impact on non-TT traffic

After having shown that our approach do not penalize TT traffic, let us discuss the impact on non-TT traffic.

5.2.1 Bandwidth gain

In exclusive gating, the TT traffic has two negative impacts on the other flows. First, a frame can be blocked if it cannot be fully transmitted before the gate closing (called "*dynamic guard band*"). This blocking time (B in Figure 5) is at most the transmission time of a frame of maximal size. Second, no frame can be sent during a TT slot, even if no TT frame is being sent. This unused time (U in Figure 5) can be as large as the slot.

Our proposal does not address the blocking, but mitigates the unused time. It creates a gain when a slot is not fully used by TT frames. Let us now consider when this may happen. Indeed, as presented in Section 4, there are several ways to use the TAS mechanism in TSN. We use here the terminology from [18].

The strategy used to build the TT schedule has an impact on the possible gain, and the main strategies (0GCL, FGCL, WND, FWND) will be presented further.

But let us start with a global discussion on slot size. The simplest way to build a TT schedule consists in considering each TT frame independently, and assigning a slot to each frame. But this may lead to a high number of slots. And because of the dynamic guard band, this has a negative impact. Moreover, this may lead to "a large number of GCL events exceeding the hardware capabilities of existing TSN

[3]The duration associated to each opening/closing event must be a multiple of an implementation-defined constant, `TickGranularity` [5, §8.6.9.4.16].

devices" [18]. For these two reasons, it is better to reduce the number of slots, putting several frames in the same slot. Nevertheless, a long slot increases the latency of non-TT flows, and a good TT schedule has also to consider non-TT flows [20].

We now present the main ways to use TAS in TSN, and the associated gain.

0GCL, FGCL.

The goal of 0GCL is to ensure 0 jitter to TT flows [11]. Frame-to-Window based GCL (FGCL) relaxes the 0 jitter requirement [17]. But both consider schedules that assign to each slot a fixed set of ordered frames.

These methods are not perfect. In fact, it has been shown in [11] that, due to the queue-based storage, a loss of frame can break the schedule. Counter-measures have been developed (called "*flow isolation*" and "*frame isolation*") but they tend to limit the number of frames per slot.

In these approaches, a slot may be not fully used either when TT frames are of variable size (then requiring a reservation for the maximal frame size), or when a TT frame is absent (because of loss at network level, or because the application did not produce the data, for example in case of oversampling).

The gain provided by protective gating is then limited.

WND, FWND.

The requirement of knowing in advance which frames will use which slot can be considered as an over-specification. The Window-based scheduling (WND) and Flexible Window-based scheduling (FWND) build schedules that ensure guaranteed latency for every frame, without knowing exactly which slot a frame will use. This may lead to under utilization of some slots [18, 21].

The protective gating then allows to re-use the unused part of these slots.

5.2.2 Impact on credit-based shaper

The gate closing (associated to TT slots) also has an impact on the Credit-based Shaper (CBS) Transmission Selection Algorithm. CBS is based on a credit associated to a queue. Its value decreases when the queue sends a frame, and increases either to refuel up to 0 or when the head of queue is blocked by a frame of another queue. The value of the refuel/increase slope is a network administration parameter. In case of gate closing, the value of the credit is frozen, but the slope value is globally updated to compensate this freeze time [5, §8.6.8.2 d)]. This rule seems to be designed to avoid burst when the gate re-opens, but its real impact is not well known [22]. In protective gating, most of this effect may be reduced.

5.2.3 Easy emergency traffic integration (and 802.1AS messages)

Emergency events, alarms, may be raised in an event-triggered mode in real time systems. And they may require as-soon-as-possible delivery, meaning that they need to have a priority higher than TT traffic. Let us call EmT such Emergency event-Triggered traffic, and assume a minimal inter-arrival time between two EmT messages (or between two bursts).

As shown in [23], these messages must be set in the highest priority queue, $Q\#7$, whose gate is always open. And TT queues are placed just bellow, in queues $Q\#6$, $Q\#5$... But an EmT frame maythen use a part of a TT slot, postpone the TT frames and break the TT schedule.

One solution would be to enlarge *every* TT slot to provision a possible burst of EmT frames. However, this may lead to a very high over-reservation.

Another solution consists in enhancing TAS behavior by *dynamically* enlarging TT slots in case of EmT frames [23].

Our claim is that such a feature can be implemented without any modification to the standard, but just by a modification of the algorithms building the schedule and by the use of protective gating. In fact, it is sufficient to build the schedule and the slots (using either 0GCL, FGCL, WND or FWND), as if a burst of EmT frames were present in each slot. But at run time, only the real frames will use the bandwidth.

Moreover, the computation of bounds on the latency for lower priority queues (with network calculus for example, as in [24]), can easily be updated to account correctly EmT frames. One just have to separate the interference created by TT and EmT. The EmT interference is computed using a minimal inter-arrival delay, and the TT interference is computed using the part of slots dedicated to TT flows.

Last, if a configuration devotes uses $Q\#7$ for EmT flows, one may also use it to forward synchronization messages devoted to time measurement and clock distribution [25].

5.3 Removing any protection?

In exclusive gating, the gate mechanism is used for two purposes: the opening/closing of a TT queue is used to schedule the TT frames within slots, and the opposite closing of other queues is used to protect the slots.

In protective gating, part of the protection given by gate closing is replaced by the static priority arbiter. One may wonder if the slot protection could be implemented using only static priority, to avoid the "blocking" penalty before a slot.

This solution has already been experimented in TTEthernet, and named "Shuffling" [2]. In such a case, a non-TT frame can encroach the beginning of a slot. Then, when building a schedule, each slot must be large enough to transmit the TT frames allocated to this slot, plus some non-TT perturbation. Without preemption, this effect is the maximal size of an Ethernet frame (1528 bytes, plus 20 bytes of IFG) whereas the use of preemption can decrease it to 143 bytes [26]. And comparison on latency between shuffling and "timely block" (which is equivalent to slot protection in TSN) can be found in the context of TTEthernet in [27]. The shuffling of course creates a jitter equivalent to the encroachment time.

6. CONCLUSION

One strength of TSN is its ability support both Time-Triggered (TT) and Event-Triggered (ET) flows. The implementation is usually done using "exclusive gating" where the gate mechanism is both used to implement the time slots devoted to TT frames and to protect these slots from ET frames. We propose in this paper "*protective gating*", a small modification than can offer limited gains in bandwidth for ET flows but comes at no cost, then deserving consideration. It may also open a new freedom parameter for building schedules, since over-reservation in TT slots now comes at no cost.

7. REFERENCES

[1] H. Kopetz, "Event-triggered versus time-triggered real-time systems," in *Operating Systems of the 90s and Beyond*, A. Karshmer and J. Nehmer, Eds. Springer, 1991, pp. 86–101.

[2] W. Steiner, G. Bauer, B. Hall, M. Paulitsch, and S. Varadarajan, "TTEthernet dataflow concept," in *Proc. of Eighth IEEE International Symposium on Network Computing and Applications (NCA 2009)*, july 2009, pp. 319 –322.

[3] "IEEE standard for local and metropolitan area networks – bridges and bridged networks," IEEE, IEEE Standard 802.1Q, 2018.

[4] "IEEE standard for local and metropolitan area networks – asynchronous traffic shaping," IEEE, Tech. Rep. 802.1Qcr, September 2020.

[5] "IEEE standard for local and metropolitan area networks–bridges and bridged networks–amendment 25: Enhancements for scheduled traffic," IEEE, IEEE Standard 802.1Qbv, 2015.

[6] R. Obermaisser, Ed., *Time-Triggered Communication*, ser. Embedded Systems. CRC Press, 2012.

[7] W. Steiner, "An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks," 11 2010, pp. 375–384.

[8] D. Tămaş-Selicean, P. Pop, and W. Steiner, "Synthesis of communication schedules for TTEthernet-based mixed-criticality systems," in *Proc. of the 10th Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2012)*, ACM, Ed., 2012.

[9] F. Pozo, G. Rodriguez-Navas, H. Hansson, and W. Steiner, "Smt-based synthesis of ttethernet schedules: A performance study," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2015, pp. 1–4.

[10] G. Alderisi, G. Patti, and L. L. Bello, "Introducing support for scheduled traffic over ieee audio video bridging networks," in *Proc. of the 18th IEEE Conference on Emerging Technologies Factory Automation (ETFA 2013)*, Sep. 2013, pp. 1–9.

[11] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling real-time communication in IEEE 802.1Qbv time sensitive networks," in *Proc. of the 24th Int. Conf. on Real-Time Networks and Systems (RTNS'16)*, ser. RTNS'16. New York, NY, USA: ACM, 2016, pp. 183–192.

[12] W. Steiner, "Synthesis of static communication schedules for mixed-criticality systems," in *Proc. of the 14th IEEE Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, ser. ISORCW '11. USA: IEEE, 2011, p. 11–18.

[13] F. Dürr and N. G. Nayak, "No-wait packet scheduling for ieee time-sensitive networks (tsn)," in *Proc. of the 24th Int. Conf. on Real-Time Networks and Systems*, ser. RTNS '16. New York, NY, USA: ACM, 2016.

[14] B. Houtan, M. Ashjaei, M. Daneshtalab, M. Sjödin, and S. Mubeen, "Synthesising schedules to improve qos of best-effort traffic in tsn networks," in *29th International Conference on Real-Time Networks and Systems (RTNS'21)*, April 2021. [Online]. Available: http://www.es.mdh.se/publications/6159-

[15] S. S. Craciunas, R. S. Oliver, and W. Steiner, "Formal scheduling constraints for time-sensitive networks," 2017.

[16] S. S. Craciunas and R. S. Oliver, "Combined task- and network-level scheduling for distributed time-triggered systems," *Real-Time Syst.*, vol. 52, no. 2, p. 161–200, Mar. 2016. [Online]. Available: https://doi.org/10.1007/s11241-015-9244-x

[17] R. Serna Oliver, S. S. Craciunas, and W. Steiner, "IEEE 802.1Qbv gate control list synthesis using array theory encoding," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 13–24.

[18] N. Reusch, L. Zhao, S. S. Craciunas, and P. Pop, "Window-based schedule synthesis for industrial IEEE 802.1 Qbv TSN networks," IEEE, pp. 1–4, 2020.

[19] R. Blair, "Analysis of converged network traffic using time sensitive networking (TSN)," in *Proc. of the ODVA 2018 Industry Conference*. Stone Mountain, Georgia, USA: ODVA, October 2018.

[20] B. Houtan, M. Ashjaei, M. Daneshtalab, M. Sjödin, and S. Mubeen, "Synthesising schedules to improve QoS of best-effort traffic in TSN networks," in *Proc. of the 29th Int. Conf. on Real-Time Networks and Systems (RTNS 2021)*, Nantes, France, April 2021.

[21] M. Barzegaran, N. Reusch, L. Zhao, S. S. Craciunas, and P. Pop, "Real-time guarantees for critical traffic in ieee 802.1 qbv tsn networks with unscheduled and unsynchronized end-systems," Technical University of Denmark (TUD), Tech. Rep., 2021. [Online]. Available: https://arxiv.org/abs/2105.01641

[22] H. Daigmorte and M. Boyer, "Impact on credit freeze before gate closing in cbs and gcl integration into tsn," in *Proc. of the 27th Int. Conf. on Real-Time Networks and Systems (RTNS 2019)*, Toulouse, France, November 2019.

[23] M. Kim, D. Hyeon, and J. Paek, "eTAS: enhanced time-aware shaper for supporting non-isochronous emergency traffic in time-sensitive networks," *IEEE Internet of Things Journal*, 2021.

[24] L. Zhao, P. Pop, Z. Zheng, H. Daigmorte, and M. Boyer, "Latency analysis of multiple classes of AVB traffic in TSN with standard credit behavior using network calculus," *IEEE Transactions on Industrial Electronics*, 2020.

[25] "Local and metropolitan area networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Network," IEEE, Tech. Rep. IEEE 802.1AS, 2020.

[26] D. Thiele and R. Ernst, "Formal worst-case performance analysis of time-sensitive ethernet with frame preemption," in *Proc. of 2016 IEEE 21th Conference on Emerging Technologies and Factory Automation (ETFA 2016)*, Berlin, Germany, September 6–9 2016.

[27] M. Boyer, H. Daigmorte, N. Navet, and J. Migge, "Performance impact of the interactions between time-triggered and rate-constrained transmissions in TTEthernet," in *8th European Congress on Embedded Real Time Software and Systems (ERTSS 2016)*, Toulouse, France, Jan. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01255939