Proceedings of the

13^{th} Junior Researcher Workshop on Real-Time Computing

JRWRTC 2019

https://www.irit.fr/rtns2019/jrwrtc/

Toulouse, November 6-8, 2019

INP TOULOUSE ENSEEIHT

IRIT

Preface

This volume contains the papers presented at JRWRTC'2019: Junior Researcher Workshop on Real-Time Computing 2019 held on November 6, 2019 in Toulouse.

There were 13 submissions. Each submission was reviewed by at least 3 program committee members. The committee decided to accept 13 papers.

We would like to thank all members of the PC committee, and authors, as well as Liliana Cucu-Grosjean for her support. We also would like to thank Jérôme Ermont, general chair of the RTNS conference, to give us the opportunity to organise this workshop.

Sponsors:

- RTNS 2019 conference – https://www.irit.fr/rtns2019/
- IRIT lab – https://www.irit.fr
- INP ENSEEIHT – http://www.enseeiht.fr/
- Easychair – https://easychair.org/

October 9, 2019
Amsterdam

Benjamin Rouxel
Antonio Paolillo

Program Committee

Muhammad Ali Awan	CISTER Research Center, Portugal
Nicola Capodieci	Univerisity of Modena and Reggio Emilia, Italy
Stefano Cherubin	Politecnico di Milano
Oana Hotescu	IRIT - INP, France
Tomasz Kloda	University of Modena and Reggio Emilia, Italy
Cláudio Maia	CISTER Research Center, Portugal
Yoann Marquer	INRIA Rennes - Irisa, France
Borislav Nikolic	IDA - TU Braunschweig, Germany
Antonio Paolillo	Université Libre de Bruxelles, Belgium
Xavier Poczekajlo	Université Libre de Bruxelles, Belgium
Syed Aftab Rashid	CISTER Research Center, Portugal
Juan Rivas	Université Libre de Bruxelles, Belgium
Simon Rokicki	INRIA Rennes - Irisa, France
Benjamin Rouxel	University of Amsterdam, Netherlands
Stefanos Skalistis	INRIA Rennes - Irisa, France
Aakash Soni	IRIT - INP - ECE, France
Jun Xiao	University of Amsterdam, Netherlands

Additional Reviewers

Reghenzani, Federico

Table of Contents

Exploration of Timing Anomalies on Simplistic Processor with Model-Checking

Coralie Allioux*
VERIMAG
700 Avenue centrale
Saint-Martin-d'Hères, France
coralie.allioux@grenoble-inp.org

ABSTRACT

For real-time applications, a timely answer is at least as important as a correct one. Thus, real-time applications require bounded guarantees on their wort-case execution time (WCET). During WCET analysis, a counter-intuitive phenomenon, called *timing anomaly*, can happen, in which a local shorter execution time does not imply a better global execution time. It is difficult to integrate this phenomenon in timing analysis as it cannot be predicted. In most case, handling timing anomalies greatly increases complexity which makes such analyses unscalable.

In this article, we are interested in exploring multiple examples of timing anomalies, to understand in greater details their causes and impact on a whole program, in a certain context. Are there timing anomalies? If so, how to analyse them with model-checking and determine what kind of timing anomalies are present?

Keywords

Real-time system, WCET, Timing anomaly, Model-checking

1. INTRODUCTION

For real-time systems, timing constraints must be guaranteed. The satisfaction of such requirements is even more important for critical systems, which must be highly reliable because of a critical need (like aeronautics embedded systems). The timing analysis usually relies on WCET analysis (Worst Case Execution Time) to determine upper bounds on the execution times of a program.

Hardware platform performances are continuously improving, at the cost of an increased complexity. A pipeline implements instruction-level parallelism, cache memory allows instruction with multiple timing behaviour, analysis for multicore processors have to take care of interferences, etc. Supporting all these new features make timing analysis more and more complex.

These new features are potential sources of some kind of counter-intuitive phenomena. We are interested here in *timing anomalies*, where a local shorter execution time can provide a global longer execution time. Timing anomalies interfere with timing analysis methods, and more generally reduce timing predictability. In certain cases — for instance, timing compositionality for a multicore processor [4] —, taking into consideration timing anomalies in the analysis is likely infeasible because of a prohibitive complexity. This paper aims at a better understanding of the causes and impacts of timing anomalies. More precisely, we explore the different impacts of timing anomalies coming from a single cause: a cache access.

In this article, we present a method to explore possible timing anomalies and their impacts on a given program, with model-checking. The goal is to discover other patterns of timing anomalies in order to understand in more details what kinds exist, in the reduced context of a simplistic processor. This processor consists in an in-order 4-stage pipeline with instruction and data caches. Caches are the source of uncertainty on instruction's timing execution because of the two possible response of the cache: a cache hit or a cache miss. Both cases differ by their execution time (retrieving from main memory in case of a miss takes longer than when the data is already cached). Our model simulates the different running possibilities of the program on the processor. The detection of a timing anomaly is expressed by a simple query in the model-checker verifier. This query is based on double runs: it compares timing execution of each instruction in both cases, as in [1]. We construct our model on a simplistic processor with a minimal program.

The remainder of the paper is organized as follow. The next section gives more details about the model and exposes the chosen exploration context. It explains the processor properties and types of observed timing anomalies. Section 3 presents some kind of timing anomalies, found with our model. Section 4 and 5 concludes the paper (related work, conclusion, future work).

2. EXPLORATION CONTEXT

We use model-checking to explore more quickly the existing timing anomalies and to evaluate their different possible impacts on the program execution time.

We use UPPAAL as our model-checker. Our model is split in two parts: one for the processor, the other for the instruction block under consideration. The source of timing anomalies is a dynamic event: here, the execution time behaviour of an instruction, due to a cache access. A cache access can have two different behaviours: hit or miss. We explore the different behaviours by trying if the worst case (a cache miss) can provide a better global execution time than a cache hit. It is this aspect of the *timing anomaly* phenomenon that is particularly counter-intuitive.

This paragraph gives some details on the model itself. A

*Student at ENSIMAG, supervised for this work by Claire Maiza, Jacques Combaz, Lionel Rieg, Catherine Parent-Vigouroux

Table 1: Instruction types

Instruction	Role
`nop`	Do nothing
`load`	Register Reading
`add`	Sum between two registers

UPPAAL model is a network of timed automata. Each stage of the pipeline is represented by a timed automaton. The instruction block is a passive structure, which is updated by the pipeline whenever an instruction is completed in a stage. The program can be determined statically or randomly. In UPPAAL, time is continuous whereas we need discrete time. Our model implement a new type of clock to measure time: an integer clock, inspired by [5].

To detect timing anomalies, the model runs the program twice sequentially, to explore both behaviours of a cache access (hit or miss), inspired by [1]. The UPPAAL verifier compares the execution time of each instruction between both runs of the target program. It is easier to compare execution times this way because all the information we need is present in the same run in the model-checker: the query, which detects timing anomalies, is simplified.

Model-checking is an exhaustive method, i.e. all possibilities are explored. In consequence, for a 5-instruction block with all parameters randomly chosen (instruction type, data dependencies, data and instruction cache response, etc), there are around 2 million different instruction blocks. The model-checker will execute the 2 million possibilities and will store results in a structure. For each possibility, it will compare with the existing structure traces and states, whether this is a new case. In this context, the complexity is emphasised by the exhaustive search of the model-checker. The exploration will often reach the limit of model-checking: memory exhausted. The model-checker cannot explore and store the 2 millions possibilities: it is likely infeasible.

For all these reasons, we reduce the exploration context as much as possible on the processor itself and sources of timing anomalies, as described below.

2.1 Simplistic Processor

The processor is as simplistic as possible, to observe directly simple sources of timing anomalies.

- **Instruction Types:** The different instructions are described in Table 1.
- **Pipeline:** We consider a *in-order* 4-stage pipeline, as described in Table 2. In-order means that instructions are executed in each stage of the pipeline in the same order that they are written in the program.
- **Resources:** The processor have two caches: an instruction cache and a data cache. The FETCH stage uses the instruction cache whereas EXEC_MEMORY_ACCESS stage uses the data cache. Caches access the memory by a shared bus: it implies that both caches have a common access and the memory is shared. It creates conflicts between caches to access memory. Conflicting accesses to the memory are solved based on a fixed-priority: the data cache has priority over the instruction cache. In the model, the execution time for a *cache hit is 1 cycle*. Execution time for a *cache miss is 7 cycles*. We keep the realistic range between cache hit and cache miss: for a MPPA3 processor, cache hit is 3 cycles and cache miss is 23 cycles.

Table 2: Pipeline

Stage	Role	Execution time	Concerned instruction
FETCH	Loading instruction	Hit: 1, Miss: 7	All
DECODE	Decode instruction	1	All
EXEC _MEMORY _ACCESS	Moving data between register and memory	Hit: 1, Miss: 7	`load`
EXEC	Sum between two registers	1	`add` and `nop`

2.2 Types of observed Timing Anomalies

We distinguish two different kinds of timing anomaly: *local* (*absorbed*) and *global* (*unabsorbed*) timing anomalies.

When there is a timing anomaly in the middle of a program and it is observed only on some instructions, it is a *local* timing anomaly (see example in Section 3.2).

When a timing anomaly appears somewhere in a program and spreads until the end of the program to finally impacts the global execution time, it is a *unabsorbed timing anomaly*. In this case, a better local execution time produces a worst global execution time (example in Section 3.3).

To limit the exploration in our model, we only consider timing anomalies on the data cache. Therefore, only a `load` instruction can trigger a timing anomaly in our model. With this restriction, we can still observe timing anomalies produced by the fixed-priority between instruction and data caches.

3. EXAMPLES OF TIMING ANOMALIES

These examples are verified by our model, described in the previous section. They do not correspond to an exhaustive exploration of the timing anomalies, so it is certainly possible to find other types of timing anomalies for our use case.

The examples have the same source: the fixed priority between caches. Actually, the timing anomaly depends on which stage between FETCH and EXEC_MEM_ACCESS waits to access memory. More generally, a gap of some cycles can be created by the waiting time of a stage to access memory.

We show here that other examples derived from the same timing anomaly can have a different impact on the global execution time of a program.

3.1 Notations

The figures in the next sections, which expose timing anomaly examples, use the following notations:

- Instructions: L: `load`; A: `add`
- r: register
- M: cache miss which results in a memory access
- Cache hit:
  - DC: Cache hit in the data cache
  - IC: Cache hit in the instruction cache

3.2 A Local Timing Anomaly

We present here a local timing anomaly with only four instructions, inspired by the well-known example presented in [3]. The instruction block and the state of the pipeline are shown on Figure 1. We consider both possible responses by the data cache for the first instruction: a hit (left) or a miss (right).

As shown by Figure 1, the shortest global execution time is obtained for the cache hit case. However, there is a local timing anomaly on the third instruction (blue colour): it takes one more cycle with a cache hit than a cache miss. This means that the worst-case termination time of the third instruction is obtained when the first instruction issues a hit, which is counter-intuitive at first thought.

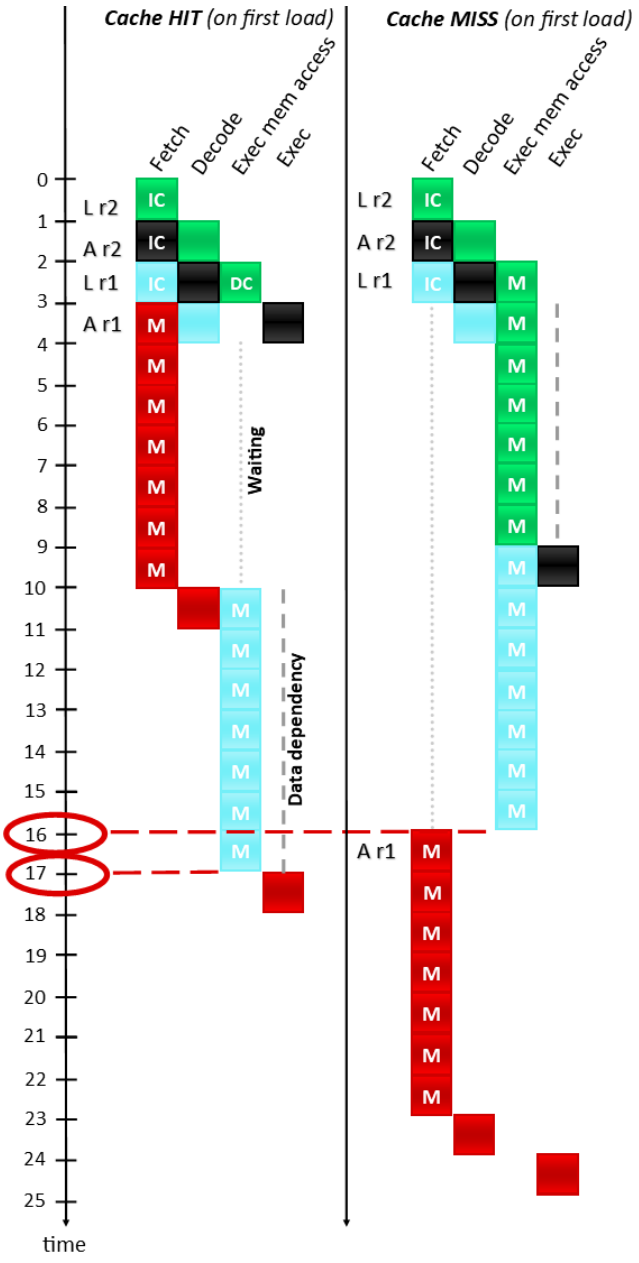


Figure 1: Example of a local timing anomaly

As we can see, on the left case, the second `load` instruction waits until the memory is free, at the 10^{th} cycle — it waits six cycles. In the miss case, it waits only five cycles: it is the instruction cache which waits until memory is free.

As said before, the source of the timing anomaly is the fixed priority between caches. The cache miss of the 1^{st} instruction (data cache, green colour) and 4^{th} instruction (instruction cache, red colour) do not request memory access at the same cycle: there is a delay of one cycle. For this reason, we can have a gap of one cycle for the termination time of the 3^{rd} instruction (blue colour): the beginning of its memory access depends of which instruction, the 1^{st} (green colour) or 4^{th} (red colour), uses memory.

In this example, the timing anomaly is only visible locally: only one instruction of the block has such an anomalous behaviour, and it sits in the middle of the block. This timing anomaly does not impact the global execution time and is absorbed as the execution goes by.

3.3 Unabsorbed Timing Anomaly

The previous timing anomaly can impact the whole execution time of a program. The example is represented on Figure 2.

Remember that such a timing anomaly is due to the presence of an initial `load` instruction. Because of the fixed-priority between the instruction cache and the data cache, it is possible to postpone the start of an instruction in the FETCH stage. This phenomenon is visible on the 6^{th} instruction (second black instruction, `add r2`), which terminates later when the first `load` has cache hit.

This delay is kept throughout the rest of the program, thanks to the pipeline which implements instruction-level parallelism. Instead, the 5^{th} instruction (second green on Figure 2) has no dependency on any other instruction. The last `add` instructions are executed at the same time than the 5^{th} instruction: the parallelism of stages EXEC and EXEC_MEMORY_ACCESS causes the *unabsorbed timing anomaly*.

This example shows that fixed priority can create a unabsorbed timing anomaly, which impacts the global execution time of the program. This is due to the gap at the FETCH stage for the 6^{th} instruction (second black): this is why the previous example only impacts locally.

4. RELATED WORK

Authors in [1] and [2] have used model-checking to detect timing anomaly. This present work have some similarities with their methods but the context and the objectives are different.

[1] develops a formal executable models for automatic detection of timing anomalies. It focuses about the construction of the model itself and how to adapt it to a given architecture. The goal is to apply this method on a real platform, applicable in a given computer architecture design. We were inspired by the method of the detection of timing anomalies in the search tree of the model-checker, which is presented in [1].

[2] uses model-checking to emphasize and validate their definition and the automatic detection method of timing anomalies. Contrary to our model, their method does not use any specific program: the identification is independent of the program.

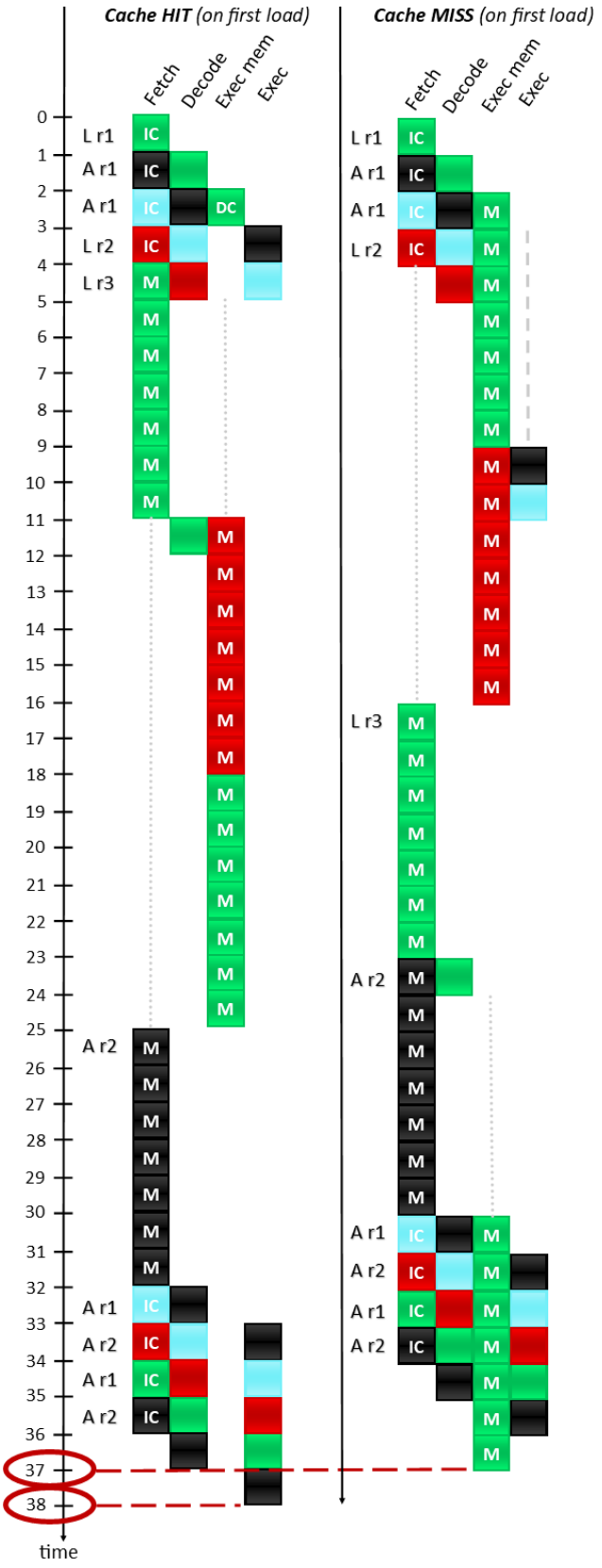


Figure 2: Example of a unabsorbed timing anomaly

However, we construct our model in order to explore the different kinds of timing anomaly for a given source on all possible program — understandable by our simplistic processor. We demonstrate that with a context reduced as much possible, we can already observe timing anomalies which impact the whole execution time of a given program.

5. CONCLUSIONS AND FUTURE WORK

Timing anomaly is a phenomenon hard to model to be taken into account in timing analysis. A possible solution to this constraint is to explore different kinds of timing anomalies and understand their impacts on the global execution time of a program. This understanding will probably offers new ideas, methods or ways to take timing anomalies into consideration on timing analysis, or specify the context where a timing anomaly has an impact on global execution time. We suggest in this article a model which explores timing anomalies for a given program, based on the model-checking method.

Our model is able to find various relevant examples of timing anomalies, even in a reduced context. More precisely, by using our model we explore impacts of timing anomaly due to fixed-priority of data and instruction cache.

Starting from a well-known example of timing anomaly, we succeed to find others situation where the same timing anomaly is unabsorbed yet, i.e., a domino effect: it impacts the global execution time of the program.

As future work, we can use this approach to search timing anomalies in a real platform — our model is adaptable and it is possible to add new features. It can also be used to verify models for WCET computation.

6. ACKNOWLEDGMENTS

I thank my supervisors — Claire Maiza, Lionel Rieg, Jacques Combaz, Catherine Parent-Vigouroux — for their participation, their support and their proofreading of this work.

7. REFERENCES

[1] M. Asavoae, B. B. Hedia, and M. Jan. Formal executable models for automatic detection of timing anomalies. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, 2018.

[2] J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Whilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *2006 IEEE Design and Diagnostics of Electronic Circuits and systems*, 2006.

[3] S. Hahn, J. Reineke, and R. Wilhelm. Toward compact abstractions for processor pipelines. In R. Meyer, A. Platzer, and H. Wehrheim, editors, *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, pages 205–220, 2015.

[4] S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015.

[5] A. S. Xiaowan Huang and S. A. Smolka. Using integer clocks to verify the timing-synch sensor network protocol. In *Proceedings of NFM 2010, April 13-15, 2010, Washington D.C, USA*, pages 79–81, 2010.

Challenges in Real-Time GPU Management*

Tanya Amert
The University of North Carolina at Chapel Hill
tamert@cs.unc.edu

ABSTRACT

Existing work on response-time analysis for real-time task systems represented as processing graphs containing cycles does not properly handle GPU-using tasks when intra-task parallelism is restricted. For such graphs to be schedulable, the execution of GPU-using tasks within the cycles may need to be prioritized over that of other tasks. This paper presents possible approaches for managing tasks using NVIDIA GPUs while allowing some tasks to be prioritized over others, and explores the tradeoff that results between response times of various tasks.

Keywords

GPUs, CUDA, real-time GPU management

1. INTRODUCTION

Advanced driver assist systems are enabled by a range of computer vision (CV) applications. These applications typically rely on cameras as an input modality. Processing of camera feeds for CV applications is greatly accelerated by graphics processing units (GPUs). Such safety-critical systems must be certified, so response-time bounds for CV applications executed on GPUs are a necessity.

However, bounding response times of tasks executed on a GPU is extremely challenging. NVIDIA's Drive-PX2 series of GPU-equipped embedded platforms are specifically designed for automotive applications, yet necessary scheduling details of NVIDIA GPUs are not available without a restrictive non-disclosure agreement, and their drivers are typically closed source. As a result, existing response-time bounds for such GPUs are not tight, and modifications to the internal scheduling policies are difficult to make. Although AMD provides open-source drivers for its GPUs, AMD is not as widely adopted by automotive companies.

Contributions. In this paper, we explore approaches to real-time GPU management that enable tighter response-time bounds for GPU-using tasks while allowing for prioritizing of some GPU-using tasks over others.

Organization. This paper is organized as follows. We provide an overview of NVIDIA GPU software and hardware in Sec. 2 and discuss related work in Sec. 3. We then explore various approaches to managing GPU execution in Sec. 4, and conclude in Sec. 5.

*Work supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors.

Listing 1 Vector Addition Pseudocode.

```
1: kernel vecAdd(A ptr to int, B: ptr to int, C: ptr to int)
       // Calculate index using built-in thread, block info
2:     i := blockDim.x * blockIdx.x + threadIdx.x
3:     C[i] := A[i] + B[i]
4: end kernel

5: procedure main
       // (i) Allocate GPU memory for arrays A, B, and C
6:     cudaMalloc(d_A)
7:     ...
       // (ii) Copy arrays A and B from CPU to GPU memory
8:     cudaMemcpy(d_A, h_A)
9:     ...
       // (iii) Launch the kernel
10:    vecAdd<<<numBlocks, threadsPerBlock>>>(
           d_A, d_B, d_C)
       // (iv) Copy results from GPU to CPU array C
11:    cudaMemcpy(h_C, d_C)
       // (v) Free GPU memory for arrays A, B, and C
12:    cudaFree(d_A)
13:    ...
14: end procedure
```

2. BACKGROUND

In this section, we provide an overview of the basics of NVIDIA GPUs.

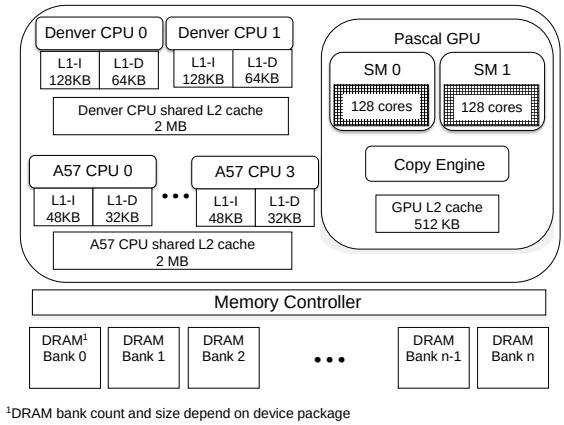
2.1 CUDA

CUDA is a C/C++ extension developed by NVIDIA to allow programmers to submit instructions to the GPU [8]. The CUDA API includes commands to copy data between the host CPU and the device GPU, as well as between multiple GPU devices, and to submit programs called *kernels* for execution on the GPU.

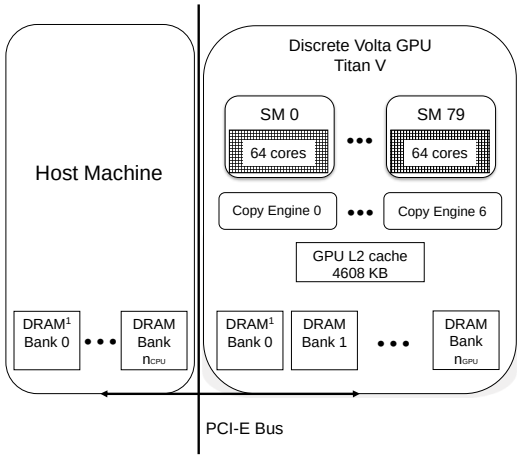
The general structure of a CUDA program is illustrated in Listing 1. This includes (i) allocating memory on the GPU, (ii) copying data to the GPU, (iii) executing one or more kernels on the GPU, (iv) copying the results back from the GPU, and (v) freeing any allocated memory on the GPU.

Kernel execution is divided into groups of 32 *threads*, called *warps*. Each thread in a warp executes in a SIMD fashion, using built-in CUDA variables to determine the data for that thread (Line 2). Warps are combined into *blocks*, which are further grouped into *grids*. As shown in Line 10, the command to issue a kernel requires the programmer to specify the layout of threads into grids and blocks (`numBlocks` and `threadsPerBlock`, respectively).

Each CUDA commands is issued to a *stream*, which is a FIFO queue of operations. By default, all CUDA commands are issued to the default stream, called the *NULL stream*.



(a) NVIDIA TX2



(b) NVIDIA Titan V

Figure 1: Comparison of two GPUs: (a) NVIDIA TX2 (integrated) and (b) Titan V (discrete).

Use of the NULL stream greatly reduces parallelism, and user-defined streams can be used to allow operations to execute in parallel, and thus better utilize powerful GPUs.

2.2 NVIDIA GPU Hardware

NVIDIA GPUs contain multiple copy engines (CEs), used to copy data to and from the GPU, and an execution engine (EE), comprised of multiple *streaming multiprocessors* (SMs). An SM can service up to 2048 threads at once. Integrated GPUs typically have an order of magnitude fewer SMs than discrete GPUs.

Example 1. *Details of two NVIDIA GPUs are depicted in Fig. 1. NVIDIA's TX2 system-on-chip (inset (a)) has a GPU comprised of one CE and two SMs. In contrast, the discrete Titan V (inset (b)) boasts seven CEs and 80 SMs.*

An SM is comprised of a number of hardware cores, serviced by a small number of warp schedulers. A warp scheduler hides memory latency by determining which of four warps should execute an instruction at a given time instant. For the rest of this paper, we consider scheduling at the thread and block level, and ignore warps, as their details are much more difficult to measure at runtime.

3. RELATED WORK

In this section, we discuss three groups of related work. In doing so, we consider the following different dimensions of GPU execution: whether the GPU is treated as a uniprocessor or multiprocessor device, if the approach uses locking protocols to synchronize GPU access or considers scheduling techniques, whether the GPU is accessed by one or multiple processes, and if one or multiple GPUs are available in a system. We consider related work first based on the choice of scheduling or synchronization as a management technique, and discuss the remaining dimensions later.

Scheduling. The NVIDIA documentation omits important details regarding GPU scheduling policies. To this end, Otterness *et al.* explored the behavior of kernels submitted from different processes (thus executed via timeslices) [9]. Amert *et al.* extended this exploration to GPU access from within a single process [1]; they described a set of rules dictating the order in which GPU operations (kernel executes or copy operations) execute on their respective GPU engines, and provided microbenchmark experiments to validate this behavior. They additionally studied the behavior when both user-defined streams and the NULL stream are used, as well as two different stream priority levels. Recently, Yang *et al.* extended this work to provide response-time bounds for GPU-using CV applications expressed as DAGs [10].

Capodieci *et al.* demonstrated a real-time scheduler for GPU operations [3]. They implemented a software scheduler module within the NVIDIA hypervisor's runlist manager, enabling preemptive earliest-deadline first (EDF) scheduling on the Drive-PX2. However, their approach requires the open-source drivers available only for NVIDIA's Drive-PX2 embedded platform, and some details and their source code were not made available due to non-disclosure agreements.

Synchronization. Access to the GPU can alternatively be arbitrated by a locking protocol. This approach was taken in developing GPUSync [4], a real-time GPU management framework. GPUSync utilizes k-exclusion locks to allow mutually exclusive access to a number of GPUs in a system. Implemented in the kernel, GPUSync is available as a fork [5] of the LITMUS$^{\mathrm{RT}}$ kernel [6].

Remaining dimensions. Of the described related work, both GPUSync and Capodieci *et al.* treat the GPU as a uniprocessor. For less-powerful embedded GPUs, this is a reasonable assumption. However, computations performed on a powerful GPU such as the Titan V might not require all of the GPU's many SMs, resulting in wasted GPU capacity. In that case, the multiprocessor-scheduling-based approaches of [1, 9, 10] might be more appropriate.

Simultaneous GPU access from different processes can greatly impact response times [9]. For discrete GPUs, the NVIDIA Multi-Process Service (MPS) allows the scheduling rules detailed in [1] to apply even when different processes access the GPU; without MPS, kernels submitted from different processes execute in timeslices based on the runlist, and thus do not truly execute concurrently on the GPU.

The above related work can also be applied for systems with multiple GPUs. The CUDA API allows the programmer to specify to which GPU an operation is submitted. The EDF approach from [3] applies if tasks are partitioned to the different GPUs, and GPUSync directly allows for multiple GPUs via its k-exclusion locking protocol to manage access.

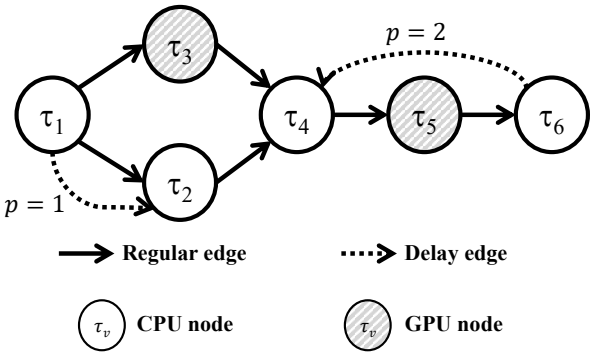


Figure 2: A sporadic task graph with CPU and GPU nodes and one cycle.

4. REAL-TIME GPU MANAGEMENT

In this section, we formalize our problem and present potential approaches to real-time GPU management that prioritize tasks within cycles.

4.1 Motivation

Processing graphs can be used to represent CV applications. Nodes represent tasks that perform some computation, and edges correspond to the data dependencies between tasks. *Delay edges* indicate that the data involves results from a prior time step (in CV applications, a time step typically corresponds to a video frame used as inupt). Corresponding to each delay edge is a value p indicating the age of the historical data dependency.

EXAMPLE 2. *An example graph is depicted in Fig. 2. The edges from τ_1 to τ_2 indicate that a job of τ_2 uses the outputs of the jobs of τ_1 from the current and previous time steps.*

Cycles induced by the presence of delay edges can wreak havoc on existing real-time analysis. If some delay edge in a cycle has $p = 1$, then some task in that cycle depends on the output of the prior frame; as a result, no two jobs of any task in that cycle can possibly execute in parallel.

EXAMPLE 2. (cont'd) *In Fig. 2, the delay edge from τ_6 to τ_4 results in a cycle. As $p = 2$ for that edge, no more than two jobs of any task in $\{\tau_4, \tau_5, \tau_6\}$ can execute in parallel.*

The implicit-deadline sporadic task model is given as $\tau_i = (\Phi_i, T_i, C_i)$, where Φ_i, T_i, and C_i represent the task's phase offset, period, and worst-case execution time, respectively. The utilization of task τ_i is defined as $u_i = C_i/T_i$.

Assume multiple jobs of a task may execute concurrently. If the utilization of a cycle is greater than 1.0, then jobs of tasks in that cycle cannot be scheduled sequentially without incurring unbounded response times; if u_{cycle} is the utilization of a cycle in the graph, then at least $\lceil u_{cycle} \rceil$ jobs of each task in the cycle must be allowed to execute simultaneously. However, the data dependency introduced by a cycle requires that no more than p jobs of a given task may execute at once. Thus, tasks have *restricted intra-task parallelism.*

In an upcoming paper, Amert *et al.* [2] propose a new *rp-sporadic task model* and provide the corresponding analysis for sporadic task graphs containing cycles with utilization greater than 1.0. The rp-sporadic task model encodes the necessary intra-task parallelism as an additional task parameter: $\tau_i = (\Phi_i, T_i, C_i, P_i)$. In this model, P_i is the maximum possible *intra-task parallelism* for the task.

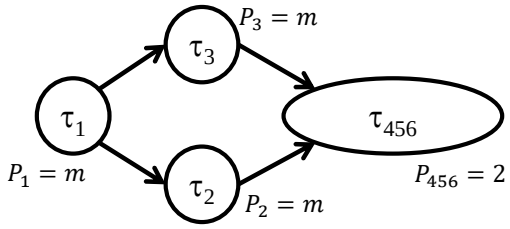


Figure 3: The sporadic task DAG corresponding to the graph in Fig. 2.

4.2 Proposed Approaches

The analysis presented in [2] reveals a circularity: to determine the response-time bound for CPU tasks, it is necessary to know the response-times of the GPU tasks, and vice versa. The GPU response-time analysis in [10] assumes full intra-task parallelism ($P_i = \infty$) for GPU tasks, so it does not apply to the rp-sporadic task model. Instead, Amert *et al.* break this circularity by assuming that access to the GPU is arbitrated with a simple FIFO mutex lock, and thus treat GPU blocking time as CPU execution time. They also combine cycles into *supernodes*; the resulting DAG corresponding to the graph from Fig. 2 is shown in Fig. 3. However, the response-time bounds derived in [2] are conservative for powerful GPUs such as the Titan V.

We now explore real-time GPU management approaches that enable more precise accounting of response times of GPU tasks under the rp-sporadic task model, and aim specifically to minimize the response times for GPU nodes that are part of cycles. We first assume each approach is taken individually; later we remove this assumption.

As in the graph in Fig. 2, we assume that each task executes on either the CPU or the GPU. For our purposes, we consider only the GPU tasks, which we divide into two sets: $\mathcal{H}$ and $\mathcal{L}$, corresponding to nodes that are (high priority) and are not (low priority) part of a cycle, respectively.

EXAMPLE 2. (cont'd) *In Fig. 2, $\mathcal{H} = \{\tau_5\}$ and $\mathcal{L} = \{\tau_3\}$.*

Multiple GPUs. In a system with many GPUs, each cycle can be assigned to its own GPU. If GPUs are not so abundant, a subset of the GPUs can be designated for use only by tasks in $\mathcal{H}$, with access arbitrated by a k-exclusion locking protocol, as in GPUSync [4]. If some task in $\mathcal{H}$ may access more than one GPU simultaneously, a locking protocol for *replicated resources* [7] can instead be used to manage access. This might be the case if multiple GPU operations are issued by a single task.

Multi-Process Service. In addition to allowing different processes' kernels to execute according to the rules in [1], MPS allows the programmer to specify a fraction of the GPU available to a given process.[1] Thus, MPS can be used to divide a GPU into two or more "virtual GPUs" (vGPUs); the scheduling rules of [1] apply to each vGPU.

EXAMPLE 3. *An example execution pattern is shown in Fig. 4. In this experiment, two processes each submitted a single kernel to a Titan V. Under MPS, each process was specified to utilize 40% of the GPU, i.e. 32 of the 80 total SMs. Each process submitted a kernel comprised of 160*

[1]MPS can only be used on discrete GPUs, and thus is not available for integrated GPUs, such as on NVIDIA's TX2 or Drive-PX2.

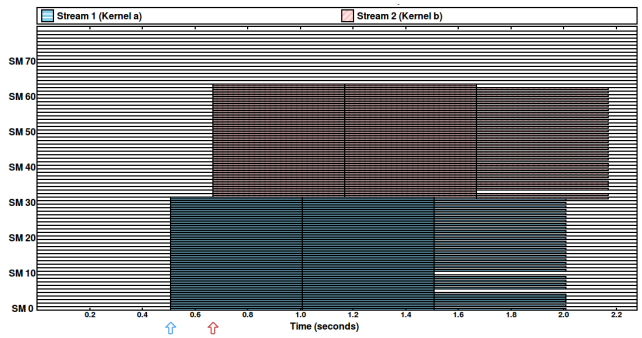


Figure 4: Two processes, each using 40% of the GPU (all of the Titan V's 80 SMs shown).

blocks of 1024 threads each (exactly enough to utilize the entire GPU). Both kernels were able to run immediately using their 40% share, and both took three times as long to complete compared to their runtime in isolation. (Without MPS, one kernel would utilize the entire GPU, and the second would start upon completion of blocks of the first.)

As demonstrated in Ex. 3, there is a tradeoff between the response time of a task and the fraction of the GPU it can utilize. One vGPU could service all tasks in $\mathcal{L}$, while the rest service tasks in $\mathcal{H}$. Alternatively, multiple processes could be configured with different GPU fractions, acting as servers of different execution power available to tasks in $\mathcal{H}$.

Stream priorities. One of the scheduling extensions explored in [1] is stream priorities. On existing NVIDIA GPUs, there are two priority levels for user-defined streams: priority-low (the default) and priority-high. As described by Amert *et al.*, the EE uses one scheduling queue per priority level. Rule A2 from [1] states that if a kernel is present at the head of the priority-high EE queue, then blocks from that kernel may execute on the EE. Otherwise, blocks of the kernel at the head of the priority-low EE queue, if any, may execute.

Example 4. *Fig. 5 depicts an experiment showing the starvation of a kernel submitted to a priority-low stream (Fig. 6 in [1], reproduced on a Titan V with only the first 8 SMs depicted for clarity). All 640 blocks of each of kernels K2 and K3 complete execution before the last 160 blocks of K1 execute. The execution of K3 is only delayed by the execution of K2, another kernel in a priority-high stream.*

Tasks in $\mathcal{H}$ could submit kernels only to priority-high streams, and tasks in $\mathcal{L}$ could submit kernels only to priority-low streams. Therefore, tasks in $\mathcal{H}$ would be delayed only by other tasks in $\mathcal{H}$ plus at most the longest block duration of any kernel submitted by a task in $\mathcal{L}$.

4.3 Combining Approaches

The approaches mentioned above can be combined. For example, in a system with multiple GPUs, allowing only tasks from a single cycle to execute on a GPU reduces blocking of those tasks, but might greatly underutilize the GPU. Instead, the prioritized stream approach could be used to allow tasks in $\mathcal{L}$ with short block durations to better utilize the GPU with only minimal delay for the tasks in $\mathcal{H}$ assigned to that GPU. Similarly, MPS and stream priorities could be combined to enable tasks in both $\mathcal{H}$ and $\mathcal{L}$ to execute on the same vGPU. The delay to tasks in $\mathcal{H}$ introduced by lower-priority jobs when using prioritized streams

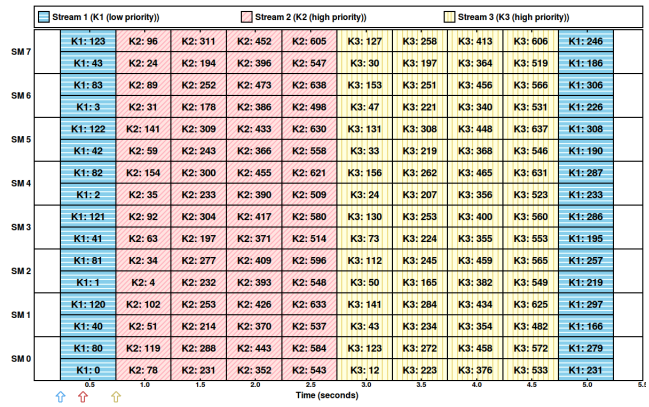


Figure 5: Kernels in priority-high streams can starve kernels in priority-low streams (truncated to show only the first 8 SMs).

suggests a design tradeoff between block duration and the number of blocks needed to complete computations.

5. CONCLUSION

In this paper, we presented multiple potential approaches for real-time GPU management, with a focus on prioritizing the operations of some GPU-using tasks over others. These techniques can enable tighter response-time bounds for GPU workloads. A deeper exploration of each approach, as well as combinations, is necessary in the future, with a case study of real automotive applications.

6. REFERENCES

[1] T. Amert, N. Otterness, M. Yang, J. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *RTSS '17*.

[2] T. Amert, S. Voronov, and J. Anderson, "OpenVX and real-time certification: the troublesome history," in *RTSS '19, to appear*.

[3] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *RTSS '18*.

[4] G. A. Elliott, "Real-time scheduling of GPUs, with applications in advanced automotive systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2015.

[5] GPUSync Project, https://www.github.com/GElliott/litmus-rt-gpusync/.

[6] LITMUS^{RT} Project, https://www.litmus-rt.org/.

[7] C. Nemitz, K. Yang, M. Yang, P. Ekberg, and J. Anderson, "Multiprocessor real-time locking protocols for replicated reources," in *ECRTS '16*.

[8] NVIDIA, "CUDA toolkit documentation v10.1.243," Online at http://docs.nvidia.com/cuda/, 2019.

[9] N. Otterness, M. Yang, S. Rust, E. Park, J. Anderson, F. Smith, A. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *RTAS '17*.

[10] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, "Making OpenVX really 'real time'," in *RTSS '18*.

PLRU Cache Analysis

Zhenyu Bai
Univ. Grenoble Alpes, CNRS,
Grenoble INP, VERIMAG,
38000 Grenoble, France
zhenyu.bai@univ-greoble-alpes.fr

David Monniaux
Univ. Grenoble Alpes, CNRS,
Grenoble INP, VERIMAG,
38000 Grenoble, France
David.Monniaux@univ-grenoble-alpes.fr

Claire Maïza
Univ. Grenoble Alpes, CNRS,
Grenoble INP, VERIMAG,
38000 Grenoble, France
claire.maiza@univ-grenoble-alpes.fr

ABSTRACT

Memory reads have very different latency whether they hit the cache or they need to go to main memory (DRAM). This has to be taken into account for worst-case execution time analysis. Caches are characterized by several parameters, including their replacement policy.

The Pseudo Least-Recently Used (PLRU) policy is widely used in industry. It is however difficult to statically predict which accesses are hits or misses with this policy, compared to the Least-Recently Used (LRU) policy. Previous analyses tend to be expensive or to lack precision—that is, they tend to fail to classify accesses that could be classified as "always hit" or "always miss'.

In this paper, we present a new canonical representation of PLRU cache state and we show a new PLRU cache analysis with better balance between precision and scalability. Our analysis is based on abstract interpretation; we show its soundness and its performance on realistic experiments.

Keywords

Cache Analysis, Abstract Interpretation, Static Analysis, PLRU cache

1. INTRODUCTION

The latency of accesses to DRAM-based main memory has long been much higher than that of basic operations on CPUs. A common solution to bridge this gap is to include one or more cache memories between the CPU and main memory, meant to store frequently accessed code and data. These cache memories respond much faster than main memory.

The certification of safety-critical real-time applications often demands to bound a program's *worst-case execution time* (WCET). In architectures involving caches, the latency of an individual memory access may vary considerably depending on whether the access is a *cache hit* (the data is retrieved from the cache) or a *cache miss* (the data is not in the cache and has to be fetched from the next-level cache or from DRAM). Obviously, a higher memory access latency may lead to a higher execution time. More complex effects may exist: on certain systems, due to *timing anomalies* (cache Hit may cause worst execution time), a lower memory access latency can translate to a higher execution time. Also, uncertainty about whether an access is a hit or a miss forces the static analysis of the microarchitecture to follow both cases, leading to higher analysis costs. It is therefore important to classify accesses that always hit or always miss the cache.

In this paper, we focus on instruction caches—an extension to data caches could be devised by prepending a pointer analysis and dealing with the writing policy, but this is out of our scope. Instruction and data caches are generally set-associative: each memory access is mapped to exactly one cache set, determined by a simple computation from its address. For most policies, including PLRU, the behavior of each cache set is independent. Therefore, we construct our analysis by focusing on a single cache set. Each cache set consists of several cache lines. After a cache miss, the memory block is loaded into the cache set that it maps to; if the cache set is already full, one of its K cache lines is evicted. The evicted line is determined by the replacement policy, in our case, the PLRU.

There exist several analyses for statically predicting which accesses to a PLRU cache are hits or misses; they however tend to be imprecise (many accesses classified as "unknown" when they could be classified as "always hit" or "always miss") and/or scale poorly, due to combinatorial explosion.[1] In this paper, we propose a static analysis approach based on abstract interpretation that is scalable and yet conserves reasonable precision.

2. PLRU REPLACEMENT BEHAVIOR

Pseudo-LRU (LRU) replacement policy is a tree-based approximation of the Least Recent Used (LRU) policy. It arranges the cache lines at the leaves of a tree with *tree bits* pointing to the line to be evicted next; a 0 indicates pointing to the left sub-tree, a 1 indicates pointing to the right sub-tree. After every access, all tree bits on the path from the root to the accessed or line are set to point away from the line. Other tree bits are left unchanged.

PLRU policy has been shown to be less predictable than LRU[7], specially due to the following properties:

- a block is not guaranteed to be evicted after K distinct accesses of new blocks;
- a block is no longer guaranteed to be cached after $log_2(K) + 1$ accesses.

Age-based analyses track for each block possible values for its "age", that is, how far it is from being the most recent access, or, equivalently, how many newer values have been

[1]This contrasts with LRU, for which there exist several analyses [3], including highly precise ones [8]. This difference between LRU and PLRU has been characterized from the point of view of complexity theory [6].

loaded into the cache after it. Such analyses are widely used for LRU caches [3, 8]. They however cannot be used for showing that some block has been evicted from a PLRU cache, due to the possibility of infinite survival. In addition, it is possible to show that a block a must be in the cache only if at most $log_2(k)$ accesses to other blocks have been occurred since the last access to a. For larger associativities this is very imprecise. Hence, we want a new analysis avoiding those limitations.

3. PLRU REPLACEMENT BEHAVIOR: FROM PHYSICAL STATE TO LOGICAL STATE

3.1 Logical cache state

Simple canonical representations can be naturally found for LRU caches and FIFO caches, by ordering cached blocks from last-in to first-in for FIFO, or from most-recently used to least-recently used for LRU.[2]

For PLRU, to each cache state we associate a canonical representation, that we call *uniform index*, by rotating simultaneously the tree bits and the subtrees so that all arrows point leftwards to the same block as before, and thus all tree bits are 0. This canonical representation stands for K equivalent states.

Since all tree bits are now 0, we can discard them. The state of the cache is now defined by the sequence of K line contents from left to right. To each block we can associate a *position* in the cache, in $\{0, ..., K-1\}$ that we use as our concrete state.

3.2 Updating the concrete state

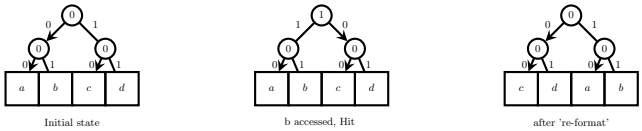


Figure 1: Tree-bits changes and re-formatting upon hit

Fig.1 shows how the concrete state is affected by memory accesses. For a given canonical state, upon cache hit, tree bits may change and hence make the state no longer canonical; then we re-format the new state to make it canonical.

In the case of hit, the contents are unchanged, and the changes of the tree bits and reformatting amount to a permutation of the positions of the block in the cache. In fact, in the case of Hit, this permutation of *position* is static and depends only on the *position* of the accessed cache line. In the example of Fig.1, the initial canonical state before access can be written as:

$$[a, b, c, d] \text{ or } a \to 0; b \to 1; c \to 2; d \to 3$$

and after access, the canonical state is

$$[c, d, a, b] \text{ or } a \to 2; b \to 3; c \to 0; d \to 1$$

[2]The actual hardware implementation may work differently, e.g. for FIFO as a circular buffer, leaving blocks in the same hardware memories and moving a pointer to the current block to be evicted. This is immaterial to our reasoning.

In the case of miss, it is not exactly a permutation : tree bits changes are the same as for a hit on *position* 0, but the block at *position* 0 is evicted and replaced by the newly loaded block. We compare in the example below, the update upon cache Hit on *position* 0 and upon cache Miss.

$$[a, b, c, d] \xrightarrow{\text{upon access a, Hit}} [c, d, b, a]$$
$$[a, b, c, d] \xrightarrow{\text{upon access e, Miss}} [b, c, d, e]$$

4. CACHE ANALYSIS - ABSTRACT INTERPRETATION

4.1 Abstract Interpretation framework

To classify a memory access as "always miss" (respectively, "always hit"), we must prove that the cache state along all feasible execution paths before this access results in a miss (respectively, a hit) upon this access. Enumerating all such paths is in general intractable. Enumerating all reachable cache states at a given location is also quickly intractable, as we shall see. Most cache analyses thus use some form of abstraction of the set of reachable cache states.

Abstract interpretation, a widely used approach in static analysis, provides a framework for abstraction. In this framework one has a *concrete domain* $\mathcal{D}_{conc}$, i.e., what we want to abstract, here the cache states, an *abstract domain*, $\mathcal{D}_{abs}$, i.e., the abstract values upon which the analysis computes, and for each operation of the system, here each possible memory access, a suitable transformer for the abstract value. These transformers must be *sound*, meaning that if an abstract value represents a concrete state that can lead to another concrete state by the operation, then the abstract value produced by the transformer should represent that other concrete states. In other words, we never ignore feasible behaviors of the program.

While our analysis is sound, it is not exact, as in most cases in abstract interpretation.[3] This means the analysis may fail to classify as "always hit" (respectively, "always miss") accesses that hit (respectively, miss) in all executions.

4.2 Abstract Interpretation in cache analysis

We assume that we are given a Control-Flow Graph (CFG) containing the control-flow and the distribution of memory accesses into cache sets. This CFG is obtained by loading the program with known addresses, and performing pointer analysis for indirect branches, including returns from procedure calls.[4]

This CFG is an abstraction of the original program. We completely lose the semantics of the program instructions (arithmetic, conditional branches); thus the CFG may exhibit behaviors that cannot occur in the original program (e.g., taking a $x < 0$ branch then a $x > 0$ branch for the same x). This is a common approach in cache analysis. It is in general impossible to include a faithful account of program semantics in analysis, since doing so entails solving Turing's halting problem. Partially accounting for unfeasible paths is possible, but is outside of the scope of this paper.

[3]For LRU there exists one scalable exact analysis [8], but again LRU has good properties not shared by PLRU.

[4]In the case of data caches, one needs pointer analysis for all values.

We also do not take the call stack into account, so calls to the same procedure will be mixed in the semantics. It is possible to refine this by inlining procedures subgraphs at the point of call, in the case of bounded call stacks, which is in general the case for safety-critical systems.

5. COLLECTING SEMANTICS

The simplest solution for cache analysis is to compute the set of all reachable concrete cache states at each program point, called *Collecting Semantics*. Thus, the domain of collecting semantics $\mathcal{D}_{coll}$ is a set of concrete state. Following abstract interpretation, we define first the concrete domain, the *collecting semantics*:

- to each program point we attach the set of reachable cache states at that point, the domain of this set is called $\mathcal{D}_{coll}$;
- upon a memory access, we update each possible state of the set by the rules of PLRU policy, this transformer is called $Update_{coll}$;
- at a control point with several incoming edges, we keep the disjoint union of all possible cache state; this transformer is called $Join_{coll}$.

Assuming the program starts with an empty cache, we attach the set $\emptyset$ to the program starting location, meaning that the only possible cache state at this location is empty. Other locations receive $\emptyset$, meaning they are so far considered to be unreachable.

We then saturate these sets as follows. We maintain a working set of locations to be updated, initially containing only the immediate successors of the initial control location. Then, as long as this set is nonempty, we pick a location from it, and replace the set of states corresponding to that location by the union of the images of the sets of states assigned to its predecessors by the suitable permutations and replacement. If the set at the location is changed from its previous value, then the successors are added to the working set.

In the end, we have computed for each location the set of reachable cache states. Unfortunately such an analysis, akin to explicit-state model-checking, suffers from combinatorial explosion, as we shall see later. The goal of our abstraction is hence to find a good way to abstract the set of concrete states and find good transformers corresponding to $Update_{coll}$ and $Join_{coll}$.

6. OUR ANALYSIS

We abstract a set of concrete states by all possible positions of each block in the canonical representation.

6.1 Abstract Domain

Formally, the abstract domain is defined as:

$$\mathcal{D}_{abs} : \mathbb{B} \to \mathcal{P}(\{0, .., K-1, \epsilon\})$$

Where $\mathcal{P}$ stands for *power set* and $\mathbb{B}$ stands for basic block (i.e. an identifier in $\mathbb{N}$). The $\{0, .., K-1, \epsilon\}$ corresponds to the *positions* in the canonical representation, with ϵ meaning 'not cached', to distinguish 'a block can only be at position p' from 'a block can be at position p or not cached'. $\{\epsilon\}$ means that a block is for sure not in the cache.

The abstraction is defined by its concretization function:

$$\gamma : \mathcal{D}_{abs} \to \mathcal{D}_{coll}$$
$$\gamma(q^{\#}) = \{pb \in \mathbb{B} \to \{0, .., K-1, \epsilon\} | \forall b \in \mathbb{B},$$
$$pb(b) \neq \epsilon \Rightarrow pb(b) \in q^{\#}(b) \wedge pb(b) = \epsilon \Rightarrow \epsilon \in q^{\#}(b)\}$$

We show below an example of abstracting from collecting semantics then concretizing it to collecting semantics:

{[a,b,c,d], [e,f,c,d]}
⇓ abstraction
$a \to \{0, \epsilon\}; b \to \{1, \epsilon\}; c \to \{2\};$
$d \to \{3\}; e \to \{0, \epsilon\}; f \to \{1, \epsilon\}.$
⇓ concretization
{[a,b,c,d], [e,f,c,d], [a,f,c,d], [e,b,c,d]}

This example also shows that our abstraction is not exact: it overapproximates (soundly) certain sets of reachable states.

6.2 Abstract Transformers

Following the Abstract Interpretation framework, we also define the transformers corresponding to those for collecting semantics : the $Update_{coll}$ upon memory access, and the $Join_{coll}$ when control flows from several locations to a single one.

6.2.1 Join transformer: $Join_{abs}$

Similar to collecting semantics, we keep the union of possible positions. Formally, the $Join_{abs}$ transformer is defined as:

$$Join_{abs} : \mathcal{D}_{abs} \times \mathcal{D}_{abs} \to \mathcal{D}_{abs}$$
$$Join_{abs} = \lambda b.s1(b) \cup s2(b)$$

The $Join_{abs}$ gives also an over-estimation of $Join_{coll}$ i.e. $\gamma \circ Join_{abs} \subset Join_{coll} \circ \gamma$. However, $Join_{abs}$ is sound.

6.2.2 Update transformer: $Update_{abs}$

As stated in Sec.3, the permutation of position is static. Hence, we can compute the new cache state position by position. Upon cache Hit, all possible blocks at the position x will be at position y where x and y depends only on the accessed position. We define the function representing this permutation $\sigma \to \{0, ..., K-1\} \to \{0, ..., K-1, \epsilon\} \to \{0, ..., K-1, \epsilon\}$ takes the accessed position and the position before update then determinate the position after update.

Upon cache miss, tree bits changes are the same as Hit at eviction position, but the block at eviction position is evicted and replaced by the new block. We use a function $\sigma_{Miss} : \{0, ..., k-1, \epsilon\} \to \{0, ..., k-1, \epsilon\}$ to represent the permutation of each position, in the case of miss, the hit position is no longer needed since the permutation is the same as hit on position 0.

$Update_{abs}$ consists in applying σ to each possible position of each block. It's defined formally by :

$$update_{abs} : \mathcal{D}_{abs} \times \mathbb{B} \to \mathcal{D}_{abs}$$
$$update_{abs}(q^{\#})(b) = \lambda x. \bigcup_{i \in q^{\#}(b)} \sigma(i)(q^{\#}(x))$$

In practice, a block that can be at $x_1, x_2, x_3, ..$ will be at y_1, y_2, y_3 where x_i and y_i are determined statically, that means the update of all possible position of a block can be pre-calculated, which accelerates considerably the analysis. In the example below, since b could be at 1 or ϵ before update, we apply $\sigma_{Hit,1}$ and σ_{Miss} respectively to each position.

$a \to \{0, \epsilon\}; b \to \{1, \epsilon\}; c \to \{2\};$
$d \to \{3\}; e \to \{0, \epsilon\}; f \to \{1, \epsilon\}.$
$\Downarrow$ access b
$a \to \{2, \epsilon\}; b \to \{3\}; c \to \{0, 1\};$
$d \to \{2, 1\}; e \to \{2, \epsilon\}; f \to \{0, \epsilon\}.$

7. EVALUATION

Both abstract analysis and collecting semantics are implemented with OTAWA [1]. The binaries of *TACLe Benchmarks* are compiled for a small ARM processor. The memory block is fixed to 32 bytes and the associativity is 8, the total cache size hence depends on the number of cache set.

In Fig.2, we compare the analysis time of both type of analysis. A timeout of 2 hours has been set and considered as reasonable [5]. In case of one cache set and small programs, we can see that the performance of collecting semantics is similar to abstract analysis. However, for big programs and 2 or 8 cache sets, collecting semantics fails to finish in 2 hours, whereas abstract analysis accomplishes it; for medium one, analysis time of collecting semantics is exponential to abstract analysis (The y-axis has logarithmic scale).

In Fig.3, we compare the percentage of classification (Always Hit (AH), Always Miss (AM) and Not Classified (NC)) of both collecting semantics and abstract analysis with one cache set. We have no complete data for some benchmarks since the corresponding analysis has not finished in 2 hours. We have observed generally 10% to 200% more Not Classified which needs more precise study to identify the influence factor. However, this lose of precision is acceptable in the most of the cases compared to the gain on the performance and we consider adding a partial-concretization style reduction before *Update* stage which may reduce the global lose of precision.

We have observed a general speed-up and up to 100 times faster compared to [4] on the *Malardalen* benchmarks even binaries are compiled for a small ARM architecture instead of MIPS in [4]. However, we have not compared our analysis to existing ones [5, 4, 2]due to the different target architecture. More generally, we have not found enough of implementation details of existing analyses to be able to compare with ours.

We have implemented our collecting semantics to evaluate our analysis and to compare with existing implementation of collecting semantics used as reference of their abstract analysis respectively. Since the same strategy i.e. collecting semantics is implemented, the relative complexity of analysis for some programs (referred by the relative ratio of analysis time) should be similar. While the experimental result confirms an incomparability: our implementation of collecting semantics shows different complexities for a giving benchmarks set compared to existing ones.

As conclusion, Our analysis is much faster than collecting semantics on realistic parameters and seems to be generally faster than existing PLRU analyses. However, experimentation leads to incomparable results for the same set of benchmarks (some better and some worst results). As future work, we aim at a better understanding of this comparison.

[5]We consider 2 hours the limit of reasonable for a desktop PC of 8G RAM and a *i5* processor

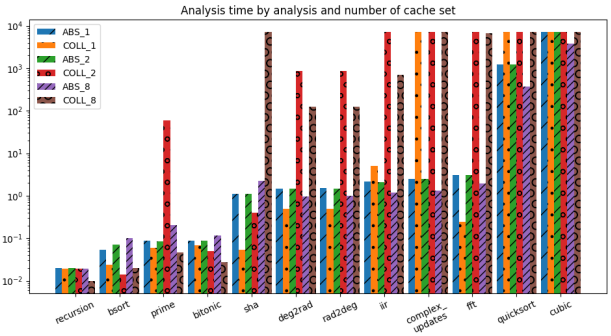


Figure 2: analysis time (in second) by analysis type (COLL as collecting semantics or ABS as abstract analysis), and number of cache set (1,2 and 8)

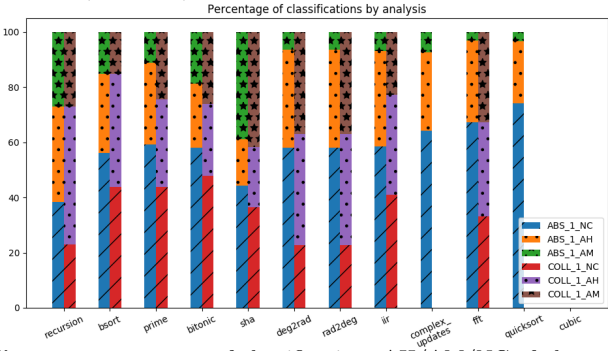


Figure 3: percentage of classifications AH/AM/NC of abstract analysis and collecting semantics with 1 cache set

8. REFERENCES

[1] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In S. L. Min, R. Pettit, P. Puschner, and T. Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 431–446, Washington, D.C., 2013. USENIX.

[3] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2):131–181, Nov 1999.

[4] D. Griffin, B. Lesage, A. Burns, and R. I. Davis. Lossy compression for worst-case execution time analysis of plru caches. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 203:203–203:212, New York, NY, USA, 2014. ACM.

[5] D. Grund and J. Reineke. Toward precise plru cache analysis. In B. Lisper, editor, *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 28–39, July 2010.

[6] D. Monniaux and V. Touzeau. On the complexity of cache analysis for different replacement policies.

[7] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, Nov 2007.

[8] V. Touzeau, C. Maïza, D. Monniaux, and J. Reineke. Fast and exact analysis for lru caches. *Proc. ACM Program. Lang.*, 3(POPL):54:1–54:29, Jan. 2019.

TSN Support for Quality of Service in Space

Pierre-Julien CHAINE
Airbus Defence & Space
Toulouse, FRANCE
pierre-julien.chaine@airbus.com

Marc BOYER
ONERA
Toulouse, FRANCE
marc.boyer@onera.fr

Claire PAGETTI
ONERA
Toulouse, FRANCE
claire.pagetti@onera.fr

Franck WARTEL
Airbus Defence & Space
Toulouse, FRANCE
franck.wartel@airbus.com

***Abstract*—The European spacecraft industry has developed guidelines for generic satellite development known as SAVOIR (Space AVionics Open Interface Architecture). While the current satellites on-board networks implementations are compliant with this standard, their evolution opportunities are strongly limited. New missions and new clients are always more demanding on performance on-board, leading to the conclusion that the satellite embedded network must be upgraded. One opportunity appears with Time Sensitive Networking, an IEEE Ethernet technology capable of supporting both real-time and high-bandwidth traffic. The goal of this paper is to discuss, in a qualitative study, how TSN protocols can help to integrate Quality of Service in new generation satellites.**

***Index Terms*—Time Sensitive Network (TSN), Embedded Networks, Satellites, SAVOIR-OSRA, Ethernet**

I. Satellite Architecture Overview

A. Introduction

For a long time, agencies and space companies, at prime and supplier levels, have raised the need of increasing the level of reuse and standardization in spacecraft avionics systems in order to improve efficiency and reduce development costs. This has led to studies and initiatives which are now federated under the Space Avionics Open Interface Architecture (SAVOIR) [1] initiative through different working groups.

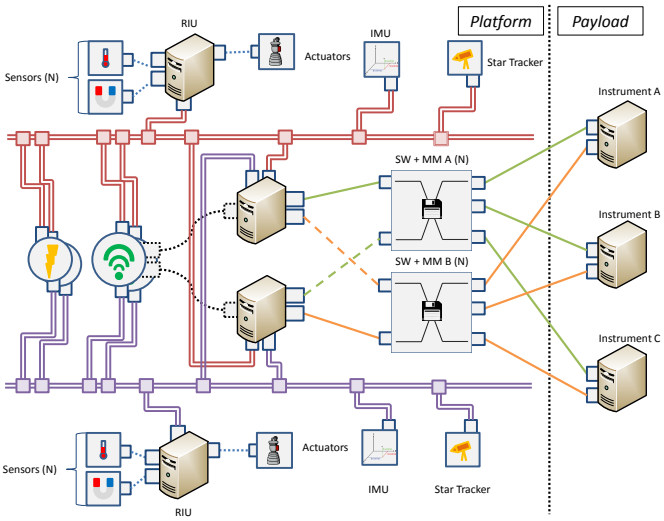


Fig. 1. Traditional Satellite Network Topology

B. Current Satellite Network

Among all the elements of the SAVOIR Reference Architecture, let us focus on the communication links. In a generic satellite architecture, the on-board network is commonly "composed" of two interconnected networks: *platform* and *payload*. Each of these networks fulfils different and sometimes opposite needs.

On the one hand, the *platform network* is in charge of conveying all the necessary information in order to guarantee the nominal behaviour of the satellite. It transmits data from sensors (position, magnetic field, temperature) as well as, among others, flight control commands. This kind of traffic, often described as *time critical traffic* requires bounded latency and low jitter communications. However, due to the small size and small volume of messages, a low data rate is enough to achieve the platform needs. In general, the platform network is implemented using a dual MIL-STD-1553 bus [3] or CAN [4] bus.

On the other hand, the *payload network* requires a very high data rate in order to convey the huge amount of raw data generated by the payload instruments such as pictures from a telescope (GAIA, SPOT), telemeters from a weather sensor or IoT (*Internet of Things)* data. However, the constraints are less stringent for a payload network: a delay in the packet communication path will not impact the nominal behaviour of the satellite. The payload network is based in general on SpaceWire [5].

C. Current Satellite Architecture

The generic satellite network topology shown in Fig. 1 is compliant with SAVOIR reference architecture: it contains several equipments corresponding to different functions interconnected with communication links. First of all, this architecture has one (duplicated) *On-Board Computer* or *OBC*. This OBC is the master that manages all the platform equipments. Using a 1553 bus, it handles a polling mechanism to all the sensors or actuators, being eventually gathered in a *Remote Interface Unit - RIU* - also called *Data Concentrator*. OBC N, for *Nominal*, (and OBC R, for *Redundant*) is also hosting *AOCS - Attitude and Orbit Control Subsystem* functions. To do so, it gathers information from several sensors, including one or several Star Trackers, processes and exploits them in order to control the propulsion system of the satellite. OBC is also connected to a data storage system, usually a solid state mass memory, mainly used for storing payload data, which is connected to the

instruments through a SpaceWire network (payload network). Finally, it is generally in charge of routing Telecommand and Telemetry between the communication subsystem and the instruments.

The network itself, platform and payload, is mirrored. There are always 2 distinct links from one device to one other. On the platform side, as well as on the payload side, there is only 1 active bus (1553 or SpaceWire) at any given time, resulting in a cold redundancy scheme. On the payload side, there is also only one active SpaceWire switch. This active switch is constantly monitored in order to be able to trigger the other switch in case of failure.

II. Opportunity for an unified TSN network

Although the actual architecture works perfectly fine, it has started to show its limits: new instruments and more generally new equipments are capable of generating gigabits of data that the network cannot handle in its current version (100Mbits/s on a SpaceWire network). Using a gigabit-capable network could allow satellite users to access this huge amount of raw data. Furthermore, Spacewire bus is only used in the spacecraft industry, thus its development and update is quite expensive. Using a technology based on Ethernet and COTS - *Commercial-off-the-shelves* - components could help lower the overall price of the satellite network. Finally, adding *Quality of Service* mechanisms provided by TSN could ease the integration of an increasing number of equipments on-board.

A. Challenge

The Unified TSN-based network is a path to investigate in order define the upgrade of the satellite on-board network. Thus, the question is to find whether TSN is a superset of 1553 + Spacewire, i.e. is it possible to satisfy both *Platform* and *Payload* network requirements using this technology?

In order to deal with this challenge, several aspects have to be analysed:

1) What are the valid topologies for the future on-board network?
2) What are the protocols/features offered by TSN ?
3) What are the requirements of the satellite network ?
4) What is the minimum subset of TSN protocols satisfying these requirements ?

We provide some answers in the next sections.

B. Which topology for a Unified TSN-based Network ?

We have to keep in mind that any new topology must be compliant with SAVOIR requirements. The question of how to organize the switches and links is still open at this stage and will be explored in the next year. For instance there could be 2 independent networks : one for *Platform* and one for *Payload*.

C. Overview of TSN protocols

For the last years, an IEEE Ethernet technology capable of supporting both real-time and high-bandwidth traffic has been defined in the working group TSN (Time Sensitive Networking) [6], continuing the work of the former AVB (Audio Video Broadcasting) working group. Founded in 2012, the working group has already published a dozen of amendments to the 802.1 standard family in order to ensure a behaviour that is simultaneously real-time, adaptive and flexible, mixing synchronous and asynchronous approaches. Fig. 2 summarizes the available TSN features/protocols.

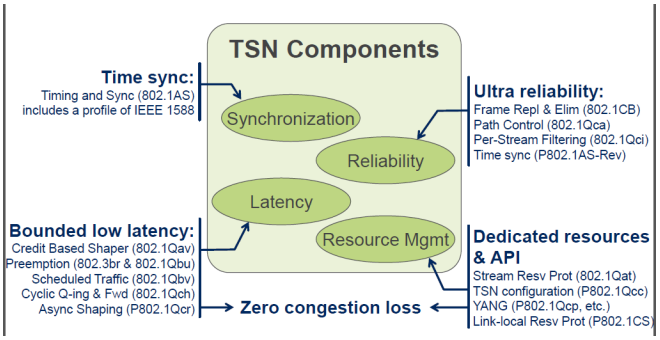


Fig. 2. TSN available services [7]

From all available TSN addenda, five main directions are identified:

- *Time Synchronization* (802.1AS, 802.1AS-rev),
- *Ultra Reliability* (802.1CB, 802.1Qca, 802.1Qci, 802.1As-Rev),
- *Bounded Low Latency* (802.1Qav, 802.1Qbu, 802.1Qbv, 802.1Qch, 802.1Qcr),
- *Dedicated Resources and API* (802.1Qat, 802.1Qcc, 802.1Qcp),
- *Zero Congestion Loss* (802.1Qav, 802.1Qbu, 802.1Qbv, 802.1Qch, 802.1Qcr, 802.1Qat, 802.1Qcc, 802.1Qcp).

III. Which protocols for a Unified TSN Network ?

A. Proposed Methodology

From the brief introduction, it is clear that TSN is wide and its complexity could increase quickly. As a consequence, we believe that TSN protocols are very likely not to be used all at the same time and that a selection of TSN features must be realized to define which protocols should or should not be targeted for a use in space. In order to do so, the important task is to define a goal, with precise KPIs or *Key Performance Indicators*, that the On-Board Network should reach using TSN protocols. This will lead to the identification of one or several combinations of TSN features that satisfy this goal. Some higher level criterion (cost, complexity, code size, targeted mission, etc.) will help refine the choice from one combination to one other.

B. On-Board Network: current and future requirements

The general goal is to fulfil the satellite network requirements. This seems easy, however today, these requirements need to be redefined. In fact, in the actual on-board network specification, many requirements suffer from *over-specification* i.e. being more inherited from the actual network

Requirements	Platfom - 1553	Payload - SpaceWire	TSN
Number of end-systems	5-8	1-15	✔
Data Rate	Low (1Mbit/s)	High (200Mbits/s)	✔
Perf. Requirements	Low Latency(<10ms)	High average throughput	?
	No Jitter (<10μs)		
Transaction size	2-64bytes	Unlimited (typically 2048bytes)	?
Simple behaviour	✔	✔	?

Fig. 3. Platform and Payload Network Performance Requirements

technology than driven by the satellite architecture itself. Hence, the first job will consist in eliminating this *over-specification*.

Once this has been done, the goal i.e. the network requirements will be classified into two categories: *performance & real-time properties* requirements on the one hand and *safety* requirements on the other hand. The rest of the paper will deal with the first category of requirements.

C. Performance & Real-time properties requirements

The table in Fig 3 gives an overview of the very first *performance & real-time properties* requirements and their associated KPIs, that we have identified for the satellite network. It is still a work in progress: the current KPIs may be clarified and new requirements may be added to the list.

The figure also gives a glimpse of what TSN could be capable of with respect to the previously introduced requirements. The evaluation, for some lines, was very simple; for instance, the supported number of end-station in TSN is, in theory, equal to the number of available Ethernet MAC addresses and it is way higher that what is required in space (around 25 end-stations). For other lines however, the analysis is still pending.

The rest of the paper will focus on the third line of the table i.e. the following requirement:

Requirement 1: The unified on-board network shall be capable of handling, in the same time, low latency, low jitter traffic and high throughput traffic.

IV. Mixing different traffic with QoS using TSN-802.1Qbv

This section will enlighten how TSN protocol IEEE 802.1Qbv - *Enhancement for Scheduled Traffic* could stasify Req. 1. In order to understand how to validate the previous statement, let us apply 802.1Qbv to a simple network example, show in the figure below.

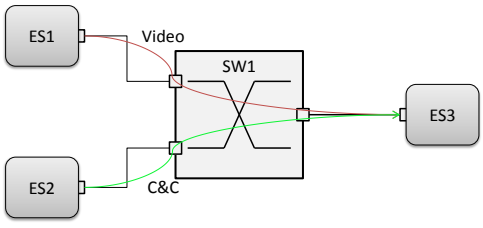


Fig. 4. Motivating network example

A. Motivating Example

This network has 3 End-Stations (ES), 1 Switch (SW), all of them are 1Gbits/s capable. The physical medium is also capable of conveying a 1 Gbits/s traffic load. The nature of the physical medium (optical or copper) is outside of the scope of this study. ES1 sends video traffic to ES3 and ES2 sends C&C traffic to ES3. The characteristics and requirements of both flows are listed in the following table.

Type	C&C	Video
Size	512bits	10Mbits
Freq	100Hz	8Hz
Priority	High	Low
Latency req.	1ms	2μs
Jitter req.	100ms	100ms

Fig. 5. Traffic characteristics and requirements

TSN-802.1Qbv protocol will be used in the output port of SW1. This protocol has several parameters that can be configured. Finder whether using 802.1Qbv satisfies Req. 1 (in our example) relays now in finding if there exist any combination of said parameters that satisfies the network requirements listed in Fig. 5. In this paper, we have chosen, among all 802.1Qv parameters, only three to study :

- *Transmission Selection Algorithm (TSA)*
- *Transmission Gates*
- *Preemption*

The parameters and their possible values are summarized in Fig 7, a schematic representation of SW1 output port.

Already, with only 3 parameters and their possible values, twelve 802.1Qbv combinations/configurations exist and should be analysed to see if Video and C&C requirements are met. Again, for this short paper, we have limited the study to only 2 configurations, detailed in Fig 6.

Configuration	1 (see. IV-B)	2 (see .IV-C)
TSA	*none*	*none*
Transmission Gates	*always open*	*always open*
Preemption	*non-preemptive*	*preemptive*

Fig. 6. 802.1Qbv summary

B. First configuration: Static Priority

In this first configuration, the TSN port has no transmission selection algorithm configured, no gate control list configured,

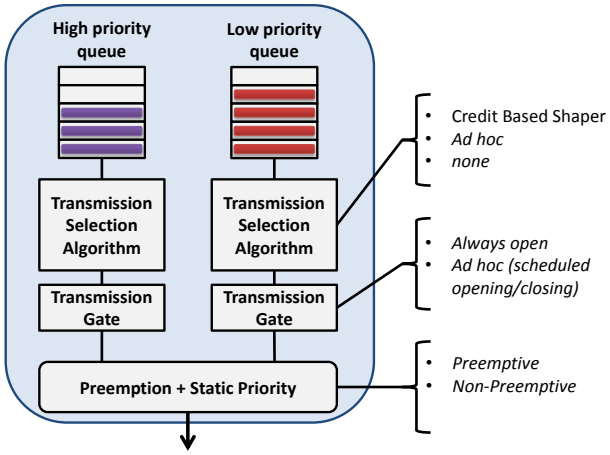


Fig. 7. SW1 Output port schematic view

and no frame preemption. In that case, this port is equivalent to a port with two queues arbitrated using static priority. In fact, it behaves like a standard Ethernet switch port would i.e. without any TSN or AVB protocols.

As specified in Fig. 3, for this example, C&C traffic has a higher priority than Video traffic. In terms of delay:

- The best-case scenario for this configuration is the following: a C&C frames arrives in the switch, no other packet is being processed or stored by the switch and the frame is *"immediately"* transmitted to ES3. In this best-case, the delay induced by SW1 is its technological latency. Let us assume that this latency is not significant when compared to the frame's delay requirements.

- The worst-case scenario is as follows: if a C&C frame arrives just after the beginning of the transmission of a Video frame of 1518 bytes (max. length for an Ethernet frame), the C&C frame will have to wait for that frame to be transmitted before being granted access to the medium. The waiting time is obtained by computing the formula below:

$$\frac{MessageSize}{LinkSpeed} = \frac{1518 * 8}{1.10^9} = 12,144\mu s \qquad (1)$$

In this configuration, the jitter, understood as delay variability, for a C&C frame, has a value of 12,114μs, which is higher than the maximum admissible jitter for this class of traffic. Obviously, this is only one part of the delay analysis that should be realised *End-to-End* on this network example for this configuration. Nevertheless, if the jitter is already higher than the requirements in the switch only, the rest of the analysis does not matter: this configuration is declared not valid and Req. 1 is not satisfied.

C. Second configuration: Strict Priority + Frame Preemption

In the first configuration, the length of the frame, in the worst case, was too big; so was the C&C frames' jitter. Fortunately, TSN introduced 802.1Qbu (with the help of 802.3br): a protocol dedicated to *Frame Preemption*. Thanks to this feature, an Ethernet frame of lower priority can now be preempted by a higher priority frame. With this new protocol, the length of the shortest non-preemptable frame drops down to 143 bytes. By re-applying (1) with this new frame size, the delay is reduced to 1,144μs which is compatible with C&C traffic class jitter requirement. Considering that C&C traffic has its constraints satisfied, it is necessary to check that Video frames still have enough bandwidth. C&C traffic uses less than 1% of the available bandwidth and Video requires 1%. Hence there should enough bandwidth available for Video frames and Viveo constraints are fulfilled (formal analysis pending). As a result, this configuration is a valid configuration with respect to the initial goal and Req. 1 is satisfied.

V. Way Forward

The last section gave a hint of the level of complexity associated with the upgrade of the satellite on-board network using TSN, only focusing on *Quality of Service*. Standard Ethernet (using static priority) is not enough for a future satellite network. One simple asynchronous solution relies on TSN Frame Preemption and offer a minimal jitter of 1,144μs at low cost on a 1Gbits/s network. For applications demanding an even higher jitter, one possible solution relies on using 802.1Qbv synchronous feature i.e. configure a schedule for the transmission gates. The analysis is still pending but it could potentially help building a "*zero-jitter*" path for the C&C flow. Going to a synchronous TSN configuration however has a heavy impact as time distribution and synchronization services must be added to the current architecture. The trade-off of going to one solution or one other will depend on the results of *Safety* analysis as well as mission requirements and other parameters.

The study will continue to define all the satellite network requirements and analyse all targeted TSN protocols/features.

References

[1] European Space Agency, "SAVOIR Functional Reference Architecture," Technical Note 001, https://essr.esa.int/project/savoir, April 2016.
[2] ESA, "OSRA Communication Network Specification," Technical Note 003, https://essr.esa.int/project/savoir, April 2017.
[3] "Aircraft Internal Time Division Command/Response Multiplex Data Bus," MIL-STD-1553B, September 1978.
[4] "Road vehicles – Controller area network (CAN)," ISO 11898-2:2016, December 2016.
[5] ESA, "Spacewire-Links, nodes, routers and networks," ECSS-E-ST-50-12C, July 2018.
[6] IEEE, Time Sensitive Networking Task Group, "https://1.ieee802.org/tsn/".
[7] J. Farkas, "Introduction to 802.1, Focus on the Time-Sensitive Network Task Group," March 2018.
[8] J. Migge and J. Villanueva and N. Navet and M. Boyer, "Insights on the Performance and Configuration of AVB and TSN in Automotive Ethernet Networks," ERTS, January 2018.
[9] C. Pruvost and T. Planche and O. Notebaert and A. Rossignol and H. Herpel and A. Schüttauf, "Ethernet for Space: an enabler for next generation avionics," DASIA, May 2016.
[10] O. Notebaert and J. Lachaize and R. Clavier and A. Fueser and H. Herpel and G. Montano and L. Planche, "SpaceWire 2: Needs and Evaluation Metrics," 6th SpaceWire Conference, November 2014.

Generating substation network simulations from substation configuration description files

Théo Docquier
University of Lorraine - CNRS - LORIA - SCLE SFE
Nancy, France
theo.docquier@loria.fr

Ye-Qiong Song
University of Lorraine - CNRS - LORIA
Nancy, France
ye-qiong.song@loria.fr

Vincent Chevrier
University of Lorraine - CNRS - LORIA
Nancy, France
vincent.chevrier@loria.fr

Ludovic Pontnau
SCLE SFE
Toulouse, France
ludovic.pontnau@scle.fr

Abdelaziz Ahmed-Nacer
SCLE SFE
Toulouse, France
abdelaziz.ahmed-nacer@scle.fr

ABSTRACT

The IEC 61850 standard has become the reference standard for substation configuration in smart electric grids, introducing data and service models to achieve interoperability between the network nodes. As the standard is currently based on the switched Ethernet architecture, there is a lot of work on its performance evaluation for guaranteeing real-time constraints. However, there still lacks a link between the substation configuration and its underlying Ethernet performance models (analytic or simulation). For bridging this gap, we propose in this paper a tool, called Simulation Tool for Analysis of substation netwoRkS (STARS) allowing the performance evaluation of any substation configuration through automatic generation of the corresponding simulation model from the Substation Configuration Description (SCD) file. STARS is based on the OMNeT++ simulator allowing the mapping of a real IEC 61850 system configuration to simulation parameters. It also provides a simple network configuration interface. This paper gives an overview of the STARS features through a simple example and points out its future evolution towards co-simulation of substation control algorithms, network performance, and electric grid behaviors.

1. INTRODUCTION

The need for power utilities in terms of distribution, transport, security and quality of service is growing with the years.

In this context, **substations** constitute the critical part within a smart grid. Their main goals are to ensure the correct power distribution through the grid, and to react to any abnormal situations. To meet these requirements, substations are composed of multiple **Intelligent Electronic Devices** (IED), each of them dealing with specific tasks such as the monitoring of voltage/current at different points of the substation or the control of the state of primary equipment (high voltage switches, circuit breakers, transformers...).

Because of the growing number of IEDs within substations, architectures tend to move from traditional hardwired solutions to digital frame based architectures. In such a context, the **IEC 61850** standard is specifically defined to meet these challenges. Major contributions of the standard on the interoperability can be summarized in three parts: **1)** defining a complete model for electrical components (e.g., power/current/voltage transformer, circuit breaker) using an object-oriented paradigm, **2)** allowing the configuration of all equipment belonging to a substation with a language and representation format, and **3)** introducing the Ethernet standard for the communication between the equipment and the specification of performance requirement (Table 1).

Transfert time class	Transfer time [ms]	Application examples: Transfer of
TT0	>1000	Files, events, log, contents
TT1	1000	Events, alarms
TT2	500	Operator commands
TT3	100	Slow automatic interactions
TT4	20	Fast automatic interactions
TT5	10	Releases, status changes
TT6	3	Trips, blockings

Table 1: Performance requirements for different IEC 61850 applications.

The configuration part introduces the XML based Substation Configuration Language (SCL), described as both a language and a representation format. IEDs and the Substation architecture are fully described respectively by the IED Configuration Description (ICD) and Substation Configuration Description (SCD) files. These two files use the Data Object Model (DOM) defined by the IEC 61850 standard and encode them with an SCL format.

For the communication part, the standard uses switched Ethernet networks as the main basis for the communication architecture. To ensure data communication, three main protocols are introduced: *Generic Object Oriented Substation Event* (GOOSE) a layer 2 protocol used for fast messages transmission, *Sampled Measured Value* (SMV) a layer 2 protocol for periodic sample transport on the network and the *Manufacturing Message Specification* (MMS) a protocol above the TCP/IP layer used for reporting and slow messages transmission.

Contributions of the standard are illustrated Figure 1. The configuration part details the node and communication descriptions of each IED while the network part brings the use of Ethernet standard plus the previously described protocols (MMS, GOOSE & SMV).

However, neither the Ethernet standard nor the specified protocols define mechanisms to guarantee the architecture's **determinism** or the **real-time requirements**. Furthermore, the SCD file does not contain any indication about the network architecture's structure i.e., how to interconnect IEDs and devices with each other. This leads to the issue of the network architecture design. Even if the configuration and the communication are already described in the SCD file, the possibilities for the architecture design are multiple. Examples of issues are about the number of switches to use and how to interconnect them, the VLAN configuration or the bandwidth allocation. If the architecture design is not correctly carried out and e.g., the delay of an important message (e.g., a trip order) is too high, it could lead to serious situations, both economically and humanely.

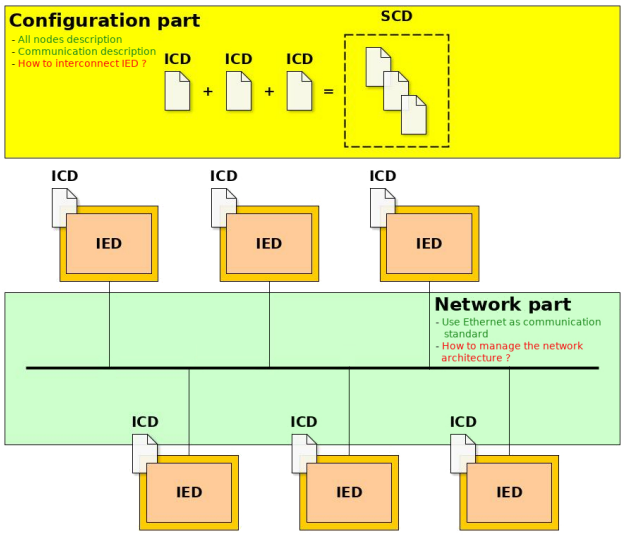


Figure 1: IEC 61850 contribution for configuration and communication part.

Since the communication architecture plays a critical role in substation operations, it must be as reliable as possible. Therefore, it appears a need to have tools that can **1)** take into account an SCD file containing IEDs and their properties and **2)** build a network architecture with all IEDs and evaluate the performance to guarantee real-time constraints.

At the best of our knowledge, no work has ever considered the configuration part along with the communication part in their works, i.e., taking a real configuration with an SCD file, how to model the network architecture accordingly and how to assess it.

In this paper, we present our approach considering the modeling of the network architecture as well as the mapping of an actual IEC 61850 SCD file to simulation parameters. We finally give an example of simulation and result analysis. The rest of the paper is organized as follows. Section II discusses about the recent works on the substation communication modeling. We describe STARS in Section III and give an overview of its utilization in Section IV. Section V concludes the paper.

2. RECENT WORKS

In the literature, modeling substation communication is based on two main approaches: analytic and simulation.

Analytic modeling: Mathematical models are used to represent the communication architecture. These models are used to determine the worst-case parameters (e.g., delay, backlog) or a statistical approximation for a given scenario. Previous works have been done on the modeling of the MMS protocol [1] or on the modeling of GOOSE and SMV at the same time with priority policy [2] both using Network Calculus approach. However, no existing work has taken into account the whole IEC 61850 protocols or existing configuration. Moreover, the pessimism for the worst case scenario always increases drastically as the complexity of the substation grows, making Network Calculus difficult to use for complex cases.

Simulation: It consists to choose one network simulator and then develop the model. Recent works have been performed to model and simulate IEC 61850 architecture using the OMNeT++ simulator. In [3] the three IEC 61850 protocols are implemented while the work made in [4] only focuses on GOOSE and SMV. However, neither of them introduce the configuration part of IEC 61850, as their works focus on the development of specific IEC 61850 network communication models.

For the rest of the paper, we focus on the simulation part.

3. DESCRIPTION OF STARS

3.1 Global description

To allow a simple utilisation of a performance evaluation tool, it appears necessary to have two elements:

User interface (UI): act as the "input" of the tool. It allows to specify the network architecture, the data model configuration (from the SCD) and simulation parameters;

Operation core (OC): act as the "engine" of the tool. It takes the input parameters coming from the UI and runs processes to obtain expected results.

The tool can be viewed as a black box system. The only tasks for the user are the configuration of the network architecture plus simulation parameters and the loading of the SCD file corresponding to the substation system configuration the user wants to evaluate.

Figure 2 gives the conceptual view of STARS.

3.2 User Interface

STARS UI composition is described by the left part Figure 2. It consists of a sequence of operations the user has to do *via* the interface before starting the simulation. These operations are as follows:

SCD loading: loads the SCD file related to a specific substation;

Node description: instantiates each IED of the architecture. Instantiation must be consistent regarding the SCD content as all the IEDs are already described with their configurations;

Switches description: instantiates each switch of the architecture;

Architecture: describes relations between each node and switch composing the architecture;

Scenario description and simulation setting: configures the simulation and scenario parameters. For scenario, the user can configure the number of fault (e.g., thunderbolt on the substation, overload) during the simulation or the period between two faults. For simulation, the user can

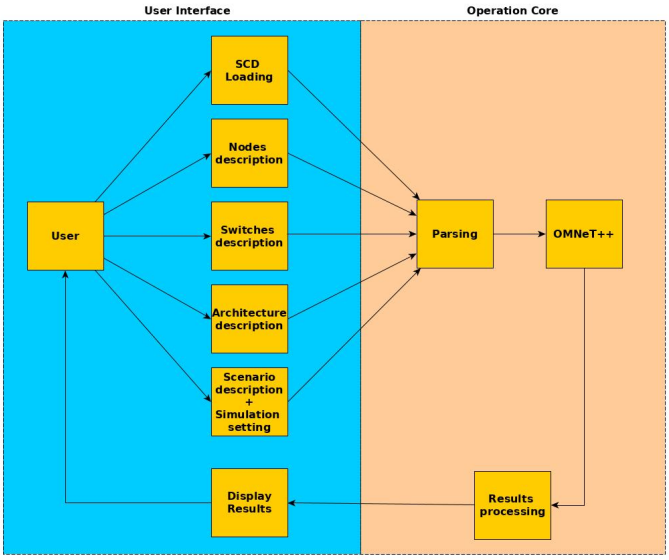


Figure 2: Conceptual view of STARS.

configure the simulation time and delay thresholds for which the user will be notified if the actual delay exceeds them.

The UI is also responsible for the display of results (e.g., delays, packet loss, buffer utilization) with graphs once the simulation is completed.

3.3 Operation core

The STARS OC composition is described by the right part of Figure 2. It is made of three fully automated operations:
Parsing: information collected during the configuration part is parsed into files that can be understood by the simulator;
Simulating : the result of the parsing operation allows to run the simulation;
Computing result: the obtained results are analyzed and processed, then passed to the UI to display the information.

We chose the OMNeT++ simulator for the simulating operation. We used the model developed by [4] which is available on the internet for the substation modeling. We added some improvement such as a light version of the MMS protocol, a better mechanism for GOOSE repetition pattern[1] and the possibility to have multiple Application Service Data Units (ASDU[2]) carried by one SMV frame.

To run a simulation, OMNeT++ needs three different files (according to the user guide manual[3]):
Simulation program: containing the compiled code for modules, simulation kernels and messages. This part contains the core of the model description;
NED (Network Description): containing modules (e.g., IEDs, switches, servers ...) and their interactions (e.g., link capacities, propagation delay, frame loss rate...);
INI: containing the simulation parameters (e.g., the simulation time) and the parameter descriptions of all modules in the network (e.g., frame sizes, emission periods...).

[1]Algorithm used by the GOOSE protocol to ensure the data exchange reliability. It consists of several repetitions of the same message several times.
[2]An ASDU can be considered as a group of samples.
[3]Available at https://doc.omnetpp.org/omnetpp/manual/

In order to provide the INI and NED files to OMNeT++, we develop two parsers whose role is to extract the required information from the SCD file and generate the NED and INI files. As the model does not change, the simulation program is already compiled and ready to use. As soon as the NED and INI files are generated, the simulation can be run. Once the simulation is completed, the result needs to be displayed. To do so, we developed a third parser taking the result file produced by OMNeT++ and extracting the main values. Once extracted, the results are displayed to the user as illustrated Figure 4. A report containing the result displayed is also automatically created.

4. STARS'S CAPABILITIES

To illustrate the capabilities of STARS, we consider an SCD file coming from a substation benchmark and describing a real case study. The substation is composed of 17 IEDs. We start by loading the corresponding SCD file into Calc Software. The next step is to define a communication architecture. We consider a network topology with two interconnected switches called "Switch_1" and "Switch_2". All links on Switch_1 are set to 1 Gbps while all links on Switch_2 are set to 100 Mbps. The link between the two switches is set to 1 Gbps. At this stage, we can describe the obtained architecture with Figure 3.

All node behaviors (i.e., protocols used for communication, data to send and the sending periods) are fully described in the INI file where the information is collected from the SCD. In this simple scenario, the parser does not parse SMV from the SCD yet, and future additional work will implement it. GOOSE traffic is emitted by each node (except SUP nodes) to all other nodes. MMS traffic is emitted by each node to one of the declared MMS receivers. In our SCD file, only SUP nodes, ACC_1 and ACC_2 are considered as MMS receivers. GOOSE messages have a higher priority than MMS ones. Before running, we need to configure the scenario and simulation parameters. We choose the following scenario: 10 successive faults appear on the grid with a 1 ms interval, triggering GOOSE and MMS events. For the configuration of simulation parameters, we set the simulation time to 0.2s.

Once the previous step is completed, we can start the run process from Calc Software. Remaining processes (describe subsection 3.3) are handled by the OC part of STARS.

After the simulation is completed, the results can be retrieved as depicted in Fig. 4. For the given SCD and the simulated scenario, we can observe that the maximum delay for GOOSE traffic exceeds the 3 milliseconds limit for trip messages given in Table 1 (all APR nodes). Since APR nodes are set to 100 Mbps, we can speculate that the problem comes from an insufficient bandwidth. If this speculation proves to be true, then we have identified the problem that can be solved to meet the real-time requirement by replacing, for instance, the 100 Mbps link by a 1 Gbps one.

5. CONCLUSION AND FUTURE WORKS

The main issues addressed in this paper is to design and assess a substation communication architecture while taking into account a real substation configuration. To meet these requirements, STARS brings the following features:
Real configuration consideration: the obtained results are based on real SCD files;

Figure 3: Conversion from SCD file to OMNeT++ model.

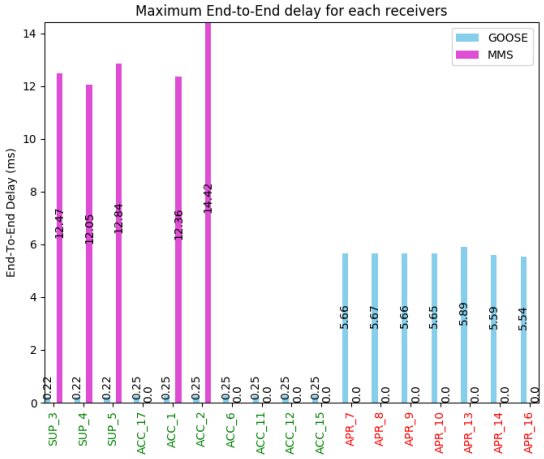


Figure 4: Example of a GUI result at the end of the simulation process

Network modeling and investigation: possibility to model a network architecture and assess it in case of non-compliance with real-time requirements. This allows users to detect a potential performance issue and points of failure, and deal with them;
Usable by non-expert: spreadsheet interface to design the network architecture, load the SCD, set the simulation parameters and run the simulation with results at the end;
Automatic Configuration: the SCD file is automatically parsed into files understandable by OMNeT++ reducing the time of the simulation configuration.

The main perspective is the improvement of our IEC 61850 model by integrating the new IEEE Time Sensitive Networking (TSN) standard, the latter describing mechanisms and protocols to guarantee determinism for Ethernet-based architecture. Some works have already been done [5], where a simulation model for TSN standard is proposed. TSN is also investigated by the IEC 61850-90-13 draft about deterministic networking for substations to improve overall performance of the communication network by using network resources efficiently. In addition to TSN study, an investigation on scheduling theory to formally describe how to set network parameters for network optimization is planned. Finally, we plan to investigate the co-simulation of our already made communication network model with control and electrical models with the help of the MECSYCO software [6] to improve results by integrating different fields of expertise to be as close as possible to the reality.

6. REFERENCES

[1] Nils Dorsch, Hanno Georg, and Christian Wietfeld. Analysing the real-time-capability of wide area communication in smart grids. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, pages 682–687. IEEE, 2014.

[2] Can Huang, Fangxing Li, Tao Ding, Yuming Jiang, Jiahui Guo, and Yilu Liu. A bounded model of the communication delay for system integrity protection schemes. *IEEE Transactions on Power Delivery*, 31(4):1921–1933, 2016.

[3] Javier Juárez, Carlos Rodríguez-Morcillo, and José Antonio Rodríguez-Mondéjar. Simulation of iec 61850-based substations under omnet++. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, pages 319–326, 2012.

[4] Héctor León, Carlos Montez, Marcelo Stemmer, and Francisco Vasques. Simulation models for iec 61850 communication in electrical substations using goose and smv time-critical messages. In *Factory Communication Systems (WFCS), 2016 IEEE World Conference on*, pages 1–8. IEEE, 2016.

[5] Jonathan Falk, David Hellmanns, Ben Carabelli, Naresh Nayak, Frank Dürr, Stephan Kehrer, and Kurt Rothermel. NeSTiNg: Simulating IEEE time-sensitive networking (TSN) in OMNeT++. In *Proceedings of the 2019 International Conference on Networked Systems (NetSys)*, Garching b. München, Germany, March 2019.

[6] Benjamin Camus, Thomas Paris, Julien Vaubourg, Yannick Presse, Christine Bourjot, Laurent Ciarletta, and Vincent Chevrier. Co-simulation of cyber-physical systems using a DEVS wrapping strategy in the MECSYCO middleware. *SIMULATION, vol. 94, 12: pp. 1099-1127.*, January 2018.

Formal Model of the Lipsi Processor: Definition and Use of its Timing Behavior

Imane Haur
CEA, List, France
Imane.Haur@cea.fr

Mihail Asavoae
CEA, List, France
Mihail.Asavoae@cea.fr

Mathieu Jan
CEA, List, France
Mathieu.Jan@cea.fr

ABSTRACT

Timing analysis of safety-critical systems derives timing bounds of applications (SW) executed on dedicated platforms (HW). The ensemble HW–SW features, from a timing perspective, two different types of computation – an SW-specific, instruction-driven timing progression and an HW-specific, cycle-driven one. In this paper, we focus on the SW side of this ensemble by defining formal models of architectures enriched with timing behaviors. Our (long-term) goal is to investigate how such models can help to handle memory interferences in multi-core architectures, by validating the timing behavior of our SW formal models against formal HW models extracted from hardware description languages (HDL). Our SW formal model is exemplified on a simple, accumulator-based processor called Lipsi, for which we report classical functional semantic as well as timing discrepancies issues found while applying our methodology.

Keywords

Formal semantics, timing model, Sail, memory interferences.

1. INTRODUCTION

Cyber-physical systems (CPSs) integrate computations running on embedded platforms into physical systems that they interact with. This integration, expressed through feedback loops between software and the physical environment, may need to satisfy strong timing guarantees. To verify them, timing analyses of computations (e.g., the worst-case execution time analysis), communications (e.g., the worst-case traversal time analysis) or both (e.g., the worst-case response time analysis) are combined. The commonality of all these analyses is that the application semantics, i.e. the software side (SW), is projected on the timing behavior of the underlying platforms, i.e. the hardware side (HW). A timing analysis indeed needs to be safe and accurate and the most precise timing is to be found at the execution platform-level, i.e. the HW-level.

The high-level functional and temporal properties are however obfuscated or even lost when translated to this binary low-level. High-level specifications, for instance, based on synchronous language [7], can be transformed into an intermediate language, usually the C language, in a correct by construction way. However, the initial high-level timing properties can no longer be directly expressed at the binary level, as most ISAs simply do not include timing. A few exceptions exist, such as PRET [13] and the Patmos [15] architectures that provide instructions that explicitly manipulate timings. This prevents to implement in a proper way the initial high-level temporal properties, which thus simply disappear at the binary level. It is the goal of WCET analyses to recover them at this level.

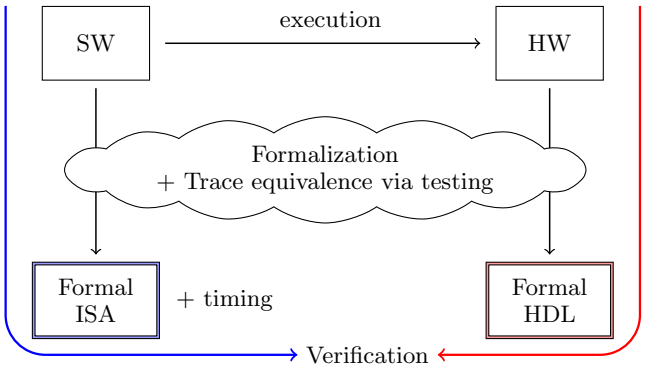


Figure 1: general workflow for the HW/SW co-validation of timing models for CPS.

We are currently building a methodology to build and validate timing models for CPS, by enriching formal SW models with timing behaviors extracted and abstracted from processor descriptions in register transfer-level (RTL) languages [3]. In this methodology, we manipulate the computations, viewed as a set of binaries of software (SW), i.e. sequences of instructions described at the instruction set architecture (ISA) binary level. We omit any networking components within CPSs and focus for now on single-core or multi-core architectures only. Now, a *timing model* in this context is a function between the time progression of SW, measured in executed instructions and the cycle-accurate timing, corresponding to HW. As shown by Figure 1, our general methodology to construct and validate such timing models is divided into 2 parts: one for the SW side and the other for the HW side. For the SW side, a timing model is added to a formal description of the ISA by relying on the input textual specification of the architecture. Both the functional and timing behaviors of this timing augmented formal SW model are tested, for confidence, against traces of actual code first over an instruction set simulator (ISS) and then over a real execution. Similarly, on the HW side, a formal model is built which thus includes by construction a timing behavior [4]. Finally, the last step consists of the co-validation of these HW and SW formal timing models, i.e. verifying the consistency of their two timing models.

While in [3] we illustrate this whole approach of both SW

and HW sides over the Lipsi processor [14], in this paper, we only focus on the formal SW model of Lipsi. Our *contribution is to show how a formal SW ISA-level model of a processor, enriched with timing behavior, can be used to detect both functional and timing discrepancies in its implementation.* We use Sail [1] to formally model programs as it is tailored for expressing ISA semantics of processors and has been successfully applied to formalize various ISAs, such as ARM, RISC-V, and MIPS. A Sail specification relies on the definition of an Abstract Syntax Type (AST) of the ISA of an architecture, i.e. a union of types with parameters. To each AST value, specific `execute` and `decode` functions are associated with respectively specify the sequential semantics of the instruction and the matching of its binary representation of its AST value. From a Sail specification, both emulators and theorem-prover definitions can be generated to support fast execution of programs or deductive reasoning.

2. FORMAL SW MODEL OF LIPSI

Overview of Lipsi. Lipsi is a tiny sequential 8-bit processor to be used in auxiliary functions or for teaching purposes. The ISA of Lipsi includes ALU operations using registers or immediate operands, loads/stores, unconditional and conditional branches, and an input/output (i/o) operation. A complete list of instructions and their encodings are shown in [14]. Instructions of Lipsi are encoded using a single byte, except branch operations and ALU operations with immediate operands. For these instructions, a second byte is used to store either the address of the target branch or the value of the immediate operand.

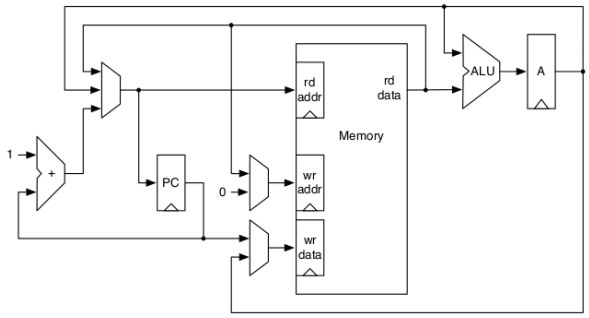


Figure 2: the datapath of Lipsi.

On the hardware side, Lipsi consists of an accumulator register (A), a program counter (PC), 16 additional registers and a single on-chip memory. Its datapath is shown in Figure 2. The memory is accessible through 2 ports: one for reads, the other one for writes. Memory Addresses are 9-bits values. The lower half of the 9-bit memory space stores up to 256 bytes of instructions, while the upper half stores first 16 additional registers ($R[x]$) followed by up to 240 bytes of data. $R[X]$ can be used to store intermediate results when performing ALU operations.

The hardware implementation of Lipsi is written in Chisel [5], and it has been synthesized to the DE2-115 FPGA board. It comes with a very simple timing model, as a single memory is connected to Lipsi. Two clock cycles are required to execute an ALU instruction: one for fetching the instruction and one for accessing the data and executing the ALU operation. Loading A with a value in $R[x]$ also takes 2 cycles while writing to $R[x]$ only takes 1 cycle due to the separated read/write ports to the memory. Updating $R[x]$ is performed while the next instruction is being fetched from the read port. Memory store and load operations use the additional registers $R[x]$ to store the targeted memory address. Those operations thus perform three memory accesses: one for fetching the instruction, another to retrieve the memory address from $R[x]$, and finally a last one to perform the memory operation at the specified memory address. A memory load thus takes 3 cycles, while a memory write takes only 2 cycles as the last access occurs meanwhile the next instruction is being fetched. The i/o operation takes only 1 cycle. Finally, an ISS written in Scala is also available.

```
type len_t = bits(8) /* 8-bit architecture */
register A : len_t /* Accumulator */
register PC : len_t /* Program Counter */
register nextPC : len_t /* For branch instructions */
register din : len_t /* i/o instruction, input port */
register dout : len_t /* output port */
```

We now present the formal SW model of Lipsi using the Sail language. We first define the architectural state of Lipsi, i.e. its accumulator `A`, its program counter `PC` and the ports used by the i/o instruction, i.e. `din` and `dout` for respectively the input and output ports. The `nextPC` register is used to store the address of the branch, i.e. the second byte of a branch instruction when it is decoded. All these variables are 8-bits registers, as in the hardware implementation of Lipsi.

Memory model. The structure `Memory` represents the memory of Lipsi. It embeds respectively the instruction and data spaces, which are defined as a vector of bytes. These vectors are organized in downward memory addresses. Finally, a vector of registers `Rs` represents the additional registers $R[x]$ of Lipsi.

```
type memory_data = vector(240, dec, bits(8))
type memory_inst = vector(256, dec, bits(8))
struct Memory = { Inst: memory_inst, Data: memory_data }
register Rs : vector(16, dec, bits(8)) /* R[x] */
```

We now show the formal specification of the write operations, for both the memory space but also for $R[x]$. The function `mem_write` updates the content of `Memory` with the value `v` at the memory address `adr`. Either the instruction or the data vector of `Memory` gets updated, depending on the value of the Most Significant Bit (MSB) of `adr`, a 9-bit value. Note that the data vector is updated only if `adr` does not target `Rs`.

```
val mem_write : (bits(9), bits(8), Memory) -> vector
    ↪ (256, dec, bits(8))
function mem_write (adr, v, mem) = {
  if (adr[8] == 0b1) then {
    if (adrbits_to_adrno(adr[7..0]) >= 16) then
      plain_vector_update (mem.Data, length(mem.Data) - 1
          ↪ - adrbits_to_adrno(adr[7..0]), v);
    else return mem.Data;
  } else {
    plain_vector_update(mem.Inst, length(mem.Inst) - 1 -
        ↪ adrbits_to_adrno(adr[7..0]),v);
  }}
```

The function `reg_write` updates $R[x]$ and shares with the function `mem_write` a similar signature. A string representing the name of register, noted `r`, is however used instead of a memory address. `r` is mapped into an offset in the data vector using the functions `reg_name` and `regbits_to_regno`. `X`

is the setter function to update `Rs` with the value `v`, which uses the overload feature of Sail to abstract read (not shown) and write accesses (function `wX`, signature not shown).

```
function wX (r, v) = if r < 16 then { Rs[15 - r] = v; }
overload X = {rX, wX}
val reg_write: (string, bits(8))->unit /* eq. to void */
function reg_write (r, v) = {
  X(regbits_to_regno(reg_name(r)), v); }
```

Finally, the overload feature of Sail is used to abstract the organization of the memory. Writing to $R[x]$ or the memory to implement the semantic of instruction is performed by simply calling the function `lipsi_write`. Similar functions are used for read operations (shown in the next paragraph).

```
overload lipsi_write = {mem_write, reg_write}
```

Instruction and timing models. We now present the part describing the semantic of instructions. We have modeled in Sail all the ISA of Lipsi. We only show the use of Sail to decode and execute ALU instructions that rely on registers. First, Sail supports the definition of scattered functions and unions allowing to group the mapping `decode`, `execute` functions and the AST union `ast` of an instruction in one place. The AST type `ALU_TYPE_REG` represents the considered ALU instructions. The mapping `encdec_alu_func_reg` matches a binary value to a constant value representing the requested ALU operation. The mapping `decode` matches the machine code of instructions to the associated AST node within `ast`. The concatenation operator `@` is used to extract, from the input bit vector, the requested ALU operation (`func`) and the index of the additional register (`reg`). Finally, the function `execute` implements the semantics of the instructions by first reading the value from the specified additional register, i.e. `reg_val` and then performing the specified ALU operation on `reg_val` and A. `accureg` is an accessor to A, for reading or writing.

```
scattered union ast
scattered mapping decode
scattered function execute

union clause ast = ALU_TYPE_Reg: (alu_func_reg, regbits)

mapping encdec_alu_func_reg: alu_func_reg<->bits(3) = {
  LIPSI_ADD <-> 0b000,
  LIPSI_SUB <-> 0b001,
  ...
  LIPSI_XOR <-> 0b110,
  LIPSI_LD <-> 0b111
}
mapping clause decode = ALU_TYPE_Reg(func, reg) <->
       0b0 @ encdec_alu_func_reg(func) @ reg

function clause execute ALU_TYPE_Reg(func, reg) = {
  let reg_val: len_t= lipsi_read(regbits_to_regno(reg));
  let ret : len_t = match func {
    LIPSI_ADD => reg_val + accureg(),
    LIPSI_SUB => accureg() - reg_val,
    ...
    LIPSI_XOR => reg_val ^ accureg(),
    LIPSI_LD => reg_val
  };
  accureg(ret);
}
```

For the timing model, we simply use a register to represent clock cycles. This register is incremented by the clock cycles associated with each instruction being decoded. The formal SW model executes instructions in single steps, which is also equivalent to an instruction-level simulation of the input program. However, this clock register tracks in a cycle-accurate manner the timing behavior of each instruction.

3. PRELIMINARY EVALUATION RESULTS

Semantic discrepancies. We identified several semantics discrepancies when we performed the trace equivalence between traces from the formal SW model, the simulators and the circuit. These discrepancies concern not only the functional semantic but also on the timing semantic, justifying the need for formalization and verification of timing models. First, the instructions `sh` and `brl`, specified in the ISA of Lipsi, are not implemented in the Chisel hardware design of Lipsi. Next, the instructions `adc` and `sbb` give output that are equivalent to respectively the `add` and the `sub` instructions, which is not the expected functional behavior.

A more interesting discrepancy concerns the i/o instruction. The specification of Lipsi allows to identify up to 16 different ports using bits [3:0] in the encoding of the i/o instruction. However, the hardware implementation of Lipsi only uses a single i/o port, the one with value index 0, to exchange values with the accumulator A: one for outputting the value of A and the other one to load a new value in A. Any other index value than 0 leads to a silent drop of the next instruction ($PC + 1$), i.e. the execution continues at $PC + 2$. Even if this unprocessed instruction leads to be interpreted as an ALU operation (default decoding), the value of the accumulator is not modified as it is guarded by a boolean value, not set by default. Note that contrary to the previous discrepancies, this difference was not explicitly documented in the hardware implementation of Lipsi.

Finally, while the instruction `exit` takes 3 cycles in the Lipsi circuit, it only takes 1 cycle in the Lipsi simulator. Our goal is to detect timing discrepancies and not to point out functionality issues in the considered design. However, such findings demonstrate that we can detect any kind of semantic discrepancy between the specified ISA and implementations of it. **Formal SW model complexity.** Chisel provides some high-level constructs to raise the level of hardware design abstraction, as it is embedded in the Scala programming language. Scala promotes functional-style programming and uses a strong static type system to facilitate concise and reusable code. Chisel then generates classical Verilog HDL. Table 1 shows the source lines of code (SLOC) of both the memory part and ISA part (i.e. datapath shown in Fig 2) of the hardware implementation of Lipsi in Chisel, the generated Verilog and our formal SW model in Sail. Both the ISA of Lipsi in Chisel and the formal SW model share approximately the same amount of SLOCs, respectively of 161 and 189 SLOCs. However, note that Lipsi is a tiny processor, the complexity of our formal SW model is thus expected to stay below the SLOCs required in Chisel to design more complex processors. As expected the SLOCs for Verilog is higher than in Chisel but is less than the SLOCs of our formal SW model in Sail for the memory part. This

Table 1: Source lines of code (SLOCs).

	Chisel	Verilog	SW model (Sail)
Memory	20	59	116
ISA	161	449	189

is explained by the fact that Sail provides helper function to manage memory only when a ELF format is used. We thus had to define our memory model from scratch, resulting in 116 SLOCs.

Detection of interferences. We now briefly report an illustration of the possible use of the built timing model: the detection of memory interferences. We thus consider three different input binaries to be run in our formal SW model of Lipsi, to represent a multi-core setting of Lipsi. Each Lipsi core is identified by an index. Besides, as each Lipsi has its private memory, we rely on the i/o instruction to emulate the access to a shared device. Note that we only emulate the access to a shared device, not the device itself nor its arbitration policy. If two i/o accesses occur at the same time, we thus assume that the one coming from the Lipsi with the smallest index wins the access. The other Lipsi cores continue their execution as if their accesses were valid, as we are mostly interested in timing properties of programs not in their functional correctness.

For the input binaries, we reuse the same memory model for the spacing of memory accesses as in [10]. In this model, programs are represented as a sequence of memory requests separated by a given number of processor clock cycles, representing the amount of computation that is performed between two memory accesses. We assume a composable computer architecture [9], which ensures that the distance between requests is independent of the execution of other programs. The only interference between the independent programs thus stems from accesses to the emulated shared device, i.e. the memory.

The sequences of memory requests of our three input programs are: $(\mathsf{A}\colon 2, 24, 12)$, $(\mathsf{B}\colon 14, 4, 2)$ and $(\mathsf{C}\colon 26, 6)$. Program A is made of 2 loops and thus generates i/o accesses at the (absolute) times 2, 26 and 38. Program B is made of a single loop and generates i/o accesses at the (absolute) times 14, 18 and 20. Finally, program C is also made of a single loop and generates i/o accesses at the absolute (times) 26 and 32. It is then trivial to detect that an interference is going to occur at time 26 between program A and C.

Our next step is to modify the input binaries, by adding appropriate `nop` instructions (ALU operations that do not change the current value of A) or by introducing a `delay` instruction, as in Patmos or PRET, to space out i/o accesses. While a straightforward algorithm can solve this problem for programs with a single path, the presence of multiple paths in input programs lead to an interesting optimization problem of minimizing the number of interferences.

4. RELATED WORK

Our approach increases the confidence in the formal semantics of ISA with the help of an ISS, providing a de facto procedure to actually verify an ISS. There are several works [6, 11], centered on the verification of ISS. The ISS presented in [6] is symbolic and addresses both functional and timing properties in processors using assertions in a similar way with our procedure as another symbolic ISS, which is constructed over an instruction-level abstraction [11]. We can also leverage existing formal SW models, such as [1] or [8] to extend their functional parts with a timing behavior. Besides, the K framework, which was recently used for a formal executable semantics of x86-64 ISA [8], is another option we have not yet considered so far [2] to design our formal SW model.

5. CONCLUSIONS

In this paper, we have presented how the Sail language is used to describe a formal SW model enriched with timing behaviors to reason about the timing properties of CPSs. We have reported functional but also temporal semantic discrepancies between the Lipsi specification and its hardware implementation written in Chisel. We have also discussed possible use of such formal SW models to handle memory interferences and their complexity compared to hardware description of them in Chisel and Verilog.

We are currently working on automatically generate formal HW models directly from the HDL code of processors to co-validate these formal models and their timing behaviors. As Lipsi is a simple processor, we have ongoing work on popular RISC-V designs, with complicated timing models due to pipelining, multi-level caches and speculation mechanism. Finally, another goal is to apply this approach to timing anomalies detection [4, 12] and minimization of memory interferences.

6. REFERENCES

[1] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell. ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. *PACMPL*, 3(POPL):71:1–71:31, 2019.

[2] M. Asavoae. K semantics for assembly languages: A case study. *Electr. Notes Theor. Comput. Sci.*, 304:111–125, 2014.

[3] M. Asavoae, I. Haur, M. Jan, B. B. Hedia, and M. Schoeberl. Towards formal co-validation of hardware and software timing models of cps. In *Proc of Model-Based Design of Cyber Physical Systems (CyPhy'19)*, October 2019. To appear.

[4] M. Asavoae, B. B. Hedia, and M. Jan. Formal executable models for automatic detection of timing anomalies. In *18th International Workshop on Worst-Case Execution Time Analysis, WCET 2018*, pages 2:1–2:13, 2018.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225. ACM, 2012.

[6] D. L. Beatty and R. E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proc. of the 31st Conf. on Design Automation*, pages 596–602, 1994.

[7] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[8] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Rosu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proc. of the 40th PLDI*, pages 1133–1148, 2019.

[9] S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015.

[10] F. Hebbache, M. Jan, F. Brandner, and L. Pautet. Shedding the shackles of time-division multiplexing. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 456–468, 2018.

[11] B. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik. Instruction-level abstraction (ILA): A uniform specification for system-on-chip (soc) verification. *ACM Trans. Design Autom. Electr. Syst.*, 24(1):10:1–10:24, 2019.

[12] M. Jan, M. Asavoae, M. Schoeberl, and E. A. Lee. Formal semantics of predictable pipelines: a comparative study. In *25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Beijing, China, January 2020. To appear.

[13] I. Liu et al. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *2012 IEEE 30th international conference on computer design (ICCD)*, pages 87–93. IEEE, 2012.

[14] M. Schoeberl. Lipsi: Probably the smallest processor in the world. In *Architecture of Computing Systems - ARCS 2018 - 31st International Conference*, pages 18–30, 2018.

[15] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, and D. Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Apr 2018.

Reasoning about non-functional properties using compiler intrinsic function annotations

Shashank Jadhav
Hamburg University of Technology
shashank.jadhav@tuhh.de

Mikko Roth
Hamburg University of Technology
mikko.roth@tuhh.de

Heiko Falk
Hamburg University of Technology
heiko.falk@tuhh.de

Christopher Brown
University of St Andrews
cmb21@st-andrews.ac.uk

Adam Barwell
University of St Andrews
adb23@st-andrews.ac.uk

ABSTRACT

Embedded systems often need to adhere to time and energy constraints. With the increasing popularity of embedded systems, the interest in evaluating and optimizing non-functional properties like execution time and energy of these systems is increasing.

In this paper, we describe a Resource-usage Estimate Expression Language (REEL), which allows the user to argue about these properties, within the source code, in a compiler understandable manner. Furthermore, we discuss the integration of REEL within a compiler framework. We, also show the propagation of REEL annotations within the compiler, and how they can be exploited to make decisions based on the non-functional properties within the source code. Finally, we explore REEL's potential to perform ILP-based optimizations.

Keywords

Compilation, Annotations, Non-functional Properties, Function Inlining

1. INTRODUCTION

Modern embedded systems, often, are subjected to constraints like time and energy consumption. Depending upon the physical and functional aspects of these systems, the severity of the correctness of these systems may vary. Therefore, the non-functional properties like time and energy consumption are needed to be taken into consideration while developing for these systems. For embedded systems that are categorized as real-time, it is paramount that they should meet their deadlines. As embedded systems are becoming more common, there is an effort to strive for reliable and energy-efficient systems. Generating an energy-efficient code that guarantees a response within the specified time constraint or deadline is a way to tackle this issue.

Common compilers are likely to perform optimizations, which can lead to reasonably optimized code, but without any guarantees on execution times. When it comes to embedded systems, this often is not sufficient. We have to take into consideration fundamental properties like, e.g., the Worst-Case Execution Time (WCET). If we can ensure the WCET of the code is within a specified deadline, we can guarantee the real-time nature of an embedded system. If such properties can be estimated by static analysis methods, that are difficult to perform before compiling for a particular platform with appropriate optimizations enabled, they should be integrated into the compilation process. The compiler can then make correct decisions depending on such properties, allowing more generic code designs.

In this paper, we introduce the Resource-usage Estimate Expression Language (REEL), which is a collection of compiler intrinsic functions. It enables transparency and easy access to information about non-functional properties provided by the compiler from within the users' source code. It allows the user, or another tool for that matter, to reason about non-functional properties using the compiler's capabilities.

```
1 /* The WCET of this for-loop is important. */
2 for ( i = 0; i < 100; ++i )
3   a +=2;
4 /* This application should further run in
       either energy-efficient or high-performance
        mode, depending on the for-loop's WCET. */
```

Consider the code snippet above, in which the WCET of the for-loop is crucial for the user to make decisions about the applications' execution mode. In such cases, having access to the non-functional properties of the code is essential. Therefore, it would be useful if the user can annotate the code to steer the compiler to generate a suitable binary and provide the essential data.

We will consider REEL and its ongoing implementation within an existing compiler framework from the embedded system, in this case, the WCET-aware C Compiler (WCC). We will also explore the optimization potential, that the user can exploit during compilation by using REEL annotations within the source code.

This paper is organized as follows: Section 2 provides an overview of the related work and background on the compiler framework used. Section 3 describes the semantics and implementation of REEL Annotations. Section 4 talks about the integration of REEL annotations within WCC. Section 5 follows two different REEL annotations within WCC. We close with a conclusion and a brief look at future work.

2. BACKGROUND

2.1 Contract Specification Language

A Contract Specification Framework *Drive* is introduced by Brown et al. [2], which includes an Embedded Do-

main Specific Language, the Contract Specification Language (CSL). CSL allows the user to reason about the energy, time, and security properties of their code in an architecture-independent way. Reasoning about the worst, best, and average cases are possible for both time and energy cases. In case of security, CSL allows capturing the security level concerning fault injection, and power- and time-related side-channel attacks. The user needs to annotate their source code with predefined CSL-functions, that have to be placed before a regular C-statement to reason about the statement's non-functional properties.

The *Drive* framework can call external tools, such as WCC, for obtaining the necessary estimations and information about non-functional properties of the source code. REEL is being developed in conjunction with the University of St. Andrews, to have a usable interface between the two frameworks.

2.2 WCET-Aware C Compiler Framework

The WCET-Aware C Compiler (WCC) [4] is a C compiler consisting of sophisticated WCET-oriented analyses and optimizations. It currently supports the ARM7TDMI from ARM and TriCore TC1796 and TC1797 processors from Infineon. WCC is currently being extended to support the ARM Cortex-M0 and Leon3 from Cobham Gaisler.

WCC has two intermediate representations (IRs): one high-level C-like IR (ICD-C) and another assembler-like low-level IR (ICD-LLIR). The ICD-C framework is a data structure that provides a machine-independent IR for C code. On the other hand, ICD-LLIR is a data structure that provides a retargetable machine-dependent low-level IR for compiler back-ends. Various optimizations can be carried out on each IR-level. WCC offers WCET-aware optimizations during the compilation process due to its tight integration with a static WCET analyzer tool called *aiT* [1].

Additionally, WCC can perform WCET- and Energy-aware multi-objective optimizations. It can, also, perform ILP-based optimizations, to find optimal solutions that are derived deterministically, as described in, e.g., [5].

3. REEL ANNOTATIONS

CSL, a high-level language, is independent of any tool or compiler, and therefore a source code annotated with CSL is architecture-independent. To reason about physical and architecture-specific properties, such as energy, WCET, etc., it needs values or estimations for specific architectures. On the contrary, REEL is a collection of compiler intrinsic C-functions used to annotate C source code and provides estimations for specific architectures. REEL annotations sit between CSL and compiler level, offering an interface for obtaining such properties automatically in the future, as envisioned in [2].

REEL annotations enable the user to argue about the non-functional properties of the code. It is done by adding calls to the compiler intrinsic functions before regular C statements, much in the same way as is done in CSL. The user will be able to make decisions within their code, based on the nature of the non-functional properties. This information is injected into C-variables defined in the REEL function's argument-list, wherever it is appropriate. The user can also declare deadlines or limits on the energy consumption of a particular piece of code, by asserting within the source code. These REEL assertions will be treated as constraints while performing ILP-based optimization.

These annotations are expressed as regular ANSI-C compliant function calls in the source code. The user can annotate the source code with these REEL functions, defined in the REEL header file, which describes the meaning of each REEL annotation and provides information on their argument types. The function bodies for REEL annotations that provide their implementations are defined locally within the compiler. The user can inform the compiler about the important parts of the code in terms of non-functional properties using REEL annotations. For example, the following REEL annotations can inform the compiler that the user requires the WCET value of the code snippet immediately after the annotation.

```
1 void __reel_worst_time( unsigned long long *var
      );
```

Here, the variable `var` always refers to the WCET of the immediate next non-REEL C statement. If a user has some information about the non-functional properties of the code snippet, then the user can use the following annotation to inform the compiler about their values and the confidence level with which the user knows this information.

```
1 void __reel_worst_time_manual(
2                 unsigned long long *var,
3                 unsigned long long dValue,
4                 const _Bool cLevel );
```

Depending upon the confidence level `cLevel`, this annotation assigns either the user-provided default WCET value or a WCET value derived by a static WCET analysis to the variable `var`. Similar annotations can be used by the user if the user needs information in regards to the energy consumption, average-case execution time (ACET), etc. of the code snippets. Therefore, these values obtained by performing analyses can be used to make decisions within the application. These values should not be used within the analyzed code section itself, as it makes the analysis result dependent on itself.

Furthermore, `__reel_assert` provides C-level contracts to assert non-functional properties during compile-time.

```
1 void __reel_assert( _Bool expr );
```

This annotation provides a boolean C expression `expr`, which should be asserted by the compiler. The expression to be asserted is treated as integer-linear inequations within the compiler, which are never translated into actual machine code, contrary to other REEL annotations. This can help the user to deal with objectives like time and energy consumption for embedded systems at the source-code level. The user can define deadlines for embedded systems, which can be treated as integer-linear constraints while performing ILP-based optimizations within a compiler like WCC.

4. INTEGRATING REEL ANNOTATIONS INTO WCC

In this section, we describe the approach to realize REEL annotations within the WCC compiler framework, cf. also Figure 1. As mentioned before, WCC treats REEL annotations as compiler-known functions. The function bodies for REEL annotations that provide their implementations are defined locally within WCC in a dedicated C file. This

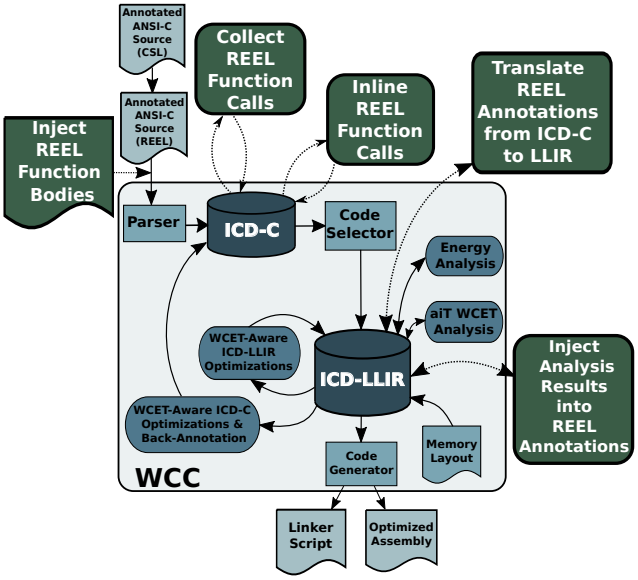


Figure 1: Integration of REEL Annotations within WCC

C file is transparently injected during WCC's compilation process, in the beginning, before the source files are parsed, as shown in Figure 1.

By using WCC's high-level intermediate representation ICD-C, REEL annotations within the source code are collected. To link these REEL annotation calls with their actual compiler-internal implementations, WCC inlines REEL function templates into which the analysis values are later injected. This way, all occurrences of REEL function calls are replaced by their respective function bodies, whose code is already injected previously within WCC using the C file.

Then ICD-C is translated into LLIR during WCC's code selection stage. For Leon and ARM architectures, WCC delegates code selection to external GCC instances, while for TriCore an internal implementation is used. As the external GCC is fully unaware of REEL annotations and WCC's internal mechanisms, it is very difficult to maintain a proper translation between the C-source level and the assembly-level. To tackle this issue, WCC exploits DWARF-2 debugging information [3]. DWARF-2 is a standardized extension of the binary object code that enables symbolic, source-level debugging. It provides an inherent mapping from assembly-level code to its original provenience at the source code-level. WCC propagates REEL annotations from ICD-C level to LLIR level by correctly exploiting the GCC-generated DWARF-2 debugging location information.

During code selection, it is necessary to properly identify the locations of code generated by the REEL function definitions. Therefore, WCC maintains a dictionary relating the ICD-C constructs to their LLIR counterparts. Every translation from an ICD-C basic block to an LLIR basic block is maintained with the help of a map. This way WCC can keep track of all the occurrences of these REEL annotations too at the assembly level. It is necessary to maintain this mapping so that the analyses for timing and energy can be performed at the assembly level.

WCC performs the required time and energy analyses at the LLIR level and annotates back this information to the LLIR at the granularity of individual basic blocks. Depending upon the scope of the REEL annotations within the LLIR, this information is aggregated such that it exactly covers the whole range of assembly-level code that the REEL annotation refers to at the source-level. For example, if a user requires information about the WCET value of a for-loop, WCC uses loop nesting to acquire a list of LLIR basic blocks associated with that for-loop and aggregates the WCET values of those basic blocks.

Once the aggregated data is collected at the LLIR level, WCC injects it into the assembly code that corresponds to the REEL construct and thus gets embedded into the binary. This data can also be exploited by the compiler optimizations, and thus allows the user to steer the compilation process.

5. EXAMPLE IMPLEMENTATION

5.1 REEL Worst Time

In this sub-section, we will consider a full implementation of one representative REEL annotation, i.e., `__reel_worst_time` defined in Section 3. WCC's internal implementation of this compiler-known REEL annotation looks as follows:

```
1 void __reel_worst_time( unsigned long long *var
      )
2 {
3   *var = 42;
4 }
```

This REEL annotation assigns WCET estimates to the variable `var`, given as a pointer parameter. A WCET value shall be assigned to this variable, which at first holds 42 as a dummy value. The WCET value assigned to this variable always refers to the WCET of the immediate next non-REEL C statement following the REEL annotation.

Consider the example C-source code from Section 1 annotated with a REEL annotation:

```
1 unsigned long long rWCET;
2 __reel_worst_time( &rWCET );
3 for ( i = 0; i < 100; ++i )
4   a +=2;
5 if ( rWCET < 10000 )
6   /* Execute energy-efficient mode. */
7 else
8   /* Execute high-performance mode. */
```

Here, a for-loop is annotated with a `__reel_worst_time` annotation. The value of variable `rWcet` is associated with the WCET of the following for-loop. Depending on the WCET value of the for-loop, the user is deciding either to perform an energy-efficient execution or to perform a high-performance execution. This code snippet is transformed via inlining into

```
1 unsigned long long rWCET;
2 *( &rWCET ) = 42;
3 for ( i = 0; i < 100; ++i )
4   a +=2;
5 if ( rWCET < 10000 )
6   /* Execute energy-efficient mode. */
7 else
8   /* Execute high-performance mode. */
```

If we are compiling this code for a ARM Cortex-M0 architecture, after code selection the above mentioned source code will be translated to the following assembly sequence:

```
.L1:
        mov     r6, #0          % b = 0
        mov     r3, #42         % rWCET = 42
        mov     r5, #0          % i = 0
        b       .L3
.L2:
        add     r6, r6, #2      % a += 2
        add     r5, r5, #1      % ++i
.L3:
        cmp     r5, #99         % i < 100
        bls     .L2
```

In the above assemble code snippet, the if-else statement is omitted for the sake of brevity. As it can be seen, the assembly-level basic block `.L1` features a `mov` instruction that corresponds to the original REEL annotation `__reel_worst_time( &rWCET );`. It assigns the dummy value of 42 to the register `r3` that holds the variable `rWCET`. This variable is supposed to represent the WCET of the entire for-loop in the above example. Thus, WCC performs a static WCET analysis of the generated assembly code by using *aiT*'s analyzer. If, for example, `aiT` determines that the WCET value of this for-loop is 420 clock cycles, WCC will finally update the `mov` instruction corresponding to the REEL annotation by

```
        mov     r3, #420        % rWCET = 420
```

Once this WCET value is replaced at the assembly code-level, this value can be accessed and used during run-time for various purposes like any other variable. In this example, the compiler could also discard the unused branch as soon as the value is known. Thus both cases can be taken into account during design time, without worrying about the properties of the final executable, as the compiler would make the right decision once the WCET has been determined. This makes the application more generic and easier to adjust for different target platforms or optimization levels.

5.2 REEL Assert

In this sub-section, we will consider a full implementation of `__reel_assert`, defined in Section 3. By using REEL assertion a user can provide a boolean C expression which is passed through WCC. WCC collects this boolean C expression and treats it as an integer-linear inequation.

A C-source code containing a REEL assert would look like as follows:

```
unsigned long long rWCET;
__reel_worst_time( &rWCET );
for ( i = 0; i < 100; ++i )
  a +=2;
/* The WCET value of the for-loop should be
    less than 10000. */
__reel_assert( rWCET < 10000 );
```

Unlike other REEL annotations, for `__reel_assert`, WCC collects the boolean expression and removes the assertion call from the source code. This boolean expression can be passed down to the ILP-based optimizations to be investigated in the near future. In the above-mentioned example, WCC collects the boolean expression `rWCET < 10000`. By using assert, the user is indicating that the WCET of the for-loop needs to be less than 10000. Once WCC performs an initial WCET analysis, the WCET value of the for-loop, associated with `__reel_worst_time`, is injected within the source.

Once `rWCET` is known, a linear-inequation is added to the ILP model as a constraint. The optimization then tries to find a solution that fulfills this assertion. If the assertion is not fulfilled by any possible solution, WCC will let the user know that the assertion is unrealistic, and a solution does not exist in such a scenario.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented REEL annotations, with which the user can argue about and obtain information about non-functional properties, like WCET, Energy, and security, at the source code level via annotations. It enables the user or other tools to obtain information about non-functional properties, and steer the compilation and optimization process. It also allows the user to make use of these non-functional properties, as the information is injected into user-defined variables to be used within the application.

We also discussed our implementation of REEL within a WCET-aware C compiler and showed an example implementation of two different REEL annotations. Furthermore, we showed that REEL assertions can be used by the user to provide constraints on the non-functional properties within the source code. These constraints will be taken into consideration by WCC while performing ILP-based optimizations, and WCC will try to find optimized solutions based on user-provided assertions.

As a next step, we will evaluate REEL annotations on a number of benchmarks and perform ILP-based optimizations to test our techniques. We plan to extend our work by taking into consideration other non-functional properties like energy. Our intention is also to extend the WCC and REEL framework to take security aspects of the code into consideration and see its behavior against WCET and Energy.

Acknowledgments

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779882.

7. REFERENCES

[1] G. AbsInt Angewandte Informatik. ait worst-case execution time analyzers, 2019.

[2] C. Brown, A. D. Barwell, Y. Marquer, C. Minh, and O. Zendra. Type-driven verification of non-functional properties. In *PPDP*, 2019. Accepted for publication.

[3] Dwarf debugging information format. `http://dwarfstd.org/`, 1993.

[4] H. Falk and P. Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2):251–298, 2010.

[5] D. Oehlert, A. Luppold, and H. Falk. Bus-aware static instruction spm allocation for multicore hard real-time systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

InterNoC: Unified Deterministic Communication For Distributed NoC-based Many-Core

Eleftherios Kyriakakis
Technical University of Denmark
Kgs. Lyngby, Denmark
elky@dtu.dk

Jens Sparsø
Technical University of Denmark
Kgs. Lyngby, Denmark
jspa@dtu.dk

Martin Schoeberl
Technical University of Denmark
Kgs. Lyngby, Denmark
masca@dtu.dk

ABSTRACT

Network-on-Chip is a popular paradigm for scalable many-core communication. There is a trend in modern system-on-chip to integrate more functionality. This combined with recent research for network-on-chip in the aerospace industry, gives room for design space exploration in new architectural paradigms for distributed and real-time many-core communication. In this paper, we present InterNoC, a deterministic communication scheme for distributed network-on-chip many-core that allows for unified IP-based time-triggered communication. It is hypothesized that such an architecture will efficiently minimize communication complexity in distributed many-core systems as well as provide hard-bounded end-to-end latency guarantees. We extend the real-time multi-core platform T-CREST by introducing a time-triggered NoC-based switching mechanism combined with a NoC packet to Ethernet frame traffic controller. The proposed architecture will be evaluated in an experimental InterNoC network that implements a 36-core real-time system distributed over four FPGA devices.

1. INTRODUCTION

The shift to chip multi-processors (CMPs) in real-time systems is inevitable as emerging technologies in the field of industrial automation and industrial internet of things continue to increase the demand for performance on the edge of the network while maintaining strict requirements in low power consumption [12]. To meet these requirements, the number of cores in modern many-core system-on-chip continues to scale and thus the requirements on resources and bandwidth of traditional buses become demanding. Network on-chip (NoC) have proven to be a viable solution for scalable many-core on-chip interconnection as they allow for efficient all-to-all communication [4].

Most research in the field of NoC focuses on improving the functional characteristics of the architecture as well as analyzing the application mapping strategies to such many-core systems. Over the years, few works have been presented that discuss the use of NoC for inter-chip communication, particularly in the field of real-time systems. In this paper, we envision a prototype network architecture that allows for a unified deterministic communication scheme for distributed real-time many-core systems. It is hypothesized, that such an architecture will allow real-time many-core applications to deploy tasks on cores in a locality agnostic way. Essentially, a task will execute the same code regardless if it needs to communicate with other tasks found on- or off-chip. We identify two possible ways to extend a NoC over physically distributed CMPs: 1) the local NoC communication (i.e. flits) gets encapsulated in an inter-chip communication protocol or 2) the Ethernet network protocols are pulled into the local NoC communication, abstracting away any on- and off-chip communication details from the application. In this paper, we focus on investigating the implementation of the second option as it abstracts away from the application any details about the internal architecture and routing of the NoC and allows to integrate heterogeneous systems.

The contribution of this paper is a first exploration of the idea of distributed CMP real-time systems that aims to provide seamless and scalable inter-chip time-triggered communication using the Internet Protocol (IP). It is based on the time-division multiplexed NoC architecture Argo, the time-predictable processor Patmos and an Ethernet controller extension. The communication is realized by implementing a new software abstraction layer between NoC packets and Ethernet IP frames. Bounded latency guarantees and contention-free communication is achieved by enforcing a network-wide time-triggered schedule.

The rest of the paper is organized in 7 sections: Section 2 provides the user with a background of the related work in the field of NoC-based inter-chip communication. Section 3 provides the reader with a background on the fundamental components the proposed communication is built on. Section 4 presents the proposed architecture and describes its integration with the T-CREST platform. Section 5 provides a preliminary evaluation of the feasibility and characteristics of the proposed architecture. Section 6 describes the future scope of this project and the immediate research focus points. Finally, Section 7 concludes the paper.

2. RELATED WORK

Inter-chip communication has been previously explored in the context of distributing the processing resources of a NoC and the following works present different techniques for implementing an off-chip bridge for NoC system-on-chip [9, 11, 20]. In this work, we implement Ethernet as the physical link for the inter-chip NoC communication as it allows for greater interoperability that can take advantage of different industrial Ethernet technologies. It also facilitates for software development as applications can communicate using a single protocol regardless if they are on the same chip or distributed over multiple devices.

More recently, in [1] the authors propose a mixed avionics full-duplex Ethernet (AFDX) and NoC communication between many-core applications. The proposed communication model is motivated by real-time use-cases in the field of avionics and health monitoring the authors investigated different application mapping strategies concerning the effects of contention on the worst-case traversal time of communication flows. Moreover, in [2] the authors propose a function mapping strategy in mixed AFDX/NoC communication that aims to minimize flow contentions and thus provide bounded jitter.

Another related distributed real-time system is presented in [19]. The authors propose and implement the Mock Turtle architecture,

which is based on 32-bit RISC-V processor cores. On-chip communication is provided by a shared memory mechanism while off-chip communication is implemented using the white-rabbit network [16]. As shared memory communication does not scale well and with a cache coherence protocol is hardly time predictable, we use a NoC for on-chip communication.

Finally, in contrast to previous works the proposed architecture does not suffer from contention as it is based on a network-wide time-triggered schedule and a time-division multiplexing (TDM) NoC design. The software driver handling the time-triggered communication is implemented on the time-predictable Patmos processor allowing for worst-case execution time (WCET) analysis. The building blocks of this architecture allow for a bounded end-to-end communication latency that can be statically calculated as shown in Section 5.

3. BACKGROUND

This section aims to briefly introduce the existing architecture components, namely the processor and the NoC architecture, which the proposed InterNoC design uses for the implementation. Both of them are part of the open-source research platform T-CREST [13], which is an FPGA-based many-core platform that has been developed for on-going research in real-time applications.

3.1 Argo NoC

Argo [17] is a packet-switched and source-routed NoC that uses TDM to guarantee bandwidth and latency. It allows for deterministic unicast message-passing communication over virtual channels, which is implemented using dedicated direct-memory access (DMA) controllers for each source-end of every virtual channel. The DMAs are integrated with a TDM mechanism in the network-interface (NI), eliminating the need for buffering and flow control. According to the TDM schedule, each virtual channel gets a time-slot during which it can start sending a message. These time-slots are assigned according to a statically scheduled period.

The TDM mechanism is implemented in each Argo NI as TDM counter. This TDM counter indexes a schedule table that in turn points to an entry in the DMA table containing the respective counters and pointers for a DMA access. The NI exposes two interfaces to the processor core, a configuration interface and a data interface. The data interface allows the processor to exchange data with the NI's dual-port scratch-pad memory. The configuration provides access to both the schedule table as well as the DMA controller.

It is worth noting, that Argo supports on-the-fly schedule reconfiguration. This can be done with zero delays experienced by any on-going messages in a virtual channel communication as described in [17], but using this mechanism is outside the scope of this work.

3.2 Patmos

Patmos [15] is a time-predictable WCET-optimized, dual-issue RISC processor that has been designed with a focus on WCET analysis. It uses special WCET-optimized instruction and data caches along with private scratchpad memories for instructions and data. It is supported by an LLVM-based [10] compiler, also optimized for WCET and by the WCET analysis tool *platin* [6].

The tool *platin* performs static analysis to compute the WCET of a certain code segment by using the information generated and preserved during compilation to determine a control flow graph. Together with low-level timing information of the processor architecture, it can calculate a safe WCET bound of the analyzed code segment.

4. ARCHITECTURE

The architecture of InterNoC is building on top of the statically scheduled TDM Argo NoC (see Section 3.1), but the communication concept can be applied to any NoC architecture as long as it can guarantee bandwidth allocation. Moreover, the NoC cores are implemented using the Patmos processor, allowing for time-predictable and statically WCET analyzable execution of software tasks. This combination allows the proposed architecture to have a fully deterministic communication scheme.

The proposed architecture defines two types of NoC cores, (1) a *computation core* and (2) a *gateway core* that is interfaced with an Ethernet controller and is responsible for handling incoming and outgoing traffic. This is illustrated in Figure 1, where *core 0* is assigned the role of the *gateway core*.

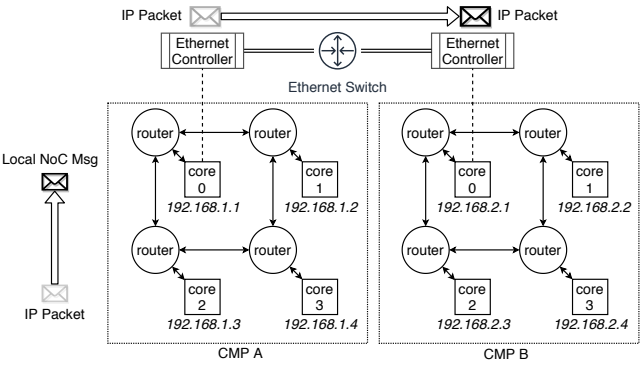


Figure 1: Local and distributed NoC communication concept over IP

Taking advantage of existing Ethernet technologies, communication is performed as a mix of NoC packets and IP traffic. Each core is statically assigned an IP address and the message-passing library of Argo is extended to allow for this translation between from IP packets to NoC messages.

When an application wants to transmit a message it constructs an IP packet and passes the data to the driver. Consecutively, the driver checks if the subnet of the requested IP address is on- or off-chip. If the destination is on-chip it uses the respective NoC virtual channel and writes the IP packet to the corresponding DMA to reach the destination *computation core*. If the destination subnet IP address is off-chip, the driver uses the respective NoC virtual channel pointing to the *gateway core* and writes the IP packet to the respective buffer.

Each *gateway core* executes two periodic tasks. The first task is responsible for collecting incoming IP packets from the on-chip *computing cores*, encapsulating them into Ethernet frames and forwarding them at specific time-slots using the Ethernet controller. The second task is responsible for polling the Ethernet controller for incoming IP packets and depending on the destination IP address creating the NoC channel and transmitting the incoming IP packet to the respective NoC node.

Due to the difference in time granularity between the local NoC schedule and the Ethernet transmission, each *gateway core* performs a store-and-forward scheme and maintains separate buffers per NoC channel for the incoming NoC packets, which it needs for to reconstructs the respective IP packets. For example, if a NoC packet can transfer a maximum of two words of data (64 bits) per NoC TDM slot, then the on-chip transmission of a maximum size Ethernet frame (1518 bytes) requires 190 NoC TDM slots to complete. This is further discussed in Section 5.

All the components of the presented architecture are developed as open-source and are hosted under the T-CREST project GitHub repository `https://github.com/t-crest/`.

5. EVALUATION

This section provides an initial evaluation of the scalability and feasibility of the presented architecture. The proposed communication is composed of two types of traffic:

1. Intra-CMP, referring to the on-chip NoC communication
2. Inter-CMP, referring to the off-chip communication that processing cores can access via Ethernet.

A full evaluation of the intra-CMP communication for Argo has been already presented in [14, 18]. In this evaluation, we try correlate the results with the proposed IP-NoC abstraction layer's added overhead and the additional inter-chip communication scheme.

Figure 2 presents the constraints for the inter-chip time-triggered communication that define the schedule period (length). These are the total number CMPs, the number of cores per CMP and the number of frames available for transmission/reception by each core.

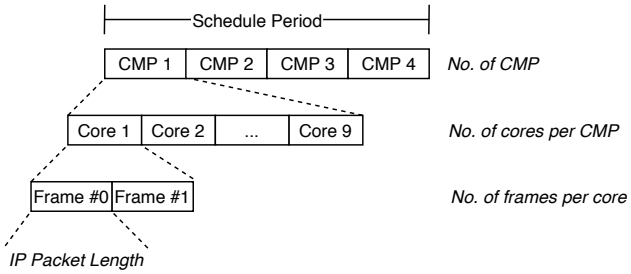


Figure 2: Constraints defining the inter-CMP schedule period

The evaluation is motivated by three industrial Ethernet traffic classes, defined in [5], that specify the schedule period requirements. The available bandwidth for inter-CMP communication is assumed to be 100 Mbps, as this is commonly used in real-time communication protocols such as the TTEthernet [8]. Table 1 presents the number of frames per schedule period that can be exchanged using the inter-chip links, by taking into account two types of frames, 1) full Ethernet frames of 1518 bytes and 2) minimum Ethernet frames of 64 bytes.

Table 1: Available frames versus inter-CMP schedule period

Class	Schedule Period (ms)	No. of Frames	
		Full frames	Min. frames
1	100	833	19531
2	10	83	1953
3	0.25 – 1	2 – 8	48 – 195

In most industrial use-cases the period of real-time traffic constraints the frame size. Assuming a hard real-time schedule period of 1 ms that aims to cover an all-to-all communication schedule, with min. frame size of 64 bytes (6 bytes of IP payload). This means that M CMPs containing N cores share a bandwidth of 195 frames or 1.17 MBps that each carries 6 bytes of IP payload. For example, a system composed of $M = 4$ then each CMP has 44 time-slots available for inter-CMP communication. This can be seen as a trade-off between the number of cores per CMP and the assigned bandwidth to each CMP core. The application designer is responsible for minimizing the off-chip communication flows. The subject of task mapping strategy is outside the scope of this work.

The worst-case traversal time (WCTT) of the proposed architecture is end-to-end bounded since the Argo NoC operates on a TDM schedule and the planned inter-CMP communication is time-triggered. Thus, the WCTT of a packet in the presented architecture can be statically expressed by $WCTT_{e2e}$ in Equation 1. The $WCTT_{e2e}$ is composed of three parts, the worst-case traversal time of the NoC as $WCTT_{noc}$, the worst-case traversal time of the Ethernet link as $WCTT_{eth}$ and the worst-case execution time $WCET_{driver}$ of the abstraction layer that performs the store-forward and the translation between NoC and IP packets.

The WCTT of the NoC is presented in Equation 2. $WCWT_{noc}$ is the worst-case waiting time that a NoC packet can experience and is equal to the period of the NoC TDM schedule, $Size_{IP}$ is the size of the transmitted/received IP packet, B_{noc} is the bandwidth of the NoC channel, L_{noc} is the end-to-end latency of the NoC and T_{noc} is the NoC schedule period.

Finally, the worst-case traversal time over Ethernet is presented in Equation 3. The $WCWT_{eth}$ is the worst-case waiting time an Ethernet frame can experience before starting the transmission, $Size_{eth}$ is the size of the Ethernet frame, B_{eth} is the available bandwidth of the Ethernet connection and L_{switch} is the Ethernet switch latency.

$$WCTT_{e2e} = WCTT_{eth} + 2 * (WCTT_{noc} + WCET_{driver}) \quad (1)$$

$$WCTT_{noc} = WCWT_{noc} + \frac{Size_{IP}}{B_{noc}} * T_{noc} + L_{noc} \quad (2)$$

$$WCTT_{eth} = WCWT_{eth} + 2 * \frac{Size_{eth}}{B_{eth}} + L_{switch} \quad (3)$$

Following is a short example of an experimental setup composed of four CMPs with nine cores each communicating over a single Ethernet switch. Assuming a bi-torus topology of the NoC we can safely assume that the worst-case latency L_{noc} is equal to one TDM NoC period. For a 3x3 Argo NoC the schedule period is 10 clock cycles [14]. Furthermore, we assume that the architecture is implemented on FPGA technology with 80 MHz clock. To calculate the $WCTT_{e2e}$ between two CMPs, first, we calculate the WCTT for Min. size IP frame, for each NoC, to be $WCTT_{noc} = 1.25\ \mu s$. Secondly, we calculate the WCTT for Ethernet, assuming $B_{eth} = 100Mbps$ and $L_{switch} = 3.9\ \mu s$ [7]. According to Table 1 the system can support a schedule period of 0.25 ms, where each core is allowed to transmit one frame per period, thus $WCTT_{eth} = 260.24\ \mu s$. Finally, we calculate $WCTT_{e2e} = 266.64\ \mu s + WCET_{driver}$. In future works, a WCET analysis of the abstraction layer driver will allow for the exact calculation of the end-to-end WCTT.

6. DISCUSSION AND FUTURE WORK

The proposed architecture is hypothesized to allow building Internet applications on-top of a deterministic IP-NoC abstraction layer that efficiently connects distributed real-time CMPs.

We plan to evaluate the communication on an experimental setup that implements a 36-core architecture distributed over four physically separate devices, similar to what was visualized in Figure 2. The system is implemented in four Altera Cyclone IV FPGA devices [3] that each implements a 3x3 NoC and communicate over Ethernet using a standard off-the-shelf switch. Such a system would be otherwise impossible to fit on a single FPGA device, something that emphasizes the importance of this communication scheme for

FPGA-based many-core real-time systems. We plan to implement the experimental setup on a TTEthernet network and integrate the off-chip time-triggered communication with the TTEthernet schedule. We will investigate distributed systems schemes such as client-server, publish-subscribe and remote process call built on top-of-the presented communication scheme.

Finally, since the abstraction layer implements IP, it allows for the physical communication link to be extended to other technologies such as wireless. However, this requires different mechanisms to guarantee determinism which are outside the scope of this work.

7. CONCLUSION

In this paper, we proposed a new paradigm of unified communication for NoC-based CMPs. The proposed architecture allows to minimize communication complexity and improve real-time systems scalability by extending the TDM Argo NoC using an IP Ethernet frame to NoC packets abstraction layer and a time-triggered off-chip communication scheme. The presented communication scheme extends the Argo message-passing library and the developed software API is statically WCET analyzable as it is implemented on the time-predictable processor Patmos.

A first evaluation of the scalability and design constraints of the proposed communication was presented. It revealed that an application mapping strategy must aim to minimize off-chip communication links, if hard real-time cycle time needs to be guaranteed. Furthermore, it was shown that the end-to-end communication delay can be guaranteed in a statically analyzable way.

8. ACKNOWLEDGEMENTS

This research has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation.

9. REFERENCES

[1] L. Abdallah, J. Ermont, J.-L. Scharbarg, and C. Fraboul. Towards a mixed noc/afdx architecture for avionics applications. In *2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS)*, pages 1–10. IEEE, 2017.

[2] L. Abdallah, J. Ermont, J.-L. Scharbarg, and C. Fraboul. Reducing afdx jitter in a mixed noc/afdx architecture. In *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 1–4. IEEE, 2018.

[3] ALTERA. *Cyclone IV FPGA Device Family Overview*, March 2016.

[4] L. Benini and G. De Micheli. Networks on chips: A new soc paradigm. *computer*, 35(1):70–78, 2002.

[5] P. Danielis, J. Skodzik, V. Altmann, E. B. Schweissguth, F. Golatowski, D. Timmermann, and J. Schacht. Survey on real-time communication via ethernet in industrial automation environments. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8. IEEE, 2014.

[6] S. Hepp, B. Huber, J. Knoop, D. Prokesch, and P. P. Puschner. The platin tool kit - the T-CREST approach for compiler and WCET integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

[7] Hewlett-Packard Development Company, L.P. *ProCurve Switch 1700 Series*, March 2007.

[8] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The time-triggered ethernet (tte) design. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 22–33. IEEE, 2005.

[9] E. Kyriakakis, K. Ngo, and J. Öberg. Implementation of a fault-tolerant, globally-asynchronous-locally-synchronous, inter-chip noc communication bridge on fpgas. In *2017 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–6. IEEE, 2017.

[10] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–88. IEEE Computer Society, 2004.

[11] W. H. Minhass, J. Öberg, and I. Sander. Implementation of a scalable, globally plesiochronous locally synchronous, off-chip noc communication protocol. In *2009 NORCHIP*, pages 1–5. IEEE, 2009.

[12] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin. The shift to multicores in real-time and safety-critical systems. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 220–229. IEEE Press, 2015.

[13] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[14] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pages 152–160. IEEE, 2012.

[15] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, and D. Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Apr 2018.

[16] J. Serrano, M. Lipinski, T. Wlostowski, E. Gousiou, E. van der Bij, M. Cattin, and G. Daniluk. The white rabbit project. 2013.

[17] R. B. Sørensen, L. Pezzarossa, M. Schoeberl, and J. Sparsø. A resource-efficient network interface supporting low latency reconfiguration of virtual circuits in time-division multiplexing networks-on-chip. *Journal of Systems Architecture*, 74(Supplement C):1–13, 2017.

[18] J. Sparsø, E. Kasapaki, and M. Schoeberl. An area-efficient network interface for a tdm-based network-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1044–1047. EDA Consortium, 2013.

[19] T. Wlostowski, F. Vaga, and J. Serrano. Developing distributed hard-real time software systems using fpgas and soft cores. 2015.

[20] Y. Yin and S. Chen. Design and implementation of a inter-chip bridge in a multi-core soc. In *2009 4th International Conference on Design & Technology of Integrated Systems in Nanoscal Era*, pages 102–106. IEEE, 2009.

A Preliminary Examination of Schedulability under Lock Servers*

Catherine E. Nemitz
The University of North Carolina at Chapel Hill
nemitz@cs.unc.edu

ABSTRACT

Allowing nested resource access in a real-time system introduces several challenges. Addressing these challenges within a synchronization protocol often leads to significant protocol overhead. Recently, a protocol-independent method was developed that significantly reduces this overhead; lock servers manage the execution of complex protocols, largely independently from the tasks that require resource access. However, some lock server configurations change the blocking caused by the underlying protocol. This leaves an as-of-yet unanswered question: how does the use of lock servers impact schedulability? I present a preliminary examination of that question and briefly explore how the assignment of tasks to lock servers can impact schedulability.

Keywords

multiprocessor locking protocols, nested locks, real-time locking protocols, priority-inversion blocking

1. INTRODUCTION

Real-time systems, those which require timing guarantees as a component of system correctness, require an efficient synchronization protocol to enable safe resource sharing while meeting deadlines. A particular challenge to protocol efficiency is the presence of nested resource requests, which occur in real-world systems [1, 3, 5] when multiple resources must be held simultaneously.

Synchronization protocols are necessary to ensure safe resource sharing, but contribute two fundamental types of delay to task execution. *Blocking* occurs when a task must wait due to the protocol managing access to resources. Blocking varies between protocols based on how each protocol orders tasks to grant resource access. The other delay introduced by locking protocols is *overhead*—the time required to execute the protocol logic and determine which task(s) may be granted access to resources at each time.

Until recently, approaches taken by locking protocols to handle nested resource access either (i) artificially limit nesting [8, 14], (ii) may cause significant blocking [6, 7, 9, 14, 15], or (iii) cause significant overhead [10]. A recently developed method [12] allows the reduction of protocol overhead and, depending on the configuration, may change the computation of worst-case blocking. This reveals a need—to fully examine the tradeoffs of overhead and blocking under this new approach *in the context of schedulability.*

In light of this, I take the uniform variant of the contention-sensitive real-time nested locking protocol (U-C-RNLP), as a case study and explore the impacts of the various lock server configurations on schedulability. While one lock server configuration tends to outperform all others in this preliminary study, the results also show that with better accounting techniques and methods of task allocation, the other configurations may lead to higher schedulability in more scenarios.

Organization. I begin by giving necessary background on the system model and on related work Sec. 2. Next, I present preliminary schedulability results Sec. 3. Finally, I conclude and present directions for future work in Sec. 4.

2. BACKGROUND AND PRIOR WORK

I begin by describing the system model before giving relevant details about the different lock server configurations and the functionality of the U-C-RNLP.

System Model. In this paper, I assume the standard sporadic task model, in which an arbitrary task is denoted τ_i. As described in more detail in Sec. 2, clustered scheduling is required for some lock server configurations, so I assume Clustered Earliest-Deadline-First (C-EDF) scheduling.

When a task τ_i requires access to one or more resources, it issues a request $\mathcal{R}_i$. I focus on a spin-based locking protocol, in which τ_i busy waits until it is granted access to its resources, after which it executes non-preemptively until completing at most L_i time units later. The maximum critical-section length is denoted L_{max}.

Lock Servers. Lock servers [12] build on an idea fundamental to remote core locking [11] by isolating the execution of the lock logic to a few processes to better utilize the cache(s). This is the mechanism that allows such a drastic reduction in overhead. There are four fundamental lock server types, which are distinguished by *locality* and *mobility.*

Let us begin by assuming a single lock server and exploring the two types of mobility. A *static* lock server is pinned to a single core. In contrast, a *floating* lock server moves between cores. More specifically, when a task is busy-waiting for access to its required resource(s), it can instead assume the role of the lock server; until it is satisfied, it cannot continue its execution, but it can execute the lock logic on behalf of

*Work supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGS-1650116. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Protocol	Worst-Case Acquisition Delay	Overhead (μs) on Platform 1	Overhead (μs) on Platform 2
U-C-RNLP	$(c_i + 1) \cdot L_{max}$	23.5	29.2
U-C-RNLP + SGLS	$(c_i + 1) \cdot L_{max}$	13.5	8.0
U-C-RNLP + SLLS	$(c_{i,s} + 1)(L_{max,1} + L_{max,2})$	8.7	2.8
U-C-RNLP + FGLS	$(c_i + 1) \cdot L_{max}$	11.5	9.1
U-C-RNLP + FLLS	$(c_{i,s} + 1)(L_{max,1} + L_{max,2})$	10.8	3.1

Table 1: Blocking bounds and overhead of each lock server configuration with the U-C-RNLP.

other tasks. In general, a static lock server will result in lower overhead than a floating lock server, as eliminating mobility can allow lock state to remain in the L1 cache.

Now let us consider the locality options. A *global* lock server executes the protocol logic on behalf of all tasks, while a *local* lock server handles only a subset of all tasks. The local lock server configurations originally presented [12] designate one lock server per CPU socket. A set of local lock servers tend to have lower overhead than a global lock server, as some level of cache affinity can be maintained. Combining both distinctions, a set of static local lock servers will have the lowest overhead of the four possible configurations. However, the use of local lock servers comes with a tradeoff: an additional synchronization mechanism is required in order to ensure tasks managed by different lock servers execute safely. This changes the worst-case blocking for the protocol used, which I describe in more detail after presenting the basic functionality of the locking protocol I consider.

U-C-RNLP. Though lock servers provide a means for reducing overhead that is protocol independent, they require use of some protocol. For this work, I focus on the Uniform C-RNLP (U-C-RNLP) variant of the C-RNLP [10].

The U-C-RNLP maintains a table of waiting and satisfied requests that indicates when each request will be satisfied. When a new request is issued, it is added to the first ("earliest") row in which there are no requests for an overlapping set of resources. Entire rows are satisfied concurrently; when a request completes, it is removed from the table and, if it was the last request in its row, it indicates that any requests in the subsequent row may become satisfied immediately.

The worst-case blocking for a request $\mathcal{R}_i$ handled by the U-C-RNLP is thus dependent on the number of other requests that conflict (require one or more of the same resources) with $\mathcal{R}_i$. This is the *contention* that $\mathcal{R}_i$ may experience and is denoted c_i. This leads to the bound of the U-C-RNLP (without any lock server) shown in Table 1. Essentially, there are at most c_i+1 rows of requests that will be satisfied before $\mathcal{R}_i$ is satisfied, and it may take up to L_{max} time units for all requests of a given row to complete.

When lock servers are used, the worst-case acquisition delay depends on the configuration. With a single lock server, as with the Floating Global Lock Server (FGLS) or the Static Global Lock Server (SGLS), the blocking remains unchanged from that of the basic U-C-RNLP protocol without lock servers. However, with multiple lock servers, like with the Floating Local (FLLS) and Static Local (SLLS) configurations, additional coordination is required between the different lock servers before resource access may be granted.

Refining the notion of a maximum critical-section length per local lock server allows a tighter bound to be computed. The maximum critical-section length of any request managed by Lock Server 1 (resp., Lock Server 2) is denoted $L_{max,1}$ (resp., $L_{max,2}$). Additionally, I refine the contention of a request $\mathcal{R}_i$ to specify the number of contending requests also served by Lock Server s, which is denoted $c_{i,s}$. The bounds on acquisition delay for the U-C-RNLP with each lock server configuration is shown in Table 1.

3. SCHEDULABILITY ANALYSIS

In this section, I describe my evaluation methodology and present the results. This evaluation is centered around the question of schedulability, but synchronization protocol overhead and blocking must first be accounted for. I computed both of these components and then incorporated them into the open-source schedulability toolkit SchedCAT [2].

3.1 Overhead

Synchronization protocol overhead is platform dependent. To accurately compare the tradeoffs of different lock server configurations, overhead must be computed on any test platform. For this preliminary investigation, I explore two platforms. Platform 1 is a dual-socket, 8-cores-per-socket Intel CPU platform. The second platform I consider, Platform 2, is a dual-socket, 18-cores-per-socket Intel CPU platform. Both platforms have three cache levels, with the lowest level shared across an entire socket, but not between sockets. Based on this two-socket structure, for the local lock server configurations, I use one local server per socket.

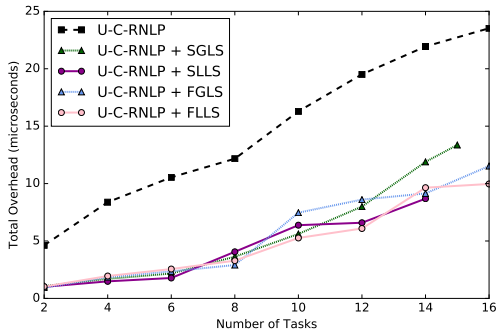
In order to measure the overhead of each configuration, I used the methodology described in [12]. I varied critical-section lengths, testing $L_i \in \{1, 15, 100\}\mu$s with nesting depth of either 2 or 4. The highest overhead for each configuration tended to be for nesting depth of 2 and $L_i = 1\mu$s, which is depicted in Fig. 1 for both platforms. The highest overhead values I measured are recorded in Table 1; these are the values I incorporated into the schedulability study by inflating the execution time of a task.

3.2 Blocking

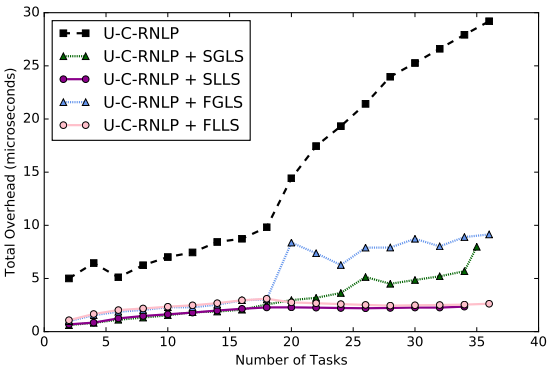
To compute blocking, I use the bounds presented in Table 1. In the implementation of these computations in the SchedCAT framework, I also made a refinement to tighten the analysis, by applying a period-based constraint [13], as it may not always take L_{max} for each row of requests to complete. Instead, I use a set of the largest critical-section lengths; I account for the number of times each critical section could delay $\mathcal{R}_i$ based on the relative periods of the tasks. Once I have computed the blocking a task may encounter, I inflate its execution time by this amount.

3.3 Schedulability

The taskset under analysis is divided into clusters by a Worst-Fit bin packing heuristic, which considers the utilization of a task to be its "weight" and the bin size to be $U = 1$.



(a) Platform 1



(b) Platform 2

Figure 1: Total protocol overhead.

Category	Name	Value
Critical-Section Length (μs)	Short	[1,15]
	Bimodal	[1,15] or [15,100]
	Moderate	[15,100]
Period (ms)	Short	[3,33]
	Moderate	[10,100]

Table 2: Named parameter distributions.

After accounting for the blocking and overhead a task may incur, I apply Baruah's G-EDF schedulability test [4] to each cluster individually. If all clusters pass this test, the taskset is deemed schedulable.

I examined a range of scenarios; the parameter values are summarized in Tables 2 and 3. I focused on nested requests; non-nested requests can be handled efficiently by other means [13]. Also, I assumed each task issues at most one request. For each value of system utilization considered, 100 tasksets were examined. To analyze the static lock server configurations, I assumed an entire core was dedicated to each server. I reevaluate this pessimistic decision below.

Platform 1. I begin with a series of observations from the 72 scenarios explored on Platform 1. Fig. 2 represents one such scenario, in which tasks have short periods, 50% of tasks issue a request, and the nesting depth is 2.

Obs. 1. *The schedulability of the U-C-RNLP with the FGLS is always as good or better than that of the U-C-RNLP with no lock server.*

Category	Options
Task Utilization	[0.1,0.4]
Period	Short, Long
Percentage Issuing Requests	5%,10%,20%, 50%, 80%, 100%
Critical-Section Length	Short, Bimodal, Moderate
Number of Resources	64
Nested Probability	1.0
Nesting Depth	2, 4

Table 3: Schedulability study parameter choices. For each range of values, a value is selected uniformly at random.

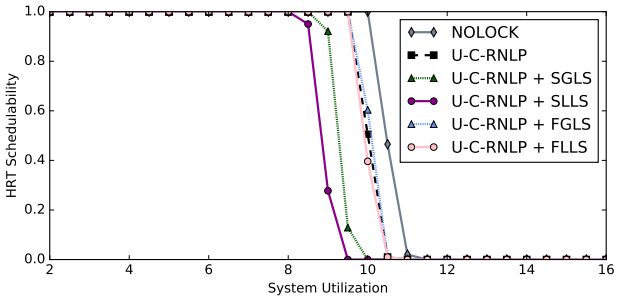


Figure 2: Schedulability on Platform 1 under a scenario in which tasks have short periods, 50% of tasks issue a request, and the nesting depth is 2.

This is illustrated in Fig. 2, and is as expected; these two protocol configurations have the same blocking, but the FGLS leads to reduced overhead.

Obs. 2. *The performance of the FLLS relative to the baseline U-C-RNLP is dependent on the percentage of tasks that require resource access.*

If the percentage of tasks issuing requests is at most 10%, FLLS is as good or better than the baseline in 98.6% of scenarios. If instead 50% or more of the tasks issue requests, FLLS is worse than the baseline in 93.1% of scenarios.

Obs. 3. *The SLLS is always the worst option, and the SGLS is almost always the second worst.*

These poor results despite significantly lower overhead highlights the need to develop a better method of ensuring rapid lock server response to newly issued requests while also allowing an analysis method that is not overly pessimistic. For example, a high-priority task could be dedicated to this without requiring the dedication of an entire core.

Platform 2. Next I considered performance on Platform 2. The same scenario depicted in Fig. 2 for Platform 1 is shown in Fig. 3 for Platform 2.

Obs. 4. *Similar performance trends hold for Platform 2.*

Relative to the baseline, the FGLS and FLLS are more dominant on Platform 2 for systems with requests with short critical-section lengths; FGLS is better than the baseline in 95.8% of the scenarios and if at most 50% of tasks issue requests, the FLLS configuration is always better than the baseline. However, for other critical-section lengths, the

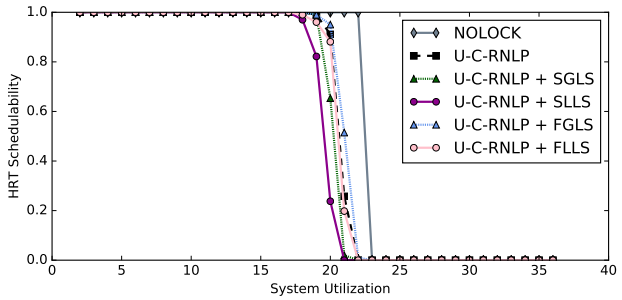


Figure 3: Schedulability on Platform 2 under a scenario in which tasks have short periods, 50% of tasks issue a request, and the nesting depth is 2.

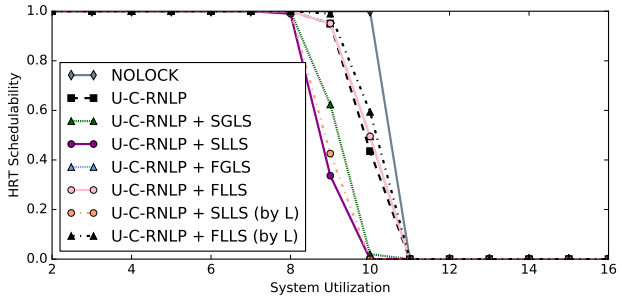


Figure 4: Schedulability on Platform 1 under a scenario in which tasks have short periods, 10% of tasks issue a request, and the nesting depth is 4. (Note: the U-C-RNLP + FGLS line is equivalent to that for the U-C-RNLP + FLLS.)

FLLS performs less well relative to the baseline on Platform 2, and is as good or better than the baseline only when 5% of tasks issue requests.

3.4 Methods for assigning tasks to clusters

I developed a basic heuristic to attempt to reduce blocking for local lock server configurations by leveraging the per-cluster definitions of maximum critical-section length. I explore this with a set of scenarios with bimodal critical-section lengths and for all percentages of tasks issuing requests except 100% (my assignment method performs poorly even at 80%). I assigned any τ_i with $\mathcal{R}_i$ with $L_i \leq 15\mu$s to Cluster 1 and τ_i with $\mathcal{R}_i$ with $L_i > 15\mu$s to Cluster 2. All tasks that did not issue requests were then assigned with Worst-Fit.

Obs. 5. *Different assignment techniques greatly impact the schedulability under local lock server configurations.*

For the FLLS configurations, switching to the assignment heuristic described above clearly improved schedulability in 40.0% of the 20 scenarios, tested on both Platform 1 and Platform 2. Similarly, the SLLS improved in 32.5% of scenarios. (These improvements ignore the 32.5% of scenarios for both platforms in which there was no significant change.)

Switching assignment heuristics also resulted in scenarios in which the FLLS with the new critical-section-length-dependent allocation method (labeled "by L" in Fig. 4) outperformed all other configurations. One such scenario is depicted in Fig. 4, in which tasks had short periods, 10% of tasks issued a request, and the nesting depth was 4.

4. CONCLUSIONS

I have explore the impact of lock server configurations on task system schedulability. While a single configuration, the Floating Global Lock Server, emerged as most effective in this preliminary study, the performance of other configurations can clearly be improved. With better methods for assigning tasks to clusters, schedulability under local lock servers improved. Additionally, more fine-grained methods for accounting for static lock servers would likely improve the results under those options significantly. For example, other tasks could be allowed to execute on the same cores and simply incur a penalty for each time the lock server may need to execute. In the future, I also plan to explore the tradeoffs of lock server configurations on four-socket systems; four local lock servers could be employed on such a platform, but doing so effectively will likely require further work on allocating tasks to clusters in order to reduce blocking.

5. REFERENCES

[1] AUTOSAR Release 4.4, Classic Platform, Specification of Operating System. `https://www.autosar.org/`, 2019.

[2] SchedCAT: Schedulability test collection and toolkit. `https://github.com/brandenburg/schedcat`, 2019. Accessed: 2019-02-07.

[3] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *PLDI '98*.

[4] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS '07*.

[5] B. Brandenburg and J. Anderson. Feather-trace: A lightweight event tracing toolkit. In *OSPERT '07*.

[6] A. Burns and A. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *ECRTS '13*.

[7] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *ECRTS '10*.

[8] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS '01*.

[9] J. Garrido, S. Zhao, A. Burns, and A. Wellings. Supporting nested resources in MrsP. In *Ada-Europe International Conference on Reliable Software Technologies '17*.

[10] C. Jarrett, B. Ward, and J. Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *RTNS '15*.

[11] J. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC'12*.

[12] C. Nemitz, T. Amert, and J. Anderson. Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts. In *ECRTS '18*.

[13] C. Nemitz, T. Amert, and J. Anderson. Real-time multiprocessor locks with nesting: optimizing the common case. *Real-Time Systems*, 55(2), 2019.

[14] H. Takada and K. Sakamura. Real-time scalability of nested spin locks. In *RTCSA '95*.

[15] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS '12*.

The temporal correlation of data in a multirate system

Evariste Ntaryamira
Inria, Paris
evariste.ntaryamira@inria.fr

Cristian Maxim
IRT System X, Saclay
cristian.maxim@irt-systemx.fr

Liliana Cucu-Grosjean
Inria, Paris
liliana.cucu@inria.fr

ABSTRACT

Technologies within embedded real-time systems are continuously evolving making them intelligent; at some point they can achieve targeted functions autonomously. Such systems are extended with capability of sensing the surrounding environment and deciding on their own. In addition to the feasible scheduling policy, the correctness of such decisions highly depends on the quality of the used input data. Thereby, the data management within such systems must fulfill some properties in order to guarantee their correct functioning. In this paper we address the problem of *data temporal correlation and validity* when the system scheduling properties are defined. We present preliminary results on expected properties of the architectures and underline future work.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

scheduling, data, real-time

1. INTRODUCTION AND RELATED WORK

The software embedded in a real-time system is composed of a large number of applications communicating through shared variables. These systems have the capability of making functional decisions autonomously. For instance, the autonomous vehicles are extended with capability of sensing the surrounding environment and navigating on their own by making driving decisions. Additionally to the traditional vehicle functions autonomous vehicles are equipped with algorithms that ease to infer the identity of the objects in the neighborhood. These algorithms work on the data propagating from different paths where these data may be resulting from a same or different source application.

In this paper we consider that the data traversing different paths result from a same source application and only the data produced during a same execution step are meant to be further associated. However, the propagation delays from the source to the associating application may differ from one path to another for different reasons. Hence, waiting queues are used to temporally store the data from the shortest paths until the corresponding data from the longest path arrive. In this paper we propose how to compute the minimum size (later referred to as optimal size) for each of the waiting queues in order to economize the memory resource utilization.

Contribution In this paper we study the temporal correlation of the data propagating along several paths taking into account the timing properties of the system scheduling. We consider the inter-task communication model (circular buffer) offering a predictable data management between communicating applications.

Paper structure. In Section 2 we present the system and the communication models with the associated notations and we formulate the correlation problem. In Section 3 we introduce our first contribution; the data reading strategy easing the correlation process is proposed in 3.1.1, a formal method computing the *Data Consistent Time Interval in 3.1.2*, the formal methods to compute the buffers optimal sizes in 3.1.3 and 3.2.2 and, finally, the proposed data correlation approach is presented in 3.2.1. We conclude and present future work in Section 4.

1.1 Motivating example

An example illustrating the need of associating correlated data is the *FADE* system [8], represented on the Figure 1. *FADE* is a vehicle detection and tracking system composed of a set of image processing components in charge of detecting the characteristics (detection of shadows, headlights, etc.) related to the presence of a vehicle in the neighborhood. For the performance optimization reasons, the image processing is done in a multi-resolution mode. Thereby, only the central part of the image is processed in the high resolution mode (for detecting vehicles being far away) and the periphery of the image is treated in the low resolution mode for detecting closer vehicles.

precisely, the high resolution image initially produced by the IAA[1] is sent to both the CDA[2] and the IPA[3]. Further, the *CDA* application reads the high resolution image as input. The latter is cropped, decimated and processed and, finally, a low resolution image is produced at the *CDA* output port. Further, the low and high resolution images are associated and fused in order to infer the identity of the targeted object.

Obviously, the associated data are shifted by a certain delay induced by the processing of the *CDA*. In order to avoid potential performance degradation, the *IPA* instances must read (from each of the buffers) the data resulting from the same execution step of the IAA [7]. In such a context we say that the used data are *temporally correlated* and, in our paper, we aim to efficiently address the *data temporal*

[1]Image Acquisition Application
[2]Crop and Decimate Application
[3]Image Processing Application

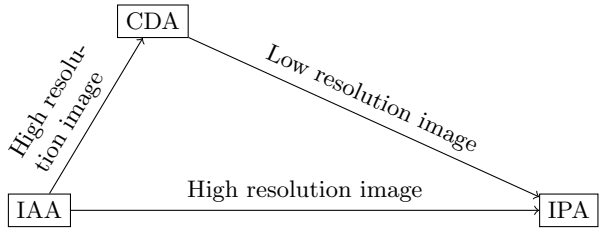


Figure 1: The vehicle detection and tracking system.

correlation problem.

1.2 Related works

The problem related to the association of the data produced during the same execution step (having the same age) has been addressed during the last decades. N. Pontisso and al. in [3, 4] proposed approaches to manage data matching in periodic systems without considering strict properties on the system scheduling; that is, earlier the implementation stage. Their work differs from ours in the sense that the correlation of the data produced during the same execution is ensured taking into account the task system scheduling properties. The FIFO waiting queues are managed sequentially (linearly) while in our work we consider the FIFO buffer to be managed circularly. The circular buffer organization is described in the Section 2.2.

Authors in [7] consider the circular buffer communication model but without considering the task system scheduling properties. In order to ensure the temporal correlation between data from different paths, the communication model is organized as follows: each data sample is double stamped with two different dates; the *timestamp* and *timeOfIssue*. The first, considered as the date of birth of the data sample, is given by the application that initially generated it and is remained unchanged.At the execution completion of each application the data a *timeOfIssue* is added. At the end of the paths, the reader application will retrieve the recent data having the same *timestamp* from the connected buffers In our work we do not time stamp the samples.

2. MODELS AND ANNOTATIONS

In this section, we introduce the system and the communication models as well as the notion of functional chains.

2.1 System Model

We consider a system τ of n periodic tasks $\{\tau_1, \tau_2, \cdots, \tau_n\}$ scheduled preemptively on one processor according to fixed-priority scheduling algorithm. Each task τ_i is described by the tuple (C_i, T_i), where C_i is the worst-case execution time and T_i is the period of the task τ_i. We assume that all tasks are released simultaneously and they have implicit deadline; that is, the deadline is equal to period.

Without any loss of generality, we consider that the tasks are ordered from the highest to the lowest priority and the larger is the task period the lower is the priority. Hence, if $i < j$, then τ_i has a higher priority than τ_j. Each task τ_i generates an infinite number of successive jobs $\tau_{i,j}|j = 1, \cdots, \infty$. We consider that tasks share data through buffers and a task may belong to two different classes: producer or consumer. The shared buffer can be accessed for writing by a single producer while one or multiple consumers can read from it. The data propagation order between tasks does not impact an execution order between those tasks. We describe the data dependencies between the tasks by a graph.

We denote by $G = (V, E)$ such graph, where V is the set of tasks $\{\tau_1, \cdots, \tau_n\}$, E the set of edges and $(\tau_i, \tau_j) \in E$ if τ_j consumes data produced by τ_i.

The graph G may contain different data propagation paths. Hence, we define $\Pi = \{pth_1, \cdots, pth_m\}$ where m is the number paths in Π. A same producer may produce data for several consumers belonging to different paths. For instance, on the Figure 2, the output data of τ_1 is utilized by τ_2 and τ_4 belonging; respectively, to pth_1 and pth_2 where $pth_1 = \{\tau_1, \tau_2, \cdots, \tau_5\}$ and $pth_2 = \{\tau_1, \tau_4, \tau_5\}$. Accordingly, τ_1 is called the *data dispatching task* or simply the *dispatcher* whereas τ_5 is called the *data associating task* or simply the *associator*.

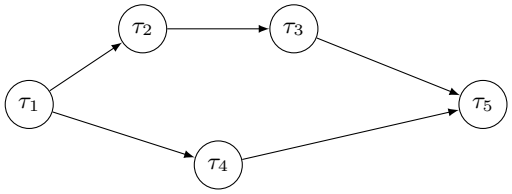


Figure 2: System tasks model

Tasks whose instances consume data produced by the *dispatcher* are referred to as *direct successors* or simply *successors*. Similarly, tasks that produce data for the *associator* are referred to as *direct predecessors* or simply *predecessors*.

2.2 Communication model

Our communication model is the circular buffer. A circular buffer is a *FIFO data structure* that considers memory to be managed circularly; that is, the *read/write* indices loop back to 0 after it reaches the buffer length [1].

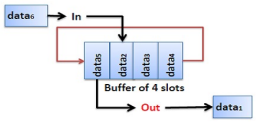


Figure 3: Example of the circular buffer

It has a fixed size allocated once at the system run-time. *tail* and *head* are the pointers indicating the reading and writing positions. Each time a new data sample is inserted into the buffer, the *head* pointer is incremented and likewise, when the data is read the *tail* pointer is incremented. *tail* and *head* are initially set to 0. The modulo operation is performed to reset head or tail to 0, every time the maximum index is reached.

The choice of a circular buffer is motivated by:

1. *The data sample already written in the buffer slot never changes the address until it is overwritten*: it avoids the data shifting from slot to the next one as it it the case for a sequential buffer. Shifting data is a resource consumption which may lead to unpredictable behavior.
2. The utilization of the circular buffer offers a high degree of the communication predictability and no dynamic memory allocation: the required size is computed offline considering the timing characteristics of the communicating tasks.
3. The circular buffer is easy to implement.

From what precedes, we consider a set of buffers $\{\beta_1, \cdots, \beta_x\}$, where each buffer β_x, in addition to *tail* and *head*, is characterized by its cardinality $|\beta_x|$; that is, its size. The size of the buffer gives the information about the number of data samples that can be stored. Each data sample is described by the tuple $\langle dID, dValue\rangle$ where dID is an integer defined as the *data identifier* and $dValue$ is the data sample value.

We consider a single buffer shared between a single producer and one or several consumers.

2.3 Correlation problem formalization

We consider a set of k data paths $\{pth_1, \cdots, pth_k\} \in \Pi$ where the data flowing along the all k paths are initially generated by a same task referred as the ***dispatcher task*** and denoted by τ_{dsp}. These paths are later associated by a same task referred to as the ***associator task*** and denoted by τ_{as}. Accordingly, we call the $\{\tau_{dsp_1}, \cdots, \tau_{dsp_k}\}$ and $\{\tau_{as_1}, \cdots, \tau_{as_k}\}$, respectively, the ***successors*** to τ_{dsp} and ***predecessors*** to τ_{as}. For a set of k paths, the data temporal correlation is required if $pth_1 \cap\ pth_2 \cap, \cdots, \cap\ pth_k = \{\tau_{dsp}, \tau_{as}\}$.

Correlation problem formalization:

$$correl(\tau_{as}) = \tau_{disp}\{pth_1(\tau_{dsp_1}, \tau_{as_1}), \cdots, pth_k(\tau_{dsp_k}, \tau_{as_k})\}$$

where k is the number of *paths* involved in the propagation of the data produced by τ_{dsp} until τ_{as}.

3. DATA CORRELATION MAINTENANCE

We consider a multirate system where the shared buffer is accessed *asynchronously in a non-blocking fashion*[4]. We decrease the uncertainty in the data management usually induced by the utilization of the arbitration mechanisms (i.e semaphores, synchronisation protocols, etc.). The later may provoke unpredictable behavior such as the priority inversion problems and possible deadlock formations [2, 5, 6].

Considering that communicating tasks may sample at different rates, we split the correlating problem into 3 subproblems, namely:

PROBLEM 1. *Ensuring that all $\{\tau_{dsp_1}, \cdots, \tau_{dsp_k}\}$ propagate the same data produced by τ_{dsp}.*

PROBLEM 2. *Setting τ_{as} to read the data resulting from the same execution step of τ_{dsp}.*

PROBLEM 3. *Computing the **optimal size** for the buffer such that Problem 1&2 have at least a solution.*

DEFINITION 1 (**Optimal size**). *A buffer size denoted by $|\beta_{dsp}|$ is optimal if it is the smallest size of β_{dsp} such that if the data read by at least one of the successors can be read by the remaining successors before being overwritten.*

3.1 Setting all successors to read the same data

We consider a set of task pairs $(\tau_{dsp}, \tau_{dsp_j}) \in E | j \in [2, k)$, where k is the number of tasks that read the data produced by τ_{dsp}. Let β_{dsp} be the buffer where τ_{dsp} instances write and the $\{\tau_{dsp_1}, \cdots, \tau_{dsp_k}\}$ read their respective inputs. Given that we are dealing with a multirate system, it is obvious that some data samples produced by τ_{dsp} may be overwritten before being read by all the k successors or can be read several times. Thus, in order to solve the Problem 1, it is required that if a given data sample is read by at least an instance of one of the τ_{dsp_j} then this sample should not be overwritten before all the $\{\tau_{dsp}\}$ consume it.

3.1.1 Data reading management

As mentioned previously, each time an instance of the producer task completes, the *head* pointer is incremented. We denote by $lp\{\tau_{dsp_j}\}$ the task of lower priority among $\{\tau_{dsp_1}, \cdots, \tau_{dsp_k}\}$ tasks that read the data produced by τ_{dsp}. In order to set all the *successors* to read the same data sample from β_{dsp} the following principle is imposed:

- **(i)**:$\forall\{\tau_{dsp_1}, \cdots, \tau_{dsp_k}\}$ the value of *tail* is set by an instance of $lp\{\tau_{dsp_j}\}$ at its completion. We assume that the $lp\{\tau_{dsp_j}\}$ instances are always the last ones to read the data produced by τ_{dsp} which is responsible of pointing*where to write next time* (*head* value). Accordingly, the value of *tail* is given by the position where an instance of τ_{dsp} previously wrote; that is, $head - 1$. Formally,

$$tail_{lp\{\tau_{dsp_j}\}} \leftarrow head - 1 \quad (1)$$

- **(ii)**: Each time an instance of any of the $\{\tau_{dsp_1}, \cdots, \tau_{dsp_k}\}$ is activated, it reads from the position pointed by *tail* pointer. We should note that the *tail* value is set by $lp\{\tau_{dsp_j}\}$ at its completion time.

$$tail_{\tau_{dsp_j}} \leftarrow tail_{lp\{\tau_{dsp_j}\}} \quad (2)$$

Further, we compute the maximum time delay that the read data can stay into the buffer before being overwritten while considering that the buffer size is optimal. We call this delay the ***Data Consistent Time Interval*** that we denote by ***DCTI*** and formally computed using the Equality 3.

3.1.2 Computing the DCTI

The Theorem 1 provides the formal way to computing the $DCTI$.

THEOREM 1. *We consider $\{\tau_{dsp_1}, \cdots, \tau_{dsp_k}\}$ where k is the number of successors to τ_{dsp}. The DCTI is found when τ_{dsp} and $lp\{\tau_{dsp_j}\}$ are released simultaneously in such a manner that the response time of the current instance of $lp\{\tau_{dsp_j}\}$ is equal to the period of τ_{dsp} and the next instance of $lp\{\tau_{dsp_j}\}$ executes for its worst case response time. Formally,*

$$DCTI = T_{lp\{\tau_{dsp_j}\}} + R_{lp\{\tau_{dsp_j}\}} - C_{dsp} \quad (3)$$

where $T_{lp\{\tau_{dsp_j}\}}$ and $R_{lp\{\tau_{dsp_j}\}}$ are, respectively, the period and the worst case response time of $lp\{\tau_{dsp_j}\}$ and C_{dsp} the worst case execution time of τ_{dsp}.

Proof: The Equality 3 is detailed as follows

$$DCTI = \underbrace{T_{dsp} - C_{dsp}}_{(a)} + \underbrace{T_{lp\{\tau_{dsp_j}\}} - T_{dsp}}_{(b)} + \underbrace{R_{lp\{\tau_{dsp_j}\}}}_{(c)}$$

where

- (a): If τ_{dsp} and $lp\{\tau_{dsp_j}\}$ were released simultaneously and the response time of $lp\{\tau_{dsp_j}\}$ is equal to T_{dsp}, it means that by the time the data produced by τ_{dsp} is tagged (at the completion of $lp\{\tau_{dsp_j}\}$ instance), the maximum time this data will have been into the buffer is given by $T_{dsp} - C_{dsp}$.

[4]Shared variable (buffer) is accessed without any arbitration mechanism such as semaphores or any kind of synchronization protocols.

- (b): Since the execution completion of the current instant of $lp\{\tau_{dsp_j}\}$ happened at a time instant equal to T_{dsp}, the release time of the next instance of $lp\{\tau_{dsp_j}\}$ is going to happen at a time instant given by $T_{lp\{\tau_{dsp_j}\}} - T_{dsp}$.
- (c): The next data to be read is going to be tagged at the execution completion of the next instance of $lp\{\tau_{dsp_j}\}$. If the latter executes for a time equal to its worst case response time, then, the time interval between the previous and the current tagged data is the largest possible computed by the Equation 3.

3.1.3 Computing the optimal size of the buffer β_{dsp}

The maximum time delay that can separate two consecutive read data is the $DCTI$. Accordingly, the formal way to compute the optimal value of $|\beta_{dsp}|$ is given by the Theorem 2.

THEOREM 2. *We consider $\{\tau_{dsp_1}, \cdots, \tau_{dsp_k}\}$ where k is the number of successors to τ_{dsp}. We denote by β_{dsp} the buffer where τ_{dsp} instances write. The optimal value of $|\beta_{dsp}|$ is equal to the number of τ_{dsp} instances that can be released and complete their execution within a time given by the $DCTI$ if $T_{dsp} < T_{lp\{\tau_{dsp_j}\}}$ or it is equal to 1 otherwise. Formally,*

$$|\beta_{dsp}| = \begin{cases} \left\lceil \frac{DCTI}{T_{dsp}} \right\rceil, & \text{if } T_{dsp} < T_{lp\{\tau_{dsp_j}\}}. \\ 1, & Otherwise \end{cases} \quad (4)$$

Proof: The $DCTI$ time defines the largest time that can separate two consecutive read data by the $\{\tau_{dsp_1}, \cdots, \tau_{dsp_k}\}$ where k is the number of *successors* to τ_{dsp}. The read data (tagged at the execution completion of an instance of the $lp\{\tau_{dsp_j}\}$) should not be overwritten before the next instance of $lp\{\tau_{dsp_j}\}$ completes.

In other words, there should be a sufficient buffer slots to keep the all data samples produced within the $DCTI$ time interval. Otherwise, the read data may be overwritten before the new data is set available to reading. Hence, the system of equations 4 is correct.

3.2 Maintaining data temporal correlation

The results in Section 3.1 guarantee that each data that propagate through different paths are read by all $\{\tau_{dsp_j}\}_{j=2}^{k}$ where k is the number of paths that propagate the data meant to be associated by τ_{as}. Moreover, different data paths may have different propagation delays for the data propagating from τ_{dsp} to τ_{as}. In the Sections 3.2.1 and 3.2.2 we propose a solution to this situation.

3.2.1 The correlating approach

When an instance of $lp\{\tau_{dsp_j}\}$ tags the data to be read, the dID of the tagged data sample is incremented. The tagged data is propagated trough different paths and all data related to it will have this same dID. So, when an instance of the *associator task* (τ_{as}) is activated it reads the data samples having the same dID from all the buffers where $\{\tau_{as_1}, \cdots, \tau_{as_k}\}$ output their computation results.

We assume that $lp\{\tau_{dsp_j}\}$ belongs to the path with the largest data propagation delay from τ_{dsp} to τ_{as}.

3.2.2 Setting the buffers sizes

We consider a set of k paths, $\{pth_1, \cdots, pth_k\}$, through which the data produced by τ_{dsp} propagate until τ_{as}. We denoted by The largest propagation delay it can take for the data to propagate from τ_{dsp} to τ_{as} is referred to the *worst case data propagation delay* that we denote by $wcdpd_{\tau_{dsp} \to \tau_{as}}$.

Additionally, we plan to compute, for each the $pth_i \in \Pi$, the smallest time delay it can take for the data to propagate from τ_{dsp} to τ_{as} and we denote it by $min\{delay\langle pth_i \rangle\}$. This calculation is presented here as a conjecture left as future work.

CONJECTURE 1. *We consider $(\tau_{as_i}, \tau_{as}) \in E | i \in \{2, \cdots, k\}$ where k is the number of paths trough which the data produced by τ_{dsp} propagate until τ_{as} and τ_{as_i} the last task belonging to pth_i path. Let β_{as_i} be the buffer where τ_{as_i} writes the outputs meant to be consumed by τ_{as}. Formally,*

$$|\beta_{as_i}| = \left\lceil \frac{wcdpd_{\tau_{dsp} \to \tau_{as}}}{min\{delay\langle pth_i \rangle\}} \right\rceil \quad (5)$$

4. CONCLUSION AND FUTURE WORKS

In this paper we have presented preliminary results on the consideration of both data propagation and the fulfilement of real-time constraints. Our future work includes the proof of our conjecture as well as the application of results on a drone use case study.

5. REFERENCES

[1] EmbedJournal. Implementing circular buffer in c. .

[2] T. Kloda, A. Bertout, and Y. Sorel. Latency analysis for data chains of real-time periodic tasks. In *23rd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, pages 360–367, 2018.

[3] N. Pontisso. *Association cohérente de données dans les systèmes temps réel à base de composants: Application aux logiciels spatiaux.* PhD thesis, Institut National Polytechnique de Toulouse, 2009.

[4] N. Pontisso, P. Quéinnec, and G. Padiou. Analysis of distributed multi-periodic systems to achieve consistent data matching. *Concurrency and Computation: Practice and Experience*, (n° 2):pp. 234–249, 2013.

[5] J. Schlatow and R. Ernst. Response-time analysis for task chains in communicating threads. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 245–254, 2016.

[6] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 1990.

[7] B. Steux. *RTMAPS, un environnement logiciel dédié à la conception d'applications embarqués tems-réel. Utilisation pour la détection automatique de véhicules par fusion radar/Vision.* PhD thesis, Ecole des mines de Paris, France, 2001.

[8] B. Steux, C. Laurgeau, L. Salesse, and D. Wautier. Fade: a vehicle detection and tracking system featuring monocular color vision and radar data fusion. In *Intelligent Vehicle Symposium, IEEE*, 2002.

Formal Verification of Real-time Networks

Lucien Rakotomalala Marc Boyer Pierre Roux
ONERA / DTIS
Université de Toulouse
F-31055 Toulouse – France
firstname.name@onera.fr

ABSTRACT

Embedded real-time networks must ensure guaranteed delays. Network calculus is a theory providing bounds on such delays. This mathematical theory currently relies on, human made, pen and paper proofs. The current work offers to formalize such proofs in Coq, an automated proof checker. We formalize a subset of the theory large enough to handle a complete proof of bounds on a representative case study.

Keywords

network-calculus, Coq, real-time network

1. INTRODUCTION

Nowadays, real-time systems are pervasive in embedded applications such as the aerospace or automotive industries. Such applications being critical, it is mandatory to establish a high degree of confidence in their functional and temporal behaviour. Whereas a lot of work is available on functional verification, this paper focuses on temporal correctness. Analysis methods in this regard do exist and they are mathematically proved. However, these proofs are only reviewed and verified by humans which implies a substantial risk of error, due to their complexity or subtle hypotheses.

Therefore, some mistakes can be made during the writing and reviewing process of a proof. A major source of mistakes is the omission of an implicit hypothesis when reusing a previous results. Such omissions have occurred several times in real-time analysis proofs. For example, it has been recently discovered that some self-suspension consideration was inexact 20 years after publication of the original paper [14]. As another example, an error in real-time analyses of the CAN bus, was discovered only 13 years after the original publication [9].

We aim at increasing the confidence in mathematical proofs by automating the proofreading process. This can be made by a computer running a proof assistant. Such tools are developed by computer scientists and mathematicians for nearly half a century. We can for instance mention Coq, Isabelle or PVS [3, 15, 16]. We can use one of them to formally define mathematical objects, enunciate theorems and finally describe proofs of these theorems. A computer is then able to automatically check these proofs.

As a first advantage to the use of a proof assistant, the confidence in the correctness of the proofs is reduced to the absence of bugs in the tool, the coherence of the implemented logic and potential axioms used. On this last point, the tool allows to know exactly which axioms are used in each proof.

Another advantage is to identify where and how hypotheses are used by a proof. In extreme cases, it happens that some hypotheses are in fact unused in the course of a proof. To check this, it is enough to remove the considered hypothesis and attempt a recompilation of the proof.

Finally, a proof assistant enables a simpler and safer reuse of the results: an application of a theorem is only possible when all hypotheses are collected. Thus, it is not possible to forget hypotheses.

The proof assistant we use in this paper is Coq [3], developed by Inria, based on a small kernel and extended by a large sets of libraries.

The kernel implements an intuitionistic logic. Our confidence is based on this reduced kernel that ultimately check all Coq proofs. This kernel uses a low level language that is simple but hardly usable by humans. Coq thus provides an interface to make it operable, it interprets user's commands to elaborate a proof in the kernel language.

Coq comes with a standard library providing mathematical models and properties. Other libraries go beyond this standard set, such as the Mathcomp [12] or the Coquelicot [4] libraries. The correctness of these libraries relies on the fact that they are checked by the kernel.

In order to formalize proofs on a specific problem, a Coq user first defines a mathematics model (the modeling of the problem). Then she expresses some properties of the model (stating lemmas or theorems) and eventually writes Coq commands to prove these properties.

During this process, Coq checks that definitions and statements of properties are well formed and that the proofs hold.

Our proofs will focus on embedded networks and will use an analysis method of temporal properties on these networks: the network calculus (NC). This theory heavily relies on tropical algebra through the dioid of *min-plus* functions.

As previously described, our first step will consist in writing NC definitions in the Coq language. Secondly, NC results found in the literature will be prove within Coq. Only a NC expert can check that the Coq definitions match with the ones in the literature. Thus our models have to be readable even without a deep Coq expertise. In contrast, the second step, proof writing, does not need to be checked by a human since the compilation guarantees the proofs.

While our final goal is to be able to verify a full industrial network, our current contributions are:

- an extension of the Coq Mathematical Components library of algebraic structures,
- formal definitions and proofs of some typical network cal-

culus theorems,

- an application on a first case study, handling 5 flows through 5 servers with FIFO policy.

Section 2 presents related works on proof assistants and real-time network analyses. Then, Section 3 presents our formal development. Section 4 is a case study on a performances analyses of a network. Finally, we conclude with Section 5 and explain our future work in Section 6.

2. RELATED WORK

Network calculus tools take as input the description of a network and compute delay bounds. The validity of these bounds relies on both the correctness of the network calculus theorems (produced by authors and checked by reviewers) and the correctness of the implementation (relying on developer skills).

Formally proving correct implementation was the aim of [11], using the proof assistant *Isabelle*. Our goal is to prove both theory and implementation correctness, using another proof assistant, Coq.

Our work is part of the project *RT-proofs* [2]. The main objective of this project is to lay the foundations for computer-assisted formal verification of timing analysis results. Many works have been performed already, for example a verification of a CAN schedulability analysis with Coq [10].

Finally, a library for the development of machine-checked schedulability analysis using Coq is also available [8]

3. NETWORK CALCULUS WITH COQ

The Network Calculus theory is based on the *min-plus* dioid [5]. Thus, our first contribution consists in adding this algebraic structure to the Mathcomp library (Sections 3.1, 3.2). We then formalize main NC results (Section 3.3). Some metrics on the Coq development are given in Section 3.4.

3.1 Algebraic structures

We use some of the existing elements in the Mathcomp library [12] to define the algebraic structure of complete dioids. The Mathcomp library provides some algebraic structures useful in our case (monoids, rings,...) but not dioids.

So, with the help of this library, we add a description of the dioid structure as defined by [13].

Definition 1 (Dioid). *A set $\mathcal{D}$ with two operators $\oplus$ and $\otimes$ is called a dioid if*

- *$\oplus$ is associative and commutative and admits a neutral element $\bar{0}$*
- *$\otimes$ is associative and admits a neutral element $\bar{1}$*
- *$\otimes$ is left and right distributive over $\oplus$*
- *$\bar{0}$ is absorbing for $\otimes$*
- *$\oplus$ is idempotent, i.e: $\forall a \in \mathcal{D}, a \oplus a = a$*

A dioid is said to be complete if it is closed for infinite sum and if the product distributes over infinite sums on both sides.[1] Under some assumptions, a subset of a complete dioid remains a complete dioid.

[1]All these algebraic structures and their properties have been developed using only intuitionistic logic. Even proofs on infinite sums don't require classical reasoning since we only prove results based on the hypothesis that such sums do exist. In contrary, the next section will deal with real numbers whose formalization in the Coq standard library requires classical logic with the excluded middle axiom.

Theorem 1. *Let $\mathcal{D}$ be a complete dioid with operators $\oplus$ and $\otimes$. Any $\mathcal{D}' \subseteq \mathcal{D}$ including $\bar{0}$ and $\bar{1}$ and stable for $\oplus$, $\otimes$ and infinite sums is also a complete dioid.*

We can then define the kleene operator, useful in NC.

Definition 2 (Kleene operator). *Let $\mathcal{D}$ be a complete dioid with operators $\oplus$ and $\otimes$, the kleene operator on $a \in \mathcal{D}$ is: $a^* = \bigoplus_{i=0}^{+\infty} a^i$ with $a^0 = \bar{1}$ and $a^{i+1} = a \otimes a^i$*

By using this definition, we can state the next theorem. This result is shared with language theory.

Theorem 2. *Let $\mathcal{D}$ be a complete dioid with $\oplus$ and $\otimes$, for all a, b in $\mathcal{D}$, $a^* \otimes b$ is the least solution of $x = (a \otimes x) \oplus b$.*

There exist many other results on dioids [5, 13]. Among them we prove 78 properties useful for NC.

All of these results have been submitted as a pull request on the Mathcomp library[2].

3.2 Instances

To use these properties in the NC context, we have to prove that sets of interest are dioids. This implies to:

- give a set $\mathcal{D}$,
- give operators $\oplus$, $\otimes$ and their neutral elements,
- prove that dioid properties (cf. Definition 1: associativity, commutativity...) hold.

NC handles functions on real values: $\mathcal{F} : \mathbb{R}^+ \to \overline{\mathbb{R}}$, with $\overline{\mathbb{R}} = \mathbb{R} \bigcup \{-\infty, +\infty\}$, and uses the following two operators:

- minimum: $(f \wedge g)(t) = \min(f(t), g(t))$
- convolution: $(f * g)(t) = \inf_{0 \leqslant s \leqslant t} \{f(t-s) + g(s)\}$

Theorem 3. *The set of functions $\mathcal{F}$ with $\oplus = \wedge$ and $\otimes = *$ is a complete dioid.*

Depending on the authors and even on the papers, NC results handle either $\mathcal{F}$ or some specific subsets:

- $\mathcal{F}^+ : \mathbb{R}^+ \to \overline{\mathbb{R}}^+$,
- $\mathcal{F}^\uparrow$: subset of non-decreasing elements of $\mathcal{F}^+$.

Theorem 4. *The sets $\mathcal{F}^+$ and $\mathcal{F}^\uparrow$ with $\oplus = \wedge$ and $\otimes = *$ are complete dioids.*

This is proved by using Theorem 1. One contribution of our work is to explicit which subset is needed for each result.

To develop these constructions, we use results of the Coquelicot library [4]: the set $\overline{\mathbb{R}}$ and its properties.

3.3 Network calculus

3.3.1 Model

NC models data flows by the cumulative amount of data at a point in a network at time t.

Definition 3 (Cumulative function). *A function $f : \mathbb{R}^+ \to \mathbb{R}^+$ is a cumulative function if f:*

- *is non-decreasing: $\forall t, d \in \mathbb{R}^+, f(t) \leqslant f(t+d)$,*
- *starts at 0: $f(0) = 0$,*
- *is left-continuous.*

[2]`https://github.com/math-comp/math-comp/pull/357`

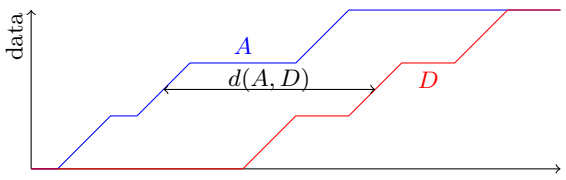


Figure 1: Illustration of the notion of delay

The set of cumulative functions is denoted $\mathcal{C}$. They are non-decreasing because they represent a cumulative amount of data and this one can not decrease. We consider that flows do not contain data before $t = 0$, so $\mathcal{C}$ functions start in 0.

A server is a relation between two cumulative functions: A for arrival and D for departure of the server.

Definition 4 (Server). *A server $S \subseteq \mathcal{C} \times \mathcal{C}$ satisfies:*

- $\forall A \in \mathcal{C},\ \exists D \in \mathcal{C},\ (A, D) \in S$
- $(A, D) \in S \Rightarrow D \leqslant A$

The first property means that, for all arrival there exists a departure. The second property means that departure can not happen before arrival so $D \leqslant A$.

Another notion of NC we use is called arrival curve. It is used to constrain cumulative functions.

Definition 5 (Arrival curve). *Let $A \in \mathcal{C}$ be a cumulative function. The function $\alpha \in \mathcal{F}^{\uparrow}$ is an arrival curve for A if $A \leqslant A * \alpha$.*

To specify servers, NC uses the notion of minimal service. We define it below, first using mathematics, then in Coq.

Definition 6 (minimal service). *Let S be a server and $\beta \in \mathcal{F}^{+}$. The server S is said to offer a* minimal service curve *β if: $\forall (A, D) \in S \implies A * \beta \leqslant D$.*

```
Definition is_min_service (S : C → C → Prop) (beta : Fplus)
           := forall A D, S A D → A ∗ beta ⩽ D.
```

The notation `Fplus` represents the set $\mathcal{F}^{+}$ presented in Section 3.2. (`S : C → C → Prop`) means that S is a relation on $\mathcal{C}$. The term `S A D` signifies $(A, D) \in S$ and the arrow $\rightarrow$ stands for logical implication in Coq.

3.3.2 Properties

To analyze temporal network performances, NC defines a notions of delay. The delay experienced by the flow whose arrival is A and departure is D is denoted $d(A, D)$, illustrated in figure 1. A formal definition can be found in [5].

Different policies for servers are defined in NC. One of them is the First-In First-Out policy. In such a server, each packet is served after all previously arrived packets have been served. For this policy, a NC theorem bounds the delay experienced by each packet. We proved this theorem in Coq.

Finally, NC provides a method to compute a contract on the output of a server,i.e., an arrival for the next server.

3.4 Coq development overview

Table 1 gives some metrics on our Coq development. It can first be observed that the number of definitions in *Instances* is higher compared to *Dioid* and *NC*. This difference has no significant meaning: it comes from a difference in Coq programming style between the two libraries.

	Definitions	Properties	Lines	Lines/prop.
Dioid	19	78	1616	21
Instances	108	159	2888	18
NC	26	52	2253	43

Table 1: Summary Table of Coq definitions and property done

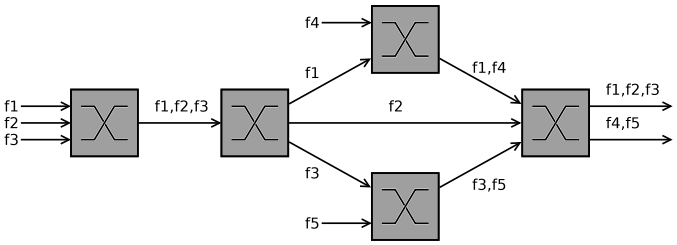


Figure 2: Network Topology

More interestingly, the ratio between the number of lines and the number of properties in *Dioid* and *Instances* is lower than in *NC*. This means that *Dioid* and *Instances* contain properties which only require short proofs, never exceeding 10 lines and most of time taking only one. On the contrary, *NC* requires larger, more complex, proofs.

These definitions and properties are very much inspired from a NC textbook [5], giving pen and paper proofs of NC properties. It is thus possible to compare pen and paper to Coq versions of these proofs. Formalizing pend and paper proofs in Coq provides several benefits. As expected, some typos have been found but we also found a few mistakes in proofs, which we had to fix[3]. More interestingly, some results appeared to be over specified: Coq helped us to simplify the hypotheses. Lastly, some properties have been both simplified and generalized: pen and paper proofs valid only for specific dioid instances have been leveraged to any dioid, with a shorter proof.

4. CASE STUDY

In this section, we work on a simple network with a particular topology shown on Figure 2. In this network, we consider that data transmissions are periodic with a 20 Mbits per second rate. The frame sizes are fixed to 1 Kbyte The scheduling policy for each server is FIFO, as introduced in subsection 3.3.2. The speed rate of each server is fixed to 100 Mbits per second. Servers are assumed to have no latency. Flows 1, 2 and 3 converge on the leftmost switch and share its output port. The next switch separate them: flow 1 goes up and competes with flow 4 on the output port of the upper switch. Flow 3 symmetrically goes down and meets the flow 5. All flows then converge to the rightmost switch, flows 1, 2 and 3 sharing one output port and flows 4 and 5 the other.

The objective is, for each flow, to bound the delay when crossing the entire network. To do so, we have written a Coq proof which consists in two steps. A first step considers each server and its crossing flows individually, and applies the Coq results presented in the previous sections. The second step consists in combining local results with respect to

[3]For example, in proof of theorem 6.2, a confusion was made between $\geqslant$ and $>$, invalidating the proof.

Flow	1	2	3	4	5
Delays bound (μs)	601.6	368	601.6	233.6	233.6

Table 2: Delay bound for each flow

the topology presented in Figure 2. This leads to algebraic expressions of the delays (in the *min-plus* dioid) whose numerical values are computed using the *min-plus* calculator from RTaW [1]. This tool implements the algorithms from [6] whose pen and paper proofs have not been formalized in Coq yet. The results are presented in Table 2.

5. CONCLUSION

The aim of this work was to formalize (using Coq) results on delay bounds of real-time network (using the NC theory).

This required the formalization in Coq of the algebraic structure of complete dioids. We rely on the Mathcomp library and we shared our development to this library. Then, we built specific instances of complete dioids used in NC with the help from the Coquelicot library.

Last, we developed a set of NC definitions and results, sufficient to perform the complete proof of a first case study.

Thus, we obtained a Coq development of 6757 lines containing a definition of the algebraic structure of dioids, instances of dioids and NC results. This work took one year, considering that the main author was a newcomer to both Coq and the NC theory.

Several benefits come with this formal development: we found a few mistakes in proofs from [5], which we had to fix. More interestingly, some results appeared to be over specified: Coq helped us to reduce the hypotheses. Last, some results have been generalized while simplifying their proofs.

Finally, the results are applied to a first case study. We used here a tool from RTaW to compute the final numerical results but Coq is used to prove the correctness of the computed expressions and all properties used to obtain these expressions.

We notice that there are three possible kinds of modifications of our case study. First, a modification of its numerical values (throughput, packet sizes...) does not change the Coq proof, since only numerical parameters of the final computation are affected. Second, a modification of the service policy requires to prove new theorems related to the new policy, but does not change the global structure of the proof. Finally, a modification in the network topology or routing breaks the structure of the proof.

6. FUTURE WORK

One may wonder how the work done for this small case study is relevant for realistic configurations.

Verification of actual embedded network, like AFDX [7] requires only two more results: on static priority scheduling and packetisation. We plan to add such Coq proofs.

In our case study, we use an external tool to compute the value of analytic expressions. We plan to either have Coq compute them by itself or verify the values computed by the external tool. This will allow us to have a complete Coq validation of performances bounds values.

The change of routing implies a manual modification of the proof. However, the structure of the proof is very repetitive, and quite a direct mapping of the network topology. We plan to automatize this part: either inside Coq, using dedicated tactics, or collaborating with an external tool, as done in [10].

7. REFERENCES

[1] Real time at work. `http://realtimeatwork.com/`.

[2] RT-proofs main page. `https://rt-proofs.inria.fr/`.

[3] Y. Bertot and P. Castéran. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions*. 01 2004.

[4] S. Boldo, C. Lelay, and G. Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Math. in Computer Science*, 9(1):41–62, Mar 2015.

[5] A. Bouillard, M. Boyer, and E. Le Corronc. *Deterministic Network Calculus: From Theory to Practical Implementation*. 10 2018.

[6] A. Bouillard and E. Thierry. An algorithmic toolbox for network calculus. *Discrete Event Dynamic Systems: Theory and Applications*, 18, 03 2008.

[7] M. Boyer, N. Navet, and M. Fumey. Experimental assessment of timing verification techniques for AFDX. In *6th European Congress on Embedded Real Time Software and Systems*, Toulouse, France, Feb. 2012.

[8] F. Cerqueira, F. Stutz, and B. B. Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 273–284, 2016.

[9] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, Apr 2007.

[10] P. Fradet, X. Guo, J.-F. Monin, and S. Quinton. CertiCAN: A Tool for the Coq Certification of CAN Analysis Results. In *RTAS 2019 - 25th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 1–10, Montreal, Canada, Apr. 2019.

[11] E. Mabille, M. Boyer, L. Fejoz, and S. Merz. Towards certifying network calculus. In *Interactive Theorem Proving, Rennes, France, July 22-26*, 2013.

[12] A. Mahboubi and E. Tassi. *Mathematical Components*. 2018.

[13] M. Minoux and M. Gondran. *Graphs, Dioids and Semirings. New Models and Algorithms*, volume 41 of *Operations Research/Computer Science Interfaces Series*. Springer, 2008.

[14] G. Nelissen, J. Fonseca, G. Raravi, and V. Nélis. Timing analysis of fixed priority self-suspending sporadic tasks. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 80–89, July 2015.

[15] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[16] M. Wenzel, L. C. Paulson, and T. Nipkow. The isabelle framework. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 33–38, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

Interdependent Multi-version Scheduling in Heterogeneous Energy-aware Embedded Systems

Julius Roeder
University of Amsterdam
Amsterdam, Netherlands
Email: j.roeder@uva.nl

Benjamin Rouxel
University of Amsterdam
Amsterdam, Netherlands
Email: benjamin.rouxel@uva.nl

Sebastian Altmeyer
University of Augsburg
Augsburg, Germany
Email: sebastian.altmeyer@informatik.uni-augsburg.de

Clemens Grelck
University of Amsterdam
Amsterdam, Netherlands
Email: c.grelck@uva.nl

***Abstract*—High-performance heterogeneous multi-core embedded systems are increasingly popular in various fields. Embedded systems engineers need to reason about more than just functional correctness of applications; they also need to reason about energy, time and security (ETS). In this paper, we sketch our coordination language and scheduling approach to enable ETS-aware applications. We present an Integer Linear Programming (ILP) based scheduler on a real life drone application, that minimizes energy consumption, guarantees timing and offers security.**

***Index Terms*—energy-aware scheduling, heterogeneous multi-core, real-time scheduling, coordination.**

I. Introduction

Increasing demand for processor performance, low power consumption and reducing heat dissipation, has lead to a surge in demand for high-performance multi-core embedded systems [1]–[4]. Powerful embedded systems, such as the Odroid-XU4 [5] and Nvidia Jetson [6], are multi-/many-core heterogeneous systems; however, with great performance comes great energy consumption (at least relative to traditional embedded systems). Furthermore, not only performance and energy consumption are important, but also security as decisions made by autonomous systems can save or cost lives. Thus, an engineer needs to take all three, time, energy and security (ETS), into account.

To tackle these challenges and enable end users to reason about ETS-characteristics of applications, we borrow from and combine two so far disjoint established research fields, namely *coordination* and *real-time scheduling* [7], they have so far addressed our targeted problem from very different angles and with very different motivations and intentions.

An application organized according to the coordination paradigm consists of a collection of interacting, independent, identifiable black-box components, i.e. the coordination language is used to describe the task graph of the given application. Components, also known as actors or tasks[1], i.e. represent application features, sequential building blocks of application, implemented in a general purpose programming language. Each component has defined functional properties, communication and interaction with other parts of the same application. Hence, coordination describes the flow of data through the different parts of an application [8]. Coordination is a well established computing paradigm with a plethora of languages, abstractions and approaches; for a survey see [9].

[1]Components and tasks will be used hereafter interchangeably.

An application described in a coordination language can then be mapped and scheduled using techniques from the real-time scheduling domain. The literature on real-time scheduling algorithms for multi-core architectures is vast, with many properties (e.g. type of scheduling algorithm, task model) and are classified in three main categories: partitioned, global and hybrid [1]. Similarly to [10], in this paper we propose an Integer-Linear-Programming (ILP) based approach to produce schedules. According to the taxonomy proposed by Davis and Burns [1] our approach can be classified as static, partitioned, time-triggered and non-preemptive.

Different approaches to minimizing energy consumption of an application on heterogeneous multi-core hard-real time systems are surveyed in [11]. Our ILP based scheduler addresses energy and security as equally important as time. The ILP scheduler can be used to generate schedules for multi-version concurrent applications, executing on heterogeneous platforms (e.g. Odroid-XU4).

In Section II, we briefly describe the architecture model, task model and coordination approach. In Section II-B, we illustrate the coordination approach with a use-case from the area of unmanned aerial vehicles. In Section III, we describe our current ILP formulation and our approach towards deriving static schedules. In Section III-B, we show the viability of our ILP scheduler on the use-case, while targeting the Odroid-XU4.

II. Architecture Model, Task Model and Coordination Approach

We consider a heterogeneous multi-core architecture (Odroid-XU4 [5]) based on the ARM big-LITTLE architecture [12]. This architecture consists of a heterogeneous CPU, with four energy efficient cores (i.e. LITTLE cores) and four high performance, deep pipeline cores (i.e. big cores). Additionally, the architecture contains a Mali GPU [5]. In such a heterogeneous platform, multi-version components can be scheduled on different computational units, e.g. a component implemented with OpenCL [13] can be scheduled on the GPU or the CPU.

In this work, we consider Directed Acyclic task Graphs (DAG). In a graph $G = (V, E)$, the set of nodes/vertices V represents the components and the set of edges E represents data dependencies between components. An edge between two components is present when they depend on each other, i.e. the source component needs to be completed before the sink

can be executed. The task graph is expressed with the help of a coordination language.

A. Coordination

A coordination language is independent from the actual code, but it guides the scheduler on how this code should be executed. An example is the coordination language S-Net [8]. However, like other coordination approaches S-Net merely addresses functional aspects of coordination programming and does not include non-functional requirements, i.e. time and security. Our coordination language [14], implemented as a Domain Specific Language (DSL), allows to incorporate the ETS-characteristics of the application.

Given the focus of the safety-critical embedded systems domain, we exclusively work with the system-level programming language *C*. Hence, a component is technically a callable *C* function with certain restrictions on its functional behaviour, together with a set of non-functional properties, i.e. timing, energy and security. Therefore, each component has specific ETS-characteristics that are crucial to determine the best mapping and schedule. Additionally, each component can have multiple implementations with equivalent functional requirements, but different ETS-characteristics. Thus, a component can have the previously mentioned multiple versions.

Our DSL coordination language allows users to specify component-specific and system-wide ETS-constraints, expressed as: 1) deadlines on the response-time of a component, 2) energy budgets of the entire system and 3) minimum security levels of each component.

B. Coordination illustrated by drone use case

Figure 1 shows the graph representation of an application that will be executed on a drone. Due to space limitation, we will not present this use-case with our coordination language. The application is currently under development at the University of Southern Denmark (SDU) and Sky-Watch, an industrial partner [15]. The application records and analyses a video on the fly and is executed on the aforementioned Odroid-XU4.

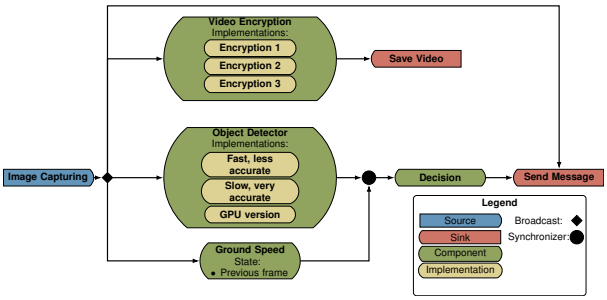


Fig. 1. Drone application coordination model

The application is organized as seen in Figure 1:

- **Image Capturing:** A frame is streamed to the computer.
- **Broadcast:** A frame is broadcasted to three components.
- **Video Encryption:** The scheduler selects the encryption level based on the ETS-characteristics of each encryption level.
- **Video Encryption & Save Video:** The frame is encrypted and saved to disk.
- **Object Detection:** The frame is analysed by the Darknet Neural Network, Tiny Darknet Neural Network [16] or OpenCV [17] , depending on the ETS-characteristics of each.
- **Ground Speed:** Computes the ground speed of the drone.
- **Synchronizer:** The results of the previous components (Object Detection & Group Speed) are synchronized and sent to the final Decision component.
- **Decision & Send Message:** A decision is made and sent to the ground station.

The two neural networks (Darknet and Tiny Darknet [16]) differ in their complexity. The Tiny Darknet neural network is much smaller and therefore can run inference approximately 70 times faster on the Odroid-XU4, at the price of reduced accuracy. Thus, scheduling one or the other version of the *Object Detection* component can have a large impact on the run time.

We aim at building a complete toolchain and workflow to compile a coordinated application to a final executable as presented by Figure 2.

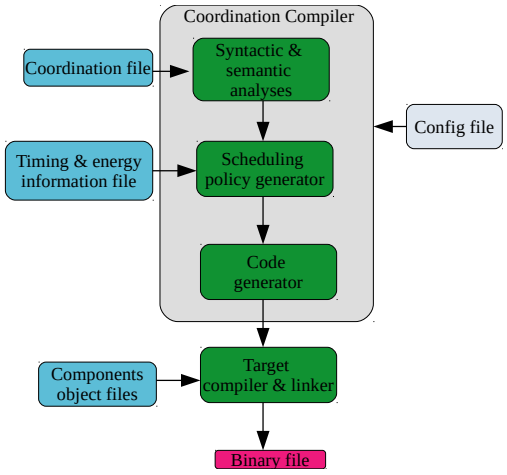


Fig. 2. Coordination workflow

A major part of our approach is to improve scheduling policies, represented by the *Scheduling policy generator* in Figure 2. Hence, in the following section we introduce our first version of a multi-version ILP based scheduler. A more thorough description of the rest of the toolchain and coordination language can be found in [14].

III. Scheduling

Once the different components are identified and defined via the coordination technique, we need to schedule the different components, i.e. assign components to different cores or co-processors (spatial mapping) and in a given order (temporal mapping). On top of different versions of the same components, the ETS-characteristics also differ depending on the exact execution unit (big core vs. LITTLE core vs. GPU) of a heterogeneous system. This increases the state space of the scheduling as we need to schedule the different components

TABLE I
ILP VARIABLES AND CONSTANTS

Function	predecessors(p)	retrieves all predecessors of p, where $p \in \tau$
Sets	B	Set of cores
	τ	Set of all components
	v_p	All versions of component p, where $p \in \tau$
	O	Set of sinks
Constants	D^C	Deadline of component set
	D^S	Minimum security level
	$C_{p,m}$	Run time of component p on core m, where $p \in \tau$, $m \in B$
	$E_{p,m}$	Energy consumption of component p on core m, where $p \in \tau$, $m \in B$
	$S_{p,m}$	Security level of component p on core m, where $p \in \tau$, $m \in B$
	$F_{p,i}$	Run time of version i of component p, where $i \in v_p$, $p \in \tau$
	$G_{p,i}$	Energy consumption of version i of component p, where $i \in v_p$, $p \in \tau$
	$H_{p,i}$	Security of version i of component p, where $i \in v_p$, $p \in \tau$
	M	Sum of run time of all components on all cores
Integer Variables	D^E	Energy constraint of component set
	ρ_p	Start time of component p, where $p \in \tau$
Binary Variables	$w_{p,m}$	component p mapped to core m, where $p \in \tau$, $m \in B$
	$a_{p,i}$	version i of components p is selected, where $i \in v_p$, $p \in \tau$
	$same_{p,q}$	components p and q are scheduled on the same core, where $p, q \in \tau$
	$order_{p,q}$	Same core variable order p to q, where $p, q \in \tau$

depending on the overall application ETS-constraints and the ETS-characteristics of each implementation of each component.

As a starting point for the space-time scheduling we used Integer Linear Programming (ILP) to generate static schedules for the aforementioned drone use-case. ILP refers to a class of constrained optimization problems and is used to tackle problems in scheduling. All variables in an ILP problem are integers, constrained by linear inequalities. The objective function is a linear function of the variables, that needs to be either minimized or maximized.

A. ILP Formulation

The set of integer variables needed for the scheduler can be found in Table I.

Objective function The goal is to minimize the energy consumption D^E over all components of the application eq. (1):

$$\textit{minimize } D^E = \sum_{p \in \tau} \sum_{m \in B} \left(E_{p,m} \times w_{p,m} \right) \tag{1}$$

where, $E_{p,m}$ is the energy consumption of component p on core m and $w_{p,m}$ is a binary variable indicating if component p is executed on core m.

Time and Security constraints Besides energy consumption, we also need to constrain the time and security aspects of the application. Thus, eq. (2) guarantees that the sinks of the application are finalized before the deadline D^C. The **finalization time** of every sink $o \in O$ is the sum of the start time ρ_o and the worst-case execution time $C_{o,m}$ of sink o executed on core m. Equation (3) ensures that every executed component p has a security level $S_{p,m}$ equal or above the minimum security level D^S.

$$\sum_{m \in B} \left(\rho_o + C_{o,m} \times w_{o,m} \right) \leqslant D^C, \forall o \in O \tag{2}$$

$$\sum_{m \in B} S_{p,m} \times w_{p,m} \geqslant D^S, \forall p \in \tau \tag{3}$$

Mapping one component to one core Equation (4) ensures that component p is mapped on one and only one processor.

$$\sum_{m \in B} w_{p,m} = 1, \forall p \in \tau \tag{4}$$

Single version constraint Equation (5) enforces that exactly one and only one version of a component is selected, represented by $a_{p,i} = 1$.

$$\sum_{i \in v_p} a_{p,i} = 1, \forall p \in \tau \tag{5}$$

Determining ETS-characteristics Equations (6) to (8) impose that the energy, time and security of a component ($E_{p,m}$, $C_{p,m}$, $S_{p,m}$) are equal to the respective ETS-characteristics ($G_{p,i}$,$F_{p,i}$, $H_{p,i}$) of the version that is selected when $a_{p,i} = 1$.

$$E_{p,m} = \sum_{i \in v_p} (a_{p,i} \times G_{p,i}), \forall p \in \tau, \forall m \in B \tag{6}$$

$$C_{p,m} = \sum_{i \in v_p} (a_{p,i} \times F_{p,i}), \forall p \in \tau, \forall m \in B \tag{7}$$

$$S_{p,m} = \sum_{i \in v_p} (a_{p,i} \times H_{p,i}), \forall p \in \tau, \forall m \in B \tag{8}$$

Prevent overlapping on same core To prevent the overlapping of components on the same core we introduce three additional constraints. Equation (9) calculates if two components are assigned to the same core ($same_{p,q}$). Instead of a *logical or*, a summation over B is sufficient, due to the uniqueness of $w_{p,m}$. The *logical and* ($\wedge$) can be linearized, see [18] for details. Equation (10) determines the order of tasks p, q, which can be p then q ($order_{p,q}$) or q then p ($order_{q,p}$). Equation (11) prevents two tasks running on the same core to execute at the same time and enforces the correct task order. The start time of p (ρ_p) has to be larger than the end time of q ($\rho_q + C_{q,m}$).

$$same_{p,q} = \sum_{m \in B} w_{p,m} \wedge w_{q,m}, \forall (p,q) \in (\tau \times \tau), p \neq q \tag{9}$$

$$same_{p,q} = order_{p,q} + order_{q,p}, \forall (p,q) \in (\tau \times \tau), p \neq q \tag{10}$$

$$\rho_p + (1 - order_{q,p}) \times M \geqslant \rho_q + (C_{q,m} \times w_{q,m}), \quad \forall (p,q) \in (\tau \times \tau) \forall m \in B, p \neq q \tag{11}$$

The constraint eq. (11) must only be activated when two tasks are mapped on the same core ($order_{q,p} = 1$). Hence, a nullification with a big-M notation is applied [19]. M, defined in eq. (12), is the sum of the maximum run time of all components p. Thus, due to the large value of M the left hand side of eq. (11) is always larger than the right hand side, if $order_{q,p} = 0$.

$$M = \sum_{p \in \tau} max(C_{p,m}) \tag{12}$$

Data dependencies in task graph In order to comply with the data dependencies in the task graph we introduce the

following constraint eq. (13). It ensures, that if one component p depends on the data of another component q, the start time of p (ρ_p) is larger than the end time of q ($\rho_q + C_{q,m}$).

$$\rho_p \geqslant \rho_q + \sum_{m \in B} (C_{q,m} \times w_{q,m}), \forall p \in \tau, \forall q \in \textit{predecessors(p)} \tag{13}$$

B. Example Use Case

We demonstrate the generation of a static schedule of the drone use case using the ILP formulation presented in Section III-A. Not all component implementations of the algorithm are fully functional at the current stage of the project. We rely on values measured at development time, to demonstrate the scheduling and coordination techniques. Once full implementations and corresponding ETS properties will be known, we will replace these hypothetical values with actual ones. The described ETS-characteristics of a component differ per computational unit, thus the heterogeneity of the Odroid-XU4 can be taken into account.

Depending on parameters (i.e. energy budget, deadline, minimum security level), the ILP identifies different schedules. One schedule resulting from the ILP formulation is shown in Figure 3. Components are executed in the right order, according to dependencies, on different computational units without overlap.

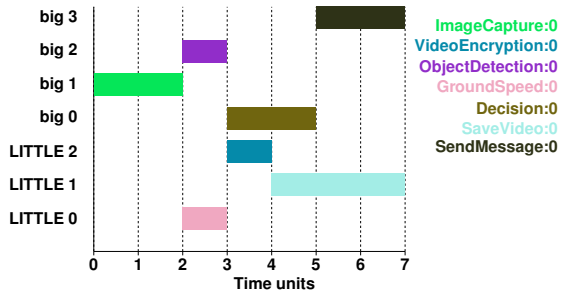


Fig. 3. Possible schedule for the drone use case showing which version of a component is scheduled and mapped on which computational unit.

As opposed to Figure 3, we increased the general security level to generate the second schedule presented in Figure 4. A higher security level lead the scheduler to pick different component versions, e.g. video encryption, and increased the energy consumption by 2 units. Once again the schedule respects component dependencies, while picking different versions and computational units due to requirement changes.

IV. Conclusion

In this paper, we briefly introduced the coordination paradigm to describe an application. We also presented an ILP formulation that allows us to generate different static schedules depending on ETS-characteristics and constraints. We further demonstrate its application on a realistic use case.

Furthermore, to enable the use of our coordination and scheduling layer for larger problems we will explore scheduling heuristics. Additionally, when all component implementations will be functional we will test the impact of our scheduling policy on real hardware and measure the impact on run time and energy consumption.

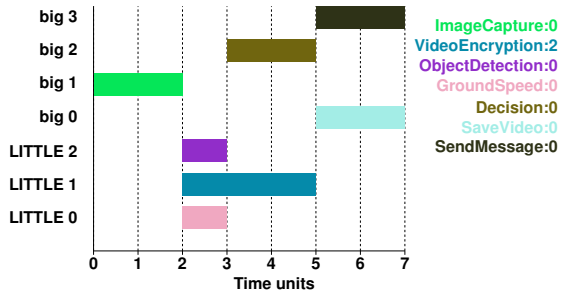


Fig. 4. Alternative schedule for the drone use case with higher security requirements.

Acknowledgment

The project has received funding from the European Unions Horizon2020 research and innovation programme under grant agreement No 779882.

References

[1] R. Davis and A. Burns, "A survey of hard real-time scheduling algorithms for multiprocessor systems," *in ACM Computing Surveys*, 2011.

[2] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte, "Contention-free execution of automotive applications on a clustered many-core platform," in *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*, pp. 14–24, IEEE, 2016.

[3] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer, "Response time analysis of synchronous data flow programs on a many-core processor," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pp. 67–76, ACM, 2016.

[4] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut, "Hiding communication delays in contention-free execution for spm-based multi-core architectures," in *31th Euromicro Conference on Real-Time Systems (ECRTS19)*, 2019.

[5] "Odroid-xu4." https://wiki.odroid.com/odroid-xu4/odroid-xu4. Accessed: 2019-09-06.

[6] "Nvidia jetson." https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/. Accessed: 2019-09-06.

[7] "Teamplay public deliverable 7.5: Achievements in the technical work packages m9." https://gitlab.inria.fr/TeamPlay_Public/TeamPlay_Public_Deliverables/blob/master/D7.5.pdf, 2018. Accessed: 2019-09-05.

[8] C. Grelck, S.-B. Scholz, and A. Shafarenko, "Asynchronous stream processing with s-net," *International Journal of Parallel Programming*, vol. 38, no. 1, pp. 38–67, 2010.

[9] G. Ciatto, S. Mariani, M. Louvel, A. Omicini, and F. Zambonelli, "Twenty years of coordination technologies: State-of-the-art and perspectives," in *International Conference on Coordination Languages and Models*, pp. 51–80, Springer, 2018.

[10] B. Rouxel, S. Derrien, and I. Puaut, "Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures," *ACM Trans. Embed. Comput. Syst*, vol. 16, no. 20, pp. 1–20, 2017.

[11] S. Z. Sheikh and M. A. Pasha, "Energy-efficient multicore scheduling for hard real-time systems: A survey," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 6, p. 94, 2018.

[12] ARM Ltd., "White Paper: big. LITTLE Technology : The Future of Mobile," p. 12, 2013.

[13] "Opencl." https://www.khronos.org/opencl/. Accessed: 2019-09-09.

[14] J. Roeder, B. Rouxel, and C. Grelck, "Towards time-, energy- and security-aware functional coordination," in *IFL 2019 - 31st Symposium on Implementation and Application of Functional Languages*, 2019.

[15] EU H2020, "TeamPlay Project," 2018. https://teamplay-h2020.eu/.

[16] J. Redmon, "Darknet: Open source neural networks in c." http://pjreddie.com/darknet/, 2013–2016.

[17] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[18] G. G. Brown and R. F. Dell, "Formulating integer linear programs: A rogues' gallery," *INFORMS Transactions on Education*, vol. 7, no. 2, pp. 153–159, 2007.

[19] I. Griva, S. G. Nash, and A. Sofer, *Linear and nonlinear optimization*, vol. 108. Siam, 2009.

PRUDA: An API for Time and Space Predictible Programming in NVDIA GPUs using CUDA

Reyyan Tekin, Houssam-Eddine ZAHAF, Giuseppe Lipari
Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL, Lille, France
{firstname.familyname}@univ-lille.fr

ABSTRACT

Recent computing platforms combine CPUs with different types of accelerators such as Graphical Processing Units (*GPUs*) to cope with the increasing computation power needed by complex real-time applications. NVIDIA GPUs are compound of hundreds of computing elements called *CUDA cores*, to achieve fast computations for parallel applications.

However, GPUs are not designed to support real-time execution, as their main goal is to achieve maximum throughput for their resources. Supporting real-time execution on NVIDIA GPUs involves not only achieving timely predictable calculations but also to optimize the CUDA cores usage.

In this work, we present the design and the implementation of *PRUDA* (Predictable Real-time CUDA), a programming platform to manage the GPU resources, therefore decide when and where a real-time task is executed. PRUDA is written in C and provides different mechanisms to manage the task priorities and allocation on the GPU. It provides tools to help a designer to properly implement real-time schedulers on the top of CUDA.

1. INTRODUCTION

Many real-time applications such as computer vision, surveillance systems, etc. demand complex processing on a large amount of data. Classical multiprocessor platforms combining only CPUs are not able to satisfy the real-time requirements of such systems as they require computing capabilities in the order of teraflops.

Recently, NVIDIA have provided computing platforms combining CPUs with different types of specialized computing unit such as GPUs, Deep Learning Accelerating (DLA), etc., on the same chip. These platforms can offer suitable solutions to meet deadlines for emerging complex real-time applications. However, the complexity of the software, combined with the complexity of the hardware architecture, makes it difficult to analyse the temporal behavior of such systems. Moreover, these accelerators are not fundamentally designed to execute real-time tasks. Therefore, they do not provide proper hardware and software mechanisms to schedule real-time tasks.

Several works [1, 3, 5] have attacked the problem of providing support to real-time systems onto GPUs from different perspectives. Kato et al. have proposed platforms (TimeGraph and RGEM) for non-preemptive scheduling for graphical tasks in GPU [5], [4]. Authors in [1] tried to study how a GPU takes scheduling decisions based on benchmarking of the Jetson TX2 platform. Capodieci et al. in [2] modified the proprietary NVIDIA driver to implement an event-driven scheduler allowing to use fine grain preemption levels provided by recent GPUs under different policies such as EDF and fixed priority. GPUSync [3] is a platform able to control scheduling within the GPU using locks. The work in [2] has closed sources whereas GPUSync [3] platform does not provide tools to freely implement real-time schedulers. In both works, GPU is used as a single core platform.

Contributions..

In this work, we develop *PRUDA*, a platform that implements different strategies to control real-time execution within a GPU using CUDA. PRUDA provides a control over priorities and task allocation and parallel execution within the same GPU at the same time. The different primitives of PRUDA allows implementing several real-time scheduling policies using different strategies. The platform is currently under active development: we are working on implementing special verison of EDF (GRUB) and Fixed priority scheduling policies with PRUDA.

The remainder of this paper is structured as follows. GPU architecture and its *known* scheduling mechanisms are details in Section 2. Section 3 presents the task and architecture models. We reserve Section 4 to define how priorities and allocations are controlled within a GPU using our platform. In Section 5, we overview the implementation of real-time schedulers using PRUDA. We draw our conclusions in Section 7.

2. GPU PROGRAMMING AND PRUDA PRIMITIVES

A GPU is compound of one or more *streaming multiprocessors* (SMs) and one or more *copy engines* (CEs). Streaming multiprocessors are able to achieve computations (*kernels*), whereas copy engines execute memory copy operations between different memory spaces. Programming the GPU requires dividing parallel computations into several grids, and each grid to several blocks. A block is a set of multiple threads. A GPU can be programmed using generic platforms such OpenCL or proprietary independent APIs. We use CUDA, a NVIDIA proprietary platform, to have a tight control on SMs and CEs in the C programming language and using the NVIDIA compiler.

When a kernel is invoked by CPU code, it submits commands to the GPU. How and when commands are executed, is hidden by constructors for intellectual property concerns. Authors in [1] have tried to reveal some *GPU scheduling secrets* by benchmarking a Jetson TX2 (abbreviated TX2 in the rest of this paper). It is compound of 6 ARM-based CPU

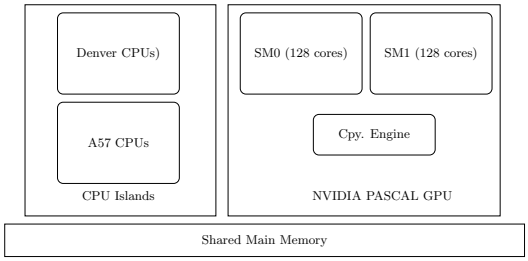


Figure 1: Jetson TX2 Architecture

cores, along with an integrated NVIDIA PASCAL-based GPU as shown in Figure 1, all running onto Ubuntu. The GPU in the TX2 is compound of 256 Cuda cores, divided into two SMs and one copy engine. CPUs and GPU share the same memory module. From a programming perspective, one may either allocate two separate memory spaces for CPU and GPU using malloc and CudaMalloc primitives respectively. The programmer may use a memory space visible logically by the CPU and the GPU called CUDA unified memory (even for discrete GPUs), therefore no memory copies are needed between CPU and GPU tasks such memory spaces (buffers) allocated using the CudaMallocManaged primitive. The current version of PRUDA supports CUDA unified memory to avoid dealing with memory copy operations, as it will be shown in PRUDA architecture. An extension to separate memory spaces is under development and will be soon available.

Typical Cuda programs are organized in the same way. First, memory is allocated both on CPU and GPU. Further, memory copies are operated between CPU and GPU. Then, the GPU kernel is launched, and finally results are copied back to the CPU by memory copy operations.

Regarding kernel execution within the GPU, authors in [1] affirm that all threads of any block are executed by only one SM, however different blocks of the same kernel may be executed on different SMs. In Figure 2, the green kernel is executed on both SM0 and SM1, the red SM is executed only on SM0. The kernel execution order and mechanisms are driven by internal closed-source NVIDIA drivers (in our case of study). A PRUDA user may obtain the SM where a given block/thread is executing by using the pruda_get_sm() function. PRUDA allows also enforcing the allocation of a given kernel to a specific SM by using PRUDA function pruda_allocate_to_sm(int sm_id), where the sm_id is the id of the target streaming multiprocessor. Implementation details about how these functions work can be found in the PRUDA description section.

To enforce an execution order between different kernels, we use a specific data structure, called *Cuda Stream*. A cuda

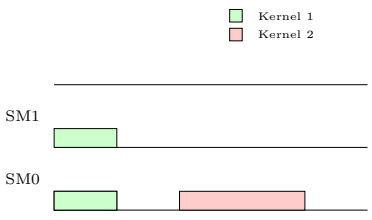


Figure 2: Example of Kernel scheduling in GPU

```
void *pruda_task(void * arg) {
  struct timespec_t next;
  p_kernel_t *pk = (p_kernel_t *)(arg);
  while (1) {
    // memory copy operation
    clock\_gettime(CLOCK\_REALTIME, &next);

    pruda_subscribe(pk->kernel,p->priority)

    timespec_addto(next, pk->T);
    clock_nanosleep(CLOCK_REALTIME, 0, &next, 0);
  }
}
```

Figure 3: Pseudo-code of PRUDA task

stream has a FIFO behavior. Therefore, kernels submitted to a Cuda stream are executed one after the other in a **sequential** fashion. Therefore, synchronization between two consecutive kernels is implicitly achieved. This property will be used later to implement non preemptive EDF and fixed priority real-time scheduling policies.

In Cuda, the user may define several streams. A priority might be set between different streams. Therefore, if a stream A has a higher priority than stream B, all kernels of A are meant to execute before kernels that are submitted to B. If a kernel in B is executing, and a kernel is activated on A, the GPU might preempt the kernel of B, to execute the kernel of A according to the GPU preemption level (we will show this behaviour in our benchmarks). We highlight that fine-grain preemption capabilities are available in NVIDIA GPUs starting from the PASCAL architecture. For example, if a preemption is set at a block level, preemption will be achieved when all already executing blocks finish their execution. Recent VOLTA GPUs allow even finer preemption levels.

Even if it is possible to create more than 2 streams, only two levels of priority are available in the Jetson TX2 platform. These properties will be used later to approximate EDF and fixed priority preemptive scheduling policies.

Other PRUDA functions will be detailed later.

3. SYSTEM MODEL

In this paper, we are only interested in GPU programming and scheduling. While this paper provides real-time support to GPUs, we do not provide any schedulability analysis yet, the analysis is work in progress.

We assume that all tasks in the system are programmed used PRUDA, therefore only PRUDA tasks are in concurrence in GPU. Each task τ_i is characterized by its deadline D_i and its period T_i. Tasks are strictly periodic, therefore the exact time between two successive activations of task τ_i is equal to T_i. The j^{th} instance of task τ_i must finish it execution no later than $\mathsf{T}_i \times j + \mathsf{D}_i$, otherwise it *misses* its deadline. The task may be scheduled using fixed priority, therefore it may be characterized by parameter priority P_i. From the implementation perspective, each PRUDA tasks is a instance of a periodic CPU thread as shown in the algorithm of Figure 3.

The PRUDA task starts by parsing the kernel parameters which are the kernel code, priority, deadline and period.

Further it starts the periodic task behavior. The task get the current time and computes the next instance activation time next. Later, the GPU job is registered in the correct (according to the desired scheduling policy) GPU run-queue (see PRUDA architecture in Figure 4). Once the PRUDA CPU thread launched the kernel, it sleeps until the next activation. Another scheduling entity checks the run-queue state and schedule the highest priority tasks first according to (i) one of the strategies details into the next section and (ii) to the desired scheduling policy.

The memory copy operation line achieves memory copies. This operation may need to copy several buffers from CPU to GPU and vice-versa. The current version of the platform use Cuda unified memory, therefore memory coherency is achieved automatically by the NVIDIA Driver.

The GPU may be scheduled as a single core platform or a parallel platform where each streaming multiprocessor is an independent core by the mean of the PRUDA __allocate_to_sm(···) function. The allocation to a given SM is achieved by testing if the task is in the correct SM, if yes, the computation is achieved, otherwise the thread on the *wrong* SM is killed.

4. TEMPORAL AND SPATIAL CONTROL OF PRUDA TASKS ON GPU

Our platform integrates several strategies to implement scheduling decisions. These strategies have different performances and overheads.

4.1 Single-stream strategy

The first strategy, called *single-stream* , uses one Cuda stream to enforce kernel scheduling decision. The scheduler uses three queues: a task queue (tq) which contains all PRUDA tasks list; an active kernels queue rq which contains the active PRUDA jobs; and the stream queue sq, which contains kernels that will be submitted to GPU. When a kernel is activated, it is added to the *correct* active kernel queue rq via the pruda_subscribe(···) function. Further, if Cuda stream queue sq is empty, it is moved from the rq to sq if it is the highest priority job according to the given scheduling policy using pruda_resched function.

As only one Cuda stream is used, once the pruda task is executing, it can not be preempted by another higher priority task, therefore only non preemptive scheduling algorithms can be implemented using this strategy. However, we would like to highlight that we allow pruda user to abort the current kernel under execution by calling pruda_abort() function.

This strategy is simple and easy to implement. It provides an implicit synchronization between active tasks, i.e. if task B is in the stream queue while A is running, B will wait until A finishes its execution before starting without overlapping. However, the use of this strategy involves reserving all the GPU resources (both SMs) for a single pruda task at a time, even if this task is not using all GPU cores, therefore resource are wasted. In the next strategies, we will show how to overcome these limitations.

4.2 Multiple stream: preemption enabling

In the second strategy, called *multiple streams*, PRUDA creates multiple streams to take scheduling decisions, allowing concurrent kernel execution on GPUs and preemption.

First, we recall that the TX2 allows only two priority levels. Therefore, we create only two streams: one with high priority and the other with low priority. The queue of the high priority stream is denoted by h-sq, the second stream queue is denoted by l-sq. We recall that using several streams allow asynchronous and concurrent execution between the two streams, however within the same stream the execution is always FIFO.

When a task is active, it is added to the correct ready-task queue rq. Further, the scheduler checks one of the following situations:

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 that the activated task has a higher priority than the task in l-sq. If yes, the task is inserted into the high priority queue h-sq, therefore it preempts the task in the l-sq if possible. Otherwise, no scheduling decision are taken.

According to the scheduling decisions mechanism described in the text above, only one preemption is allowed when a task is already in execution. For example, if a task C arrives after B has preempted A, task C must wait until B finishes even if it is the highest priority active job. We are currently developing schedulability analysis for such limited preemption system. We would like also to highlight that preempted tasks, will continue to use GPU resources if the high priority task is not using *all* of the GPU resources.

Even if this strategy solves preemption limitations of the previous one, it is more complex. It uses also a GPU as a single core. In the next section, we use each SM in the GPU as a single processor allowing parallel execution within the GPU. We highlight also that the preemption at instruction level can not be guaranteed as the later is decided by the NVIDIA closed internals. However, we ensure that the preemption can be achieved at block boundaries, therefore the worst preemption cost is in the order of the block execution.

4.3 SMs as cores strategy

The third strategy uses the GPU in similar way as the previous one; therefore two streams are created and with the same queue configuration. However, we allow tasks to call the function pruda_allocate_to_sm(···), thus using a GPU as a multiprocessor rather than a single core. We consider two types of pruda tasks : the ones that are allocated to a given SM and the other that are not (we consider that the PRUDA tasks, not calling the allocation function as tasks requiring the GPU exclusively).

In addition to the scheduling structures described for the previous strategy, this strategy uses one queue per SM : sm0-q and sm1-q. When a task is active, if it uses both SMs, no other task will be scheduled at the same time, therefore it will be added to l-sq or h-sq similarly as in the previous strategy. Otherwise, it uses a single SM and it is assigned to the correct SM queue. Later, the two job having the highest priority in sm0-q and sm1-q are scheduled first by being inserted in l-sq and h-sq. This allows parallel execution on both streaming multiprocessor. This strategy allows using the GPU of TX2 as a 2-core platform.

In fact, the allocation function tests if a given block-/thread is in the correct SM: if yes, it continues onward

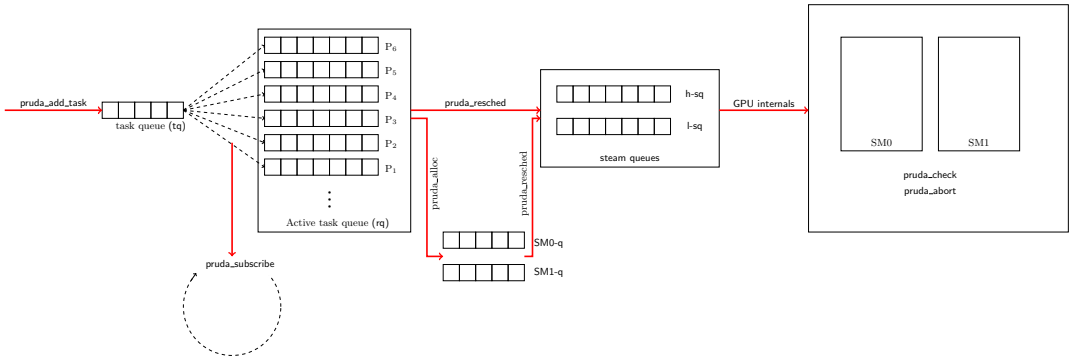


Figure 4: PRUDA global overview

execution, otherwise it exits. Therefore, the user has either to take that into account when using the block and thread indexes, or he/she must use new functions we provide to calculate indexes. The thread and block indexing mechanism we provide is simple but effective. The user is free to use the Cuda indexes, or our platform indexes, as long as there is no conflict. We highlight here that both of the previous strategies do not require any modification in the kernel code nor in the programming fashion (indexing). Although this method is more complex to implement than the two previous ones, it provides both temporal and spatial tasks execution control on GPUs. Analyzing the behavior of this final strategy is a challenging theoretical question, that is considered for future work.

5. REAL-TIME POLICIES USING PRUDA

Implementing real-time schedulers using PRUDA is simple. In fact, it requires implementing the pruda_subscribe function and the pruda_resched function. The goal of the first is to put the active task in the correct queue according to its priority. If the scheduling algorithm is fixed priority, it has to put it directly in the corresponding priority queue. If the algorithm is EDF, it requires calculating the priority and further inserting the task into the correct queue. The goal of the second function is to select which active task to select and in which Cuda stream queue it should be inserted, therefore to be submitted to the GPU. The user is also able to call pruda_abort to exit the execution of a given kernel to mix real-time with non real-time tasks if desired. The description of PRUDA provided in the current and the previous section is described in Figure 4. We highlight that pruda functions (except subscribe and resched) can be used even for non pruda tasks.

6. PRUDA API

All three strategies are integrated into the Pruda C++ API. We also have implemented for the single stream strategy both EDF and Fixed priority algorithms.

First of all, the API (Figure 4) requires the user to implement its kernel using CUDA. Further, the first step is to initialize the pruda scheduler by invoking function:

pruda_init_sched(method_t method, policy p);

where method is either SINGLESTREAM for the first strategy, MULTIPLESTREAMS for the second and for the MULTIPROC third. The policy P is the scheduling policy. The current version supports EDF or FP.

Once the scheduler has been initialized, we add kernels to the task queue tq by invoking function:

pruda_add_kernel(p_kernel_t kern, int gs, int bs, int p);

where kern is the a pointer to the kernel function, gs is the grid size, bs is the block size and p is the task priority if fixed priority policy is selected.

Once all pruda kernels have been added, the function pruda_start is invoked to start all periodic threads. Memory operations are implicitly achieved by the mean of Cuda unified memory, however explicit memory copies are under development to be soon supported.

7. CONCLUSION

In this paper, we have presented PRUDA, a programming interface to develop real-time scheduler on the top of Cuda. PRUDA provides different strategies to control temporal and space behavior of real-time tasks on the GPU. In future work, we plan to provide tools to analyze the real-time behavior of PRUDA tasks. In fact, scheduling real-time tasks does not allow free preemption and has a very limited number of priorities. These limitations has to be taken into account in the analysis of PRUDA tasks behavior to ensure the respect of timing constraints. We are also planing to develop a tool for tracing pruda tasks along with the NVIDIA nvprof profiling tool.

8. REFERENCES

[1] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *RTSS'2017*, pages 104–115. IEEE, 2017.

[2] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for gpu with preemption support. In *RTSS'2018*, pages 119–130. IEEE, 2018.

[3] Glenn A Elliott, Bryan C Ward, and James H Anderson. Gpusync: A framework for real-time gpu management. In *RTSS'2013*, pages 33–44. IEEE, 2013.

[4] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 57–66. IEEE, 2011.

[5] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

Keyword Index

Author Index