WATERS 2016

Toulouse, France

July 5th, 2016

http://waters2016.inria.fr

Keynote, Rob Davis

On the Evaluation of Schedulability Tests for Real-Time Scheduling Algorithms

This talk discusses criteria and methods that can be used to evaluate the performance of schedulability tests for real-time scheduling algorithms. We briefly review theoretical methods such as utilisation bounds, dominance relations, and speed-up factors, as well as empirical methods such as simulation and case studies. The talk then focusses on empirical evaluation and the generation of parameters for synthetic task sets. We discuss the need for a systematic approach, issues of bias, confounding variables and statistical confidence. A simple evaluation framework is outlined, covering how to generate task utilisation values, periods and other parameters, as well as how task set parameters can be grounded in data from benchmarks. A simple systematic approach to covering the parameter space is proposed, and different ways of presenting results considered. Finally, the talk ends with an open discussion of the benefits of having a de-facto standard approach, and how we might improve the quality of empirical evaluation in the real-time community.

Regular Contributions

Code Generation of Time Critical Synchronous Programs on the Kalray MPPA Many-Core architecture *Amaury Graillat*

MECHAniSer – A Timing Analysis and Synthesis Tool for Multi-Rate Effect Chains with Job-Level Dependencies *Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam and Thomas Nolte*

A Simulation Framework to Analyze the Scheduling of AVR tasks with respect to Engine Performance *Paolo Pazzaglia, Alessandro Biondi, Giorgio Buttazzo and Marco Di Natale*

Model Interpretation for an AUTOSAR compliant Engine Control Function *Sakthivel Manikandan Sundharam, Sebastian Altmeyer and Nicolas Navet*

Evaluation of Mixed-Criticality Scheduling Algorithms using a Fair Taskset Generator *Saravanan Ramanathan and Arvind Easwaran*

Dynamic criticality management with ARTEMIS *Olivier Cros, Geoffrey Ehrmann and Laurent George*

NTGEN: a Network-on-Chip Traffic Generator toolkit for latency analysis *Ermis Papastefanakis, Laurent George, Xiaoting Li and Ken Defossez*

Verification Challenge

Calculating Latencies in an Engine Management System Using Response Time Analysis with MAST *Juan M Rivas, J. Javier Gutiérrez, Julio Medina and Michael González Harbour*

A Novel Analytical Technique for Timing Analysis of FMTV 2016 Verification Challenge Benchmark *Junchul Choi, Donghyun Kang and Soonhoi Ha*

FMTV 2016: Where is the Actual Challenge? *Alessio Balsini, Alessandra Melani, Pasquale Buonocunto and Marco Di Natale*

Computational Analysis of Complex Real-Time Systems – FMTV 2016 Verification Challenge *Ingo Stierand, Philipp Reinkemeier, Sebastian Gerwinn and Thomas Peikenkamp*

Schedulability and Timing Analysis of Mixed Preemptive-Cooperative Tasks on a Partitioned Multi-Core System *Ignacio Sanudo, Paolo Burgio and Marko Bertogna*

Regular Contributions

Abstract: Code Generation of Time Critical Synchronous Programs on the Kalray MPPA Many-Core architecture

Embedded software as found in aircraft, nuclear power plants and cars, is said to be critical since bugs can have hazardous consequence to human lives. A bug can be behavioral or temporal for instance a result too late is a bad result. Hence, we talk about time-critical software. The Worst-Case Response Time is the bound of this result computation. Critical codes are often generated from formal languages such as the Dataflow Synchronous Languages (SCADE, Lustre, Esterel, etc). Today the critical systems are still running on old single-core processors since theses processors are quite simple and make the computation of the WCRT easy. But, the growing demand for computational power in avionics and automotive makes the single-core processors limited. Multi-core processors offer enough computational power but are often too complex to allow computation of the WCRT. Hence, many-core processors are promising because they offer high computational power thanks to numerous but very simple cores.

Our purpose is to parallelize and implement a Dataflow Synchronous program written in SCADE on a many-core processor. Nodes are statically scheduled on the cores to enhance temporal predictability. As the communications are in shared-memory, the solution makes the interferences as predicable as possible to allow the computation of the WCRT.

We use a prototype of the SCADE compiler that allows developer to specify the nodes of the program that must be executed in parallel. With this information, the compiler generates new blocks that communicate through data channels. A channel is composed of a data structure containing the data transferred between the blocks and special macros to write and read in this structure.

The execution platform is composed of 16 cores and a shared-memory divided into 16 banks such that access on a bank has no interference on the timing of access of another bank. According to the mapping information given by the developer, a node is executed on a core and its code stored in the corresponding memory bank. A sequencer allows several nodes to be executed sequentially one a core. Channels are implemented in a Remote Write manner, *i.e.*, the result of a node in written in the memory banks of the destination nodes. To make the interference prediction easier for each node, two releases are defined: a release for execution, and a release to write the result.

We have implemented a code generator for the Kalray MPPA Bostan many-core SoC that takes the output of the SCADE compiler and the mapping and timing information provided by the developer, to generate C code using the Kalray low level libraries.

We applied our study to ROSACE, an open source case-study of a flight controller. It is composed of an altitude controller and an environment simulator. We parallelized the altitude controller on one cluster using 5 cores. The environment simulation was located on a second cluster communicating through the network-on-chip.

MECHAniSer - A Timing Analysis and Synthesis Tool for Multi-Rate Effect Chains with Job-Level Dependencies

Matthias Becker*, Dakshina Dasari†, Saad Mubeen*, Moris Behnam*, Thomas Nolte*
*MRTC / Mälardalen University, Sweden {matthias.becker, saad.mubeen, moris.behnam, thomas.nolte}@mdh.se
† Research and Technology Centre, Robert Bosch, India dakshina.dasari@in.bosch.com

***Abstract*—Many industrial embedded systems have timing constraints on the data propagation through a chain of independent tasks. These tasks can execute at different periods which leads to under and oversampling of data. In such situations, understanding and validating the temporal correctness of end-to-end delays is not trivial. Many industrial areas further face distributed development where different functionalities are integrated on the same platform after the development process. The large effect of scheduling decisions on the end-to-end delays can lead to expensive redesigns of software parts due to the lack of analysis at early design stages. Job-level dependencies is one solution for this challenge and means of scheduling such systems are available. In this paper we present MECHAniSer, a tool targeting the early analysis of end-to-end delays in multi-rate cause effect chains with specified job-level dependencies. The tool further provides the possibility to synthesize job-level dependencies for a set of cause-effect chains in a way such that all end-to-end requirements are met. The usability and applicability of the tool to industrial problems is demonstrated via a case study.**

I. Introduction

Many application domains for embedded systems are subject to timing constraints in order to fulfill their requirements. Such real-time systems are well studied and several tools are available to analyze these properties. However, for many systems it is not only important that the individual tasks execute within their specified deadlines, but also that data propagates through a chain of tasks within a specified end-to-end delay constraint. In the automotive industry such chains are called cause-effect chains [1], [2]. The tasks in such a chain can have different activation periods which makes the calculation of such end-to-end delays a challenging task since over and undersampling effects need to be considered.

Currently it is left to the discretion of the system designer to guarantee that all end-to-end delay constraints are met in the system. While this is viable in small applications, the growing complexity of industrial applications renders this approach increasingly difficult. Automotive applications for example contain several multi-rate cause-effect chains [3]. Additionally, one task can be part of several chains which increases the problem complexity further.

This highlights the need for tool support during the system design, giving the designer viable input during early stages of the development where only limited or even no concrete knowledge of the schedule is present. This need is further increased since applications of several suppliers may be integrated on the same Electronic Control Unit (ECU) during the system integration which is usually done by the Original Equipment Manufacturer (OEM). Changes in the system design can be very expensive at this stage. Having means to obtain end-to-end delay bounds for the data propagation through a chain of tasks before the system integration can thus provide valuable information and reduce the risk of costly design changes in the later development phases.

One way to reduce the possible data propagation among tasks of different rate is the use of job-level dependencies [4]. A job-level dependency introduces a constraint in the data propagation between two tasks and is specified on job-level. Several works address the scheduling problem of systems with specified job-level dependencies. These works cover fixed-priority and dynamic priority scheduled systems [5], [6], as well as time triggered schedules [7], [8]. The problem of analyzing such systems and to synthesize job-level dependencies is addressed in [9].

A. Contributions

Several available tools support the end-to-end delay analysis of cause-effect chains, which are primarily based on the principles proposed in [10]. They however assume that knowledge of the task schedule is available when the system is analyzed. In contrast, the proposed tool MECHAniSer can be helpful in the early design phases where the exact task schedule is unknown. Its key features include analysis to i) compute bounds on the end-to-end delays ii) synthesize job-level dependencies when specified timing constraints are violated iii) compute end-to-end analysis in the systems where job-level dependencies are specified. To facilitate a faster system design, the tool implements a heuristic to place job-level dependencies in a system consisting of several, possibly interconnected, cause-effect chains. This is done in a way such that the maximum data age delay of each cause-effect chain is met.

B. Paper Layout

The rest of the paper is organized as follows, in Section II the system architecture and background information is provided. In Section III the calculations to obtain the data age delay are described before the tool itself is discussed in Section IV. The tool is evaluated based on a case study in Section V, followed by a discussion of related tools in Section VI and the conclusions and future work in Section VII.

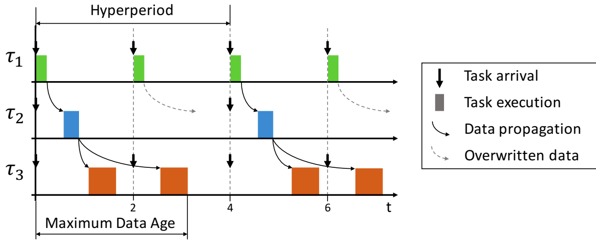


Fig. 1: Data propagation between tasks of a cause-effect chain in a real-time system with maximum data age specified.

II. System Architecture and Background

A. System Model

The system is comprised of a set of periodic tasks Γ. Each task $\tau_i \in \Gamma$ can be described by the tuple $\{C_i, T_i\}$, where C_i is the task's Worst Case Execution Time (WCET), and T_i is the task's period. All tasks have implicit deadlines, i.e. the deadline of τ_i is equal to T_i. For all tasks executing on a processor, the hyperperiod can be defined as the least common multiple of all periods, $HP = \text{LCM}(\forall T_i, i \in \Gamma)$. Hence, a task τ_i executes a number of jobs during one HP, where its jth job is denoted by $\tau_{i,j}$.

B. Communication Model

In this work inter task communication is realized via shared registers, a model commonly used in the industrial domain [10], [11]. With this, a sending task writes an output value to a shared register, which is then read by the receiving task without the need for any signaling between the communicating tasks. Also, the receiving task always consumes the newest value present in the shared register.

In order to facilitate determinism, a *read-execute-write* semantic is followed in which a task reads all its input values into *local copies* before the execution starts. It then executes by acting on these local copies and writes the output values after the execution back to the shared registers, making them available to other tasks. In short, reading and writing of input and output values is done at deterministic points in time, i.e. at the beginning and end of the tasks execution respectively. This is a common communication mechanism found in several industrial standards (i.e. in AUTOSAR this model is defined as *implicit communication* [12], the standard IEC 61131-3 for automation systems defines similar communication mechanisms [13]).

C. End-to-End Timing Requirements

A cause-effect chain is typically specified by an end-to-end timing requirement, as defined for automotive systems in [1], [2]. In this work the *data age*, the most important timing requirement in control systems, is examined. A detailed discussion of corresponding end-to-end delays is provided in [10]. For data age, the maximum time from sampling an initial input value at the beginning of the cause-effect chain, until the last time this value has influence on the produced output of the cause-effect chain is of interest. Fig. 1 depicts an example with three tasks, τ_1, τ_2, and τ_3. All tasks are part of a cause-effect chain in this order. Note that τ_1 and τ_3 are activated with a period of $T = 2$, while τ_2 is activated with a period of $T = 4$. This leads to over- and under-sampling between the different tasks. While the output value of the first instance of τ_1 is consumed by the first instance of τ_2, the data produced by the second instance of τ_1 is overwritten before τ_2 has the chance to consume it. Similarly, data produced by the first instance of τ_2 is consumed by the first instance of τ_3. Since no new data is produced before the second instance of τ_3 is scheduled the same data is consumed by τ_3 again. In the example, this constitutes the maximum data age, from sampling of the first instance of τ_1 until the last appearance of the data at the output of the second instance of τ_3.

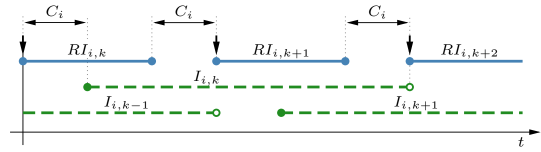


Fig. 2: Read and data intervals of consecutive jobs of τ_i if no scheduling information is available.

D. Job-Level Dependency

A job-level dependency is similar to the rate transition operator of PRELUDE [4]. Defined between two tasks, a job-level dependency specifies which job of a task needs to finish its execution before a certain job of the successor task can start.

A job level dependency is described as $\tau_i \xrightarrow{(k,l)} \tau_j$, meaning that the kth job of τ_i needs to proceed the lth job of τ_j. This also implies that the dependency between the two jobs applies for the duration of the hyperperiod of the two jobs only, e.g. $LCM(\tau_i, \tau_j)$.

III. Calculating Latencies

In this section, we recapitulate the calculation of data propagation paths for systems without prior knowledge of the schedule. For a more in depth explanation a reader is referred to [9]. Several properties of tasks under register communication are observed to determine reachability between jobs. Based on this the different data propagation paths of the cause-effect chain can be calculated.

A. Reachability between Jobs

The concepts of *read interval* and *data interval* are central to decide if data can be propagated between two distinct jobs. For a job $\tau_{i,j}$, the read interval is defined as the interval starting from the earliest time $\tau_{i,j}$ can potentially read its input data ($R_{min}(\tau_{i,j})$) until the last possible time $\tau_{i,j}$ can do so without violating its timing constraints ($R_{max}(\tau_{i,j})$). Similarly, the data interval is defined as the interval from the earliest time the output data of $\tau_{i,j}$ can be available ($D_{min}(\tau_{i,j})$) up to the latest time a predecessor job of the same task overwrites the data ($D_{min}(\tau_{i,j})$). Hence, the read interval $RI_{i,j}$ is the interval $[R_{min}(\tau_{i,j}), R_{max}(\tau_{i,j})]$, and the data interval is $[D_{min}(\tau_{i,j}), D_{max}(\tau_{i,j}))$. These concepts are depicted in Fig. 2 for jobs of a task τ_i. For a system without any knowledge of the scheduling decisions, one has to assume that a job can be scheduled anywhere, as long as it starts not

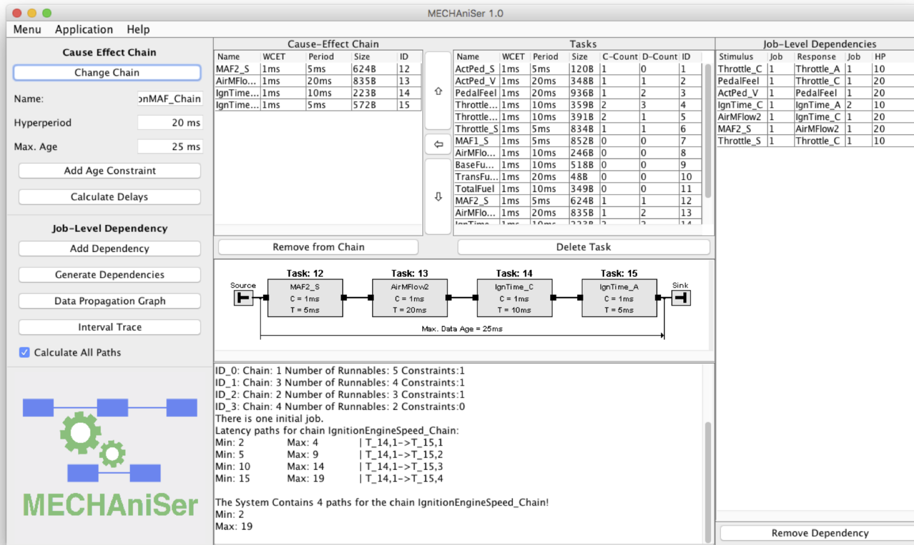
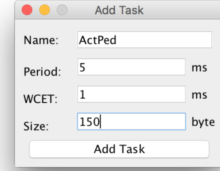
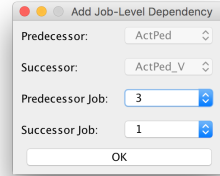


Fig. 3: Main view of the tool.



(a) Window to add a task.



(b) Window to add a dependency.

Fig. 4: Windows to add new elements.

before its release and finishes not after its deadline. In [9], the notations to define the intervals are as follows:

$$
\begin{aligned}
R_{min}(\tau_{i,j}) &= (j-1)\cdot T_i \\
R_{max}(\tau_{i,j}) &= R_{min}(\tau_{i,j+1}) - C_i \\
D_{min}(\tau_{i,j}) &= R_{min}(\tau_{i,j}) + C_i \\
D_{max}(\tau_{i,j}) &= R_{max}(\tau_{i,j+1}) + C_i
\end{aligned}
$$

1) Deciding Reachability between Jobs: In order for a job $\tau_{k,l}$ to consume data of a job $\tau_{i,j}$ the data interval of $\tau_{i,j}$ must intersect with the read interval of $\tau_{k,l}$. The function $\text{Follows}(\tau_{i,j}, \tau_{k,l})$ is defined to return *true* if this is the case:

$$
\text{Follows}(\tau_{i,j}, \tau_{k,l}) = \begin{cases} true, & \text{if } RI_{i,j} \cap DI_{i,j} \neq \emptyset \\ false, & \text{otherwise} \end{cases}
$$

2) Adjusting the Data Interval for Long Chains: In order to capture the characteristics of data propagation in a cause-effect chain of length > 2, the data interval needs to be modified. Assume the first job of τ_i, as shown in Fig. 2 is followed by a job of a task τ_k. τ_k is released with same period as τ_i, but its execution time is shorter than the one of τ_k. $\text{Follows}(\tau_{i,1}, \tau_{k,1})$ returns true and indicates that $\tau_{i,1}$ can potentially consume the data of $\tau_{k,1}$. However, in order to decide reachability between the $\tau_{k,1}$ and a third task in the chain the data interval of $\tau_{k,1}$ must be modified. This is the case because $\tau_{k,1}$ can consume the data of $\tau_{i,j}$ earliest at time $D_{min}(\tau_{i,j})$. Consequently, this data can earliest be available as output data of $\tau_{k,l}$ at time $D_{min}(\tau_{i,j}) + C_k$. $D'_{min}(\tau_{k,l}, \tau_{i,j})$ defines the starting time of the data interval of $\tau_{k,l}$ if the data produced by $\tau_{i,j}$ shall be considered as well:

$$
D'_{min}(\tau_{k,l}, \tau_{i,j}) = \max(D_{min}(\tau_{i,j}) + C_k, D_{min}(\tau_{k,l}))
$$

Note that the data interval only needs to be adjusted if $D_{min}(\tau_{k,l})$ is smaller than $D_{min}(\tau_{i,j}) + C_k$. These modifications are local for the specific data path, hence, if another combination of jobs is involved the original data interval must be used.

B. Calculating Data Paths

To calculate all possible data propagation paths in a system, a recursive function is used. This function constructs all possible data propagation paths from a job of the first node in a cause-effect chain up to the job of a last node of the chain. Consequently this needs to be done for all jobs of the first task of a chain, inside the hyperperiod of the chain.

As a result a set of data propagation paths is provided, where each path comprises an ordered list of involved jobs.

C. Constructing Data Propagation Paths and Max. Data Age

For a given data path, the maximum end-to-end latency and the data age, is computed. Given τ_{start} is a job of the first task of the cause-effect chain, and τ_{stop} is a job of the last task of a cause-effect chain:

$$
\text{AgeMax}(\tau_{start}, \tau_{end}) = (R_{max}(\tau_{end}) + C_{\tau_{end}}) - R_{min}(\tau_{start})
$$

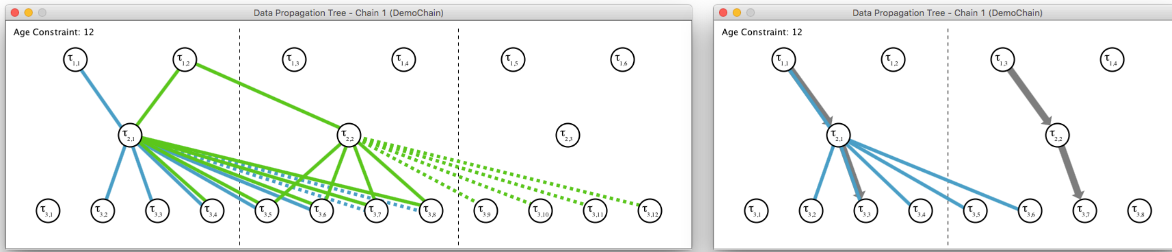
In order to compute the maximum data age for any possible path in the system, $\text{AgeMax}()$ must be computed for all data paths. The maximum of these values is the maximum data age of the cause-effect chain.

IV. Tool Layout and Usage

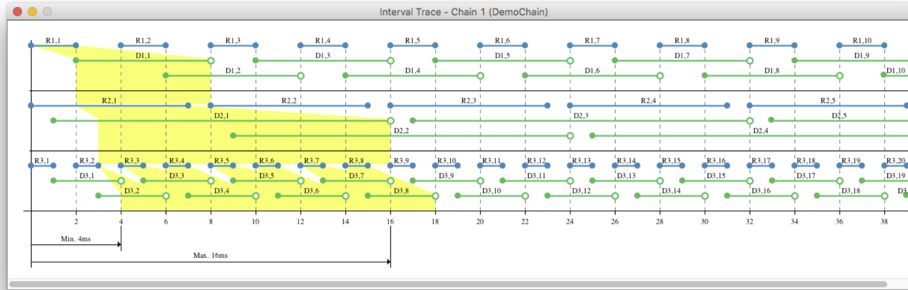
This section briefly outlines the different forms of data input to the tool. Further the tool layout and its usage are discussed and a closer look is provided into the different visualization options.

A. Input Formats

The tool specifies its own XML format to save a current project. Additionally it is possible to import projects designed with AMALTHEA V1.0[14]. AMALTHEA is an open tool platform for the design of multi-core systems in the automotive domain. The implementations for the support of additional



(a) The graph view in MECHAniSer. Edges between nodes depict possible data propagation while dashed edges show paths leading to larger data age then specified with the age constraint. The same chain with generated job-level dependencies is shown in the right window.



(b) The trace-view of MECHAniSer depicts the read- and data-interval of each involved job and visualizes the minimum and maximum data age of initial jobs as well as their possible data propagation range (in yellow).

Fig. 5: The two different visualization options for a cause-effect chain.

tools (i.e., AMALTHEA V1.1, Rubus ICE [15]) are currently ongoing and will be made available in the future.

B. Layout and Usage

The tool is built around a main panel which is shown in Fig 3. The panel depicts the chain under analyis and also provides clickable interfaces to additional features of the tool.

1) The Main Panel and its Parts: The main window displays information about all tasks of the system, in the *"Tasks"*-table, as well as on all specified job-level dependencies in the *"Job-Level Dependency"*-table. The selected chain is graphically visualized, as shown in Fig. 3, while the *"Cause-Effect Chain"*-table describes the different parameters of the chain. This chain can further be analyzed and modified. The left column also provides means to manage job-level dependencies. The additional views can also be opened here via the button *"Data Propagation Graph"* and *"Trace View"*. Output for the user is provided in the text-box at the bottom part of the window.

A user can add or delete a task over the *"Application"*-menu (see Fig. 4a) with the *"Add Task"* and *"Delete Task"* buttons. Note that the tool also displays the number of chains and the number of job-level dependencies that a task is part of. In order to keep the system consistent, a task must first be removed from all cause-effect chains and from all job-level dependencies before it can be removed from the system.

The chain which needs to be analyzed is selected via the button *"Change Chain"*. This action pops up a window wherein a user can select the desired cause-effect chain. Once approved, the tool updates the related views. A new task can be added to the chain by selecting the respective task in the task table and then clicking the left-arrow button which appends the task to the chain. The correct position of a task is set by selecting the task in the chain table and then clicking the up- and down-button which alter the tasks position. A task can be removed from the chain by selecting the task followed by the button *"Remove from Chain"*.

Finally a maximum data age constraint can be specified on the chain by clicking on the button *"Add Age Constraint"*. This pops up a window where the age constraint can be specified. Note that this new input overwrites any previously specified constraint. A constraint can be removed by specifying a maximum data age of 0.

2) Calculating Minimum and Maximum Data Age: The minimum and maximum data age of the currently selected cause-effect chain under consideration of all specified job-level dependencies can be computed by clicking on the button *"Calculate Delays"*. This action computes delays by applying the analysis presented in [9]. All data propagation paths are calculated, implying all possible paths that the data can propagate, when read from any of the initial jobs of the chain.

An initial job is defined as any job that the first task of the cause-effect chain releases during the first hyperperiod of the chain. Since the number of possible paths depends on the number of involved tasks as well as on the involved periods, a large number of data propagation paths might be generated. A user has hence the possibility to uncheck the option *"Calculate All Paths"* which will only calculate the data propagation path for the minimum and maximum job at each chain level. Hence this reduces the complexity of the calculation and simplifies the post processing by the system designer.

3) Adding and Synthesizing Job-Level Dependencies: The second strength of the tool is to handle job-level dependencies.

The left column of the main window provides means to add a job-level dependency manually as well as to synthesize job-level dependencies for all cause-effect chains in the system. The button *"Add Dependency"* opens a new window (see Fig. 4b) which allows to select the two involved tasks and the dependent instances. Note that first the two tasks need to be selected before the menu for the involved jobs becomes active. This is the case since, depending on the selected tasks, the available job instances change.

The button *"Generate Dependencies"* triggers a heuristic [9] which adds job-level dependencies to the system in a way that all specified age-constraints are met. Already specified dependencies are not affected. The main intuition behind the heuristic is that a placement of a job-level dependency can prune a branch of the data propagation tree. Hence the heuristic adds dependencies in a way such that all branches which lead to larger end-to-end delays than specified are removed.

4) The Graph View: The graph view, as shown in Fig. 5a, depicts the data propagation tree of the currently selected chain. Each data path originating from the different initial nodes is colored differently for a more effective visual presentation. The different jobs of the involved tasks are drawn in a way that the data always propagates from top to bottom, i.e. the beginning of the chain is at the top and the last task of the chain is at the bottom. Branches which lead to end-to-end delays larger than the specified constraint are shown in dashed lines. These branches need to be removed in order to meet the specified constraints. Note that this representation depicts no time information, the execution of the jobs depends on the exact path a data propagates and hence cannot be shown in this overview. However, jobs are grouped such that jobs of the same hyperperiod are arranged together and separated by the vertical dashed lines. A user can obtain further information of the different nodes by clicking on them which then displays an information box.

5) The Trace View: The trace view is shown in Fig. 5b. This view visualizes the read- and data-interval of all jobs of one chain (see Fig. 2 for a description). Initially the first initial job is selected and the propagation of the calculated data paths is visualized via yellow overlay. Additionally the minimum and maximum data age of these data paths are shown. A user can change this view to any other initial job by clicking on the respective read interval.

C. Implementation and Distribution

To be platform independent, the tool is developed in Java. The main development is performed under OSX which might cause a diverging visual appearance on other platforms. The tool is freely available online[1]. A user documentation and examples are provided under the same link.

V. Case Study

The applicability of the presented tool is demonstrated on a case study of an Engine Management System (EMS). This case study is adapted from the results presented in [11]. The EMS consists of several subsystems which control the air

[1]http://www.mechaniser.com

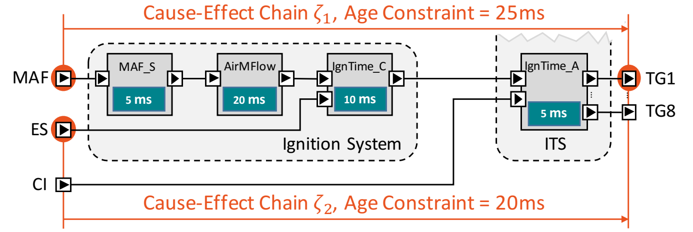


Fig. 6: Tasks and specified cause-effect chains of the IS and ITS.

and gas mixture which is injected into the cylinders. The Air Intake System (AIS) controls the amount of air via the throttle position, while the Fueling System (FS) controls the amount of gas which is injected per stroke. The Ignition System (IS) controls the exact time of the ignition, both FS and IS feed into the Injection Time and Ignition Time Actuation System (ITS). For a smooth and energy efficient operation of the vehicle, several age constraints must be met. The complete EMS of this case study comprises 16 different tasks which three different periods ($5\,\mathrm{ms}$, $10\,\mathrm{ms}$, and $20\,\mathrm{ms}$)

Due to space limitations, we discuss only part of the complete EMS. The case study includes two cause-effect chains, ζ_1, and ζ_2, which are specified from the Mass Air Flow (MAF) input to the output for Ignition Time of cylinder 1 to 8 (TG1-8). The cause-effect chain ζ_2 is specified from the Engine Speed (ES) input up to TG1-8. Both chains span from the IS to the output of the ITS. 4 tasks with 3 different periods are involved (see Fig. 6) and WCETs of all tasks are set to $1\,\mathrm{ms}$. We refer to [9] for a case study of the AIS.

A. Analysis of Latencies using MECHAniSer

Both specified cause-effect chains contain a number of runnables which are triggered at different periods. For the chain ζ_1 all four tasks are involved. The calculation of all data propagation paths results in 70 different paths, a minimum data age of $4\,\mathrm{ms}$, and a maximum data age of $55\,\mathrm{ms}$. The maximum possible data age exceeds the specified age constraint of $25\,\mathrm{ms}$ and the chain is not directly schedulable by the system. In the next step we will show how the tool generates job-level dependencies to remove the data propagation paths which exceed the constraint.

The second chain ζ_2 consists only of two tasks. Hence, the number of data propagation paths is smaller. Four paths are identified, with a minimum data age of $2\,\mathrm{ms}$ and a maximum data age of $20\,\mathrm{ms}$. Here the specified age constraint of $20\,\mathrm{ms}$ is met without the need to specify job-level dependencies.

The required computation time for the analysis of the two chains is $5\,\mathrm{ms}$ and $2\,\mathrm{ms}$ for ζ_1 and ζ_2 respectively.

B. Synthesizing Job-Level Dependencies

The initial analysis of the two cause-effect chains revealed that, while ζ_2 meets its age constraint, ζ_1 does not. Hence job-level dependencies need to be generated in order to meet the constraint.

The tool generated three different job level dependencies in order to meet the constraint of the cause-effect chain. One job-level dependency was generated between each consecutive pair of tasks. This successfully reduces the maximum data age to $25\,\mathrm{ms}$, allowing the cause-effect chain to meet its constraint.

The presence of the job-level dependencies further reduces the number of data propagation paths to 13. The required computation time is $19\,\mathrm{ms}$. Since ζ_2 is subset of ζ_1, the specified job-level dependency between the last two tasks of the cause-effect chain can have influence on ζ_2, hence ζ_2 needs to be revalidated as well. The job-level dependency specified for the two tasks is defined between the first job of task *IgnTime_C* to the second job of task *IgnTime_A*, the parameters are not influenced and the latency stays at $20\,\mathrm{ms}$ with 4 different data propagation paths.

VI. Related Tools

Many industrial standards specify constraints for the propagation of data through a chain of tasks [1], [2]. A detailed discussion of end-to-end delays is provided in [10]. The authors formally specify age- and reaction delays in multi-rate systems which communicate via register-communication and further develop a method to calculate end-to-end delays in such systems.

Several commercially available tools support the analysis of end-to-end delays in cause-effect chains. Examples are SymTA/S TraceAnalyzer for ECUs [16], Rubus ICE [15], and Timing Architects Inspector [17].

To the best of our knowledge the analysis presented in [10] is implemented in these tools [18], [19]. EELAP [20] is an open source end-to-end analyzer for the ProCom [21] real-time component model. The tool is built on the analysis of [10]. All these tools however require an existing schedule in order to analyze the system. Hence, the calculation of end-to-end delays in early design phases is not supported.

Several works address systems where job-level dependencies are specified [5], [6], [7], [8]. The application model in these works is specified by the *prelude* language [4] which specifies the rate-transition operation. On task level this operation is equivalent to a job-level dependency. The prelude compiler is available [22] and can generate synchronized multi-task C-code which then can be executed by the supported target OS. To the best of our knowledge, no tool exists that can automatically generate job-level dependencies in order to meet the end-to-end timing constraints.

VII. Conclusion and Future Work

In this paper we presented MECHAniSer, the first tool for the analysis and synthesis of multi-rate cause-effect chains with specified job-level dependencies. The tool allows to analyze systems at early design phases, where detailed scheduling knowledge is not available. Further, the tool synthesizes job-level dependencies for a set of cause-effect chains in a way that all their end-to-end timing constraints are met. This allows such systems to be scheduled on any platform which supports these concepts [5], [6], [7], [8].

The tool provides its own XML format to store the project configurations but it also provides the possibility to import projects from existing tools and hence eases the design process. Multiple graphical views are provided to support the system designer and to ease the understanding of the data propagation in multi-rate cause-effect chains. Several extensions to the tool are possible. One limitation of the current implementation is the time granularity. Future versions of the tool will allow to specify time values in smaller granularity than ms. Besides data age, many industrial applications specify reaction constraints. Analysis for this type of constraint is currently not supported but will be part of future work.

Acknowledgment

The work presented in this paper is supported by the Swedish Knowledge Foundation (KKS) through the projects PREMISE and DPAC; and the Swedish Foundation for Strategic Research (SSF) through the projects PRESS.

References

[1] *AUTOSAR - Spec. of Timing Extensions*, AUTOSAR Std. 4.2.2, 2014.
[2] *EAST-ADL - Domain Model Specification*, EAST-ADL Association Std. V2.1.12, 2014.
[3] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2015.
[4] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A real-time architecture design language for multi-rate embedded control systems," in *ACM Symposium on Applied Computing*, 2010, pp. 527–534.
[5] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010, pp. 301–310.
[6] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task implementation of multi-periodic synchronous programs," *Discrete Event Dynamic Systems*, vol. 21, no. 3, pp. 307–338, 2011.
[7] W. Puffitsch, E. Noulard, and C. Pagetti, "Off-line mapping of multi-rate dependent task sets to many-core platforms," *Real-Time Systems*, vol. 51, no. 5, pp. 526–565, 2015.
[8] ——, "Mapping a multi-rate synchronous language to a many-core processor," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 293–302.
[9] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *Proceedings of the 22th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, [Online] http://www.es.mdh.se/publications/4368-, 2016.
[10] N. Feiertag, K. Richter, J. Norlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics," in *Int. Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2008.
[11] P. Frey, "Ulmer Informatik Berichte Nr 2010-03 - Case Study: Engine Control Application," University Ulm, Tech. Rep., 2010.
[12] *AUTOSAR - Specification of RTE*, AUTOSAR Std. 4.2.2, 2014.
[13] *IEC 61131-3*, International Electrotechnical Commission Std., 2003.
[14] AMALTHEA, "An Open Platform Project for Embedded Multicore Systems," [Online] http://www.amalthea-project.org/index.php/contact, last visited 16.05.2016.
[15] Arcticus Systems, "Rubus ICE," [Online] https://www.arcticus-systems.com/products/, last visited 16.05.2016.
[16] Symtavision GmbH, "SymTA/S and TraceAnalyzer for ECUs," [Online] https://www.symtavision.com/products/ecu-timing/, last visited 16.05.2016.
[17] Timing Architects, "Timing Architects Inspector," [Online] https://www.timing-architects.com/ta-tool-suite/inspector/, last visited 16.05.2016.
[18] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, 2005.
[19] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study," *Computer Science and Information Systems*, vol. 10, no. 1, 2013.
[20] J. Kuncar, R. Inam, and M. Sjödin, "End-to-end latency analyzer for ProCom - EELAP," Tech. Rep. ISSN 1404-3041 ISRN MDH-MRTC-272/2013-1-SE, 2013.
[21] R. Inam and M. Sjödin, "Implementing and evaluating communication-strategies in the procom component technology," *SIGBED Rev.*, vol. 9, no. 4, pp. 41–44, 2012.
[22] Prelude, "programming critical real-time systems," [Online] http://www.lifl.fr/%7Eforget/prelude.html, last visited 16.05.2016.

Accepted as abstract at Waters 2016

A Simulation Framework to Analyze the Scheduling of AVR tasks with respect to Engine Performance

Paolo Pazzaglia, Alessandro Biondi, Marco Di Natale and Giorgio Buttazzo
Scuola Superiore Sant'Anna, Pisa, Italy
Email: {name.surname}@sssup.it

***Abstract*—We present a simulation framework, based on Simulink and an extension of the T-Res scheduling simulator tool to help provide a better characterization of the very popular problem of scheduling and analysis of Adaptive Variable Rate Tasks (AVR) in engine control. The purpose of the tool is to go beyond the simplistic model that assumes hard deadlines for all tasks and to study the impact of scheduling decisions with respect to the functional implementations of the control algorithms and the true performance of the engine.**

I. Introduction

The study of the schedulability conditions for engine control tasks (or adaptive variable rate - AVR) is gaining popularity in the real-time research community because of the novel nature of the problem and the special activation conditions that apply to some of the system tasks. These tasks are not periodic or sporadic, but are activated by the rotation of the engine crankshaft (a parameter of the physical controlled system). In addition, to compensate for the increased CPU load at high rotation speeds (and more frequent activation times), the code implementation of these tasks is defined in such a way that at given speed boundaries, the implementation is simplified and the execution time is reduced. A typical engine control application consists of time-driven periodic tasks with fixed periods, typically between a few milliseconds and 100 ms (see [1], page 152), and angular tasks triggered at specific crankshaft angles. The activation rate of such angular tasks hence varies with the engine speed (variable-rate tasks). For example, for engines where the speed varies from 500 to 6500 revolutions per minute (RPM), the interarrival times of the angular tasks range from about 10 to 120 ms (assuming a single activation per cycle).

With respect to the set of activation instants, the dependency from a physical phenomenon characterizes this problem as truly belonging to the class of problems in cyber-physical systems (CPS). However, in many papers the dependency of the timing and scheduling problem from the physics of the controlled system is restricted to the set of activation events and every other concern is hidden under the typical assumption of hard deadlines.

In reality, this problem (as many others) is representative of a class of control systems in which deadlines can be missed without catastrophic consequences, and the problem should actually be defined as a design optimization, where the objective is to select the controls implementations and the scheduling policy in such a way that a set of engine performance functions are optimized (including power, emissions, noise, pollution). These performance functions depend in complex ways from timing parameters, such as jitter and latency. Informally, the objective of the scheduler is not to miss too many deadlines or produce actuation signals that are too much delayed.

Formally, the problem is quite complex and extremely unlikely to be solved in a simple, closed analytical form or even with a general procedure for expressing the dependency of th performance from scheduling. This is the reason for the investigation of alternative approaches that are based on the simulation of the three system components in a joint environment:

- A model of the engine and the combustion process in it (the physical system or plant)
- A model of the engine controls
- A model of the task configuration and the scheduling

II. Our Simulation Framework for the Analysis of the Performance impact of Scheduling

Our cosimulation framework follows the principles of CPS system analysis. It is based on the popular Simulink toolset and leverages the T-Res cosimulation environment for the simulation of the task scheduling [2].

For the development of the engine model we leveraged information from several sources, including engine models for the steady state and event-based models as described in [1] and other empirical models found online.

The engine controls are currently extremely simple and only contain a simple analytical formula that computes the angle of injection and the injection time that is defined by a calibration table.

Finally, the T-Res simulation framework described in [2] is used for modeling the scheduling delays.

III. Extending T-Res for modeling AVR tasks

T-Res consists of a set of custom Simulink blocks representing tasks and kernels and allows to interface the Simulink simulation engine, acting as master, with a scheduling simulator in a co-simulation environment (see Figure 1). The scheduling simulator (we use RTSim [3], but the backend simulation engine can be changed) computes the scheduling delays and latches the outputs of the corresponding tasks until their simulated completion time. This allows to simulate delays in the production of output values and the corresponding impact on the control function.

T-Res provides a custom block for representing the kernel and its scheduler. The block is configured with the selection of the scheduling policy and the behavior in case of deadline (period) overrun. The kernel block provides a set of activation signals as output. These activation signals go to instances of the the second type of custom blocks, representing tasks. Each task receives an activation signal from the kernel (indicating when the task begins or resumes execution), and is characterized by an execution time estimate (a configuration parameter), and a signal going back to the kenel and providing the amount of time that is still required by the task at each point in time. The task block produces as output a set of activation and latch signals for all the functional subsystems that are executed by the task.

With respect to the activation, sporadic tasks are characterized by an activation event going as input to the kernel block, or a periodic activation specification, provided as a configuration parameter to the kernel (for details, refer to [2]). The execution time description is provided to the kernel for each task using a simple language.

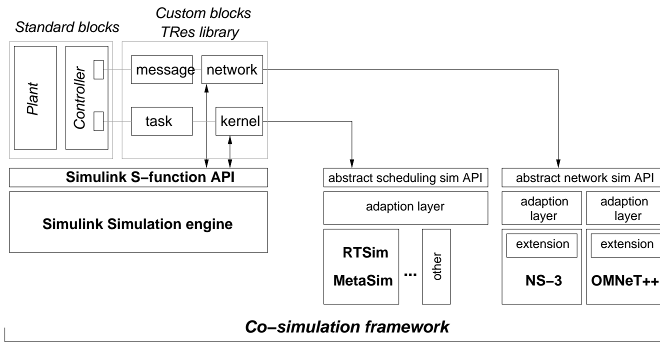


Figure 1. The TRes cosimulation architecture.

For the purpose of this project we entended the task model block and the timing information associated with it to allow for the modeling of the AVR behavior, as shown in Figure 2. The task block in T-Res includes a signal for the explicit activation in case of event-triggered tasks. This signal is used to define the activation of the task in correspondence to given angular positions of the engine crankshaft. In addition, the block has been extended to include another imput that refers to a *mode* index. This input can be used for multiple purposes and defines a different execution time behavior for a finite and enumerated set of conditions.

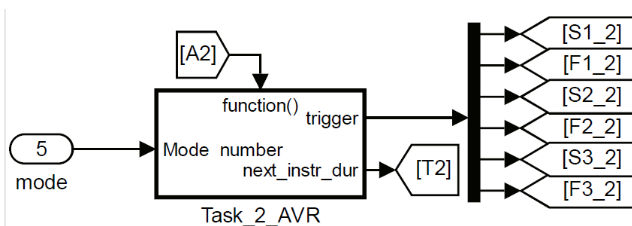


Figure 2. A custom block for modeling an AVR task.

In the case of AVR tasks, the mode index is provided from a simple block that looks at the engine rotation speeds and, based on the speed range, defines the execution time that the task requires.

The task will have different execution times for different speed modes according to a specification of execution times as a function of the mode (speed) index provided as a workspace variable.

IV. Simulink Models of the Engine and the Control Tasks

Figure 3 shows the model of the engine and the control functionality in Simulink. The blocks in the upper part of the figure represent the engine subsystems that are currently considered and includes the turbocharger, the compressor manifold, the intercooler, the intake and exhaust manifolds and the model of the engine cylinders. The subsystem on the bottom part of the figure wraps our model of the engine controller, with its outputs: the injection angle and duration and the VGT.

Figure 4 shows the subsystems realizing the controller functions and the task model of the controller. The model consists of a kernel (top left side), and four tasks on the bottom left side. One of the four tasks is an AVR, two are periodic and one represents background computations. The chains of subsystems on the right side represent the control functions implemented by the tasks. The second from the top contains the six subsystems that are executed by the AVR task (matching the six output signals from the AVR task block).

V. Objective and Status

A detailed modeling of the control function is necessary to better understand the impact of deadline misses or long latencies. Depending on the implementation of the control function, a deadline miss may result in a late actuation, or a missed actuation or even an actuation with old data. In our controls implementation, the AVR task computes the phase and duration of the injection and passes them to the task that simulates the injection actuators. Hence, a missed deadline results in actuating the injectors with the values computed in the previous cycle with a likely error in phase and duration with respect to the ideal values.

The objective of our framework is multifold:

- To understand the effect of the scheduling on the engine performance and to use the environment for analyzing the impact of scheduling policies and parameters, such as evaluating fixed priority vs EDF or different possible priority assignments and task configurations.
- To analyze the timing parameters that truly of interest for evaluating the performance of the engine and possibly attept a characterization that isolates the attributes of interest. This includes, among others, the evaluation of schemes like m-k deadline misses, or overload management (maximum lateness).

- To better characterize the design problem consisting in the optimal selection of the transition speeds for AVR tasks.

Currently, within the assumptions of our model, the simulation is able to show how the scheduling delays result in errors in the angle/duration of the injection actuation. Figure 5 shows preliminary results. In the figure graph, the vertical axis shows the phase error in the actuation of the injection for a sample manoeuvre consisting of a sudden acceleration and a corresponding increase in the engine rotation speed from low to high values. Two graphs are plotted in the figure. The graph in red (lighter) color shows the angle error when the execution time of the AVR task is kept constant, regardless of the engine speed. At high rotations, the task misses deadlines and the injection angle error grows to almost 50 degrees. When the execution time of the AVR task is reduced at high rates, the scheduling delays are much lower and, correspondingly, the angle error of the injection is much lower, as shown by the blue line in the graph. The angular error in the injection is related to a variation (loss) in the power performance of the engine.

Our objective is to relate the errors in phase and duration of the injection to a possible loss of power, providing ways to analyze the impact of scheduling with respect to the first performance function of interest. However, even within the limited scope of power performance analysis, the evaluation of the scheduling impact (and the AVR characteristics of tasks), requires that the model includes multiple representations of the control functionality, one for each possible execution mode of the AVR tasks. When these are available, the model will provide an early capability of expressing the performance impact of control implementations at different levels of complexity (for variable execution times or WCETs). Clearly, this is only the initial objective, given that a realistic model should also include the characterization of pollution, noise and efficiency.

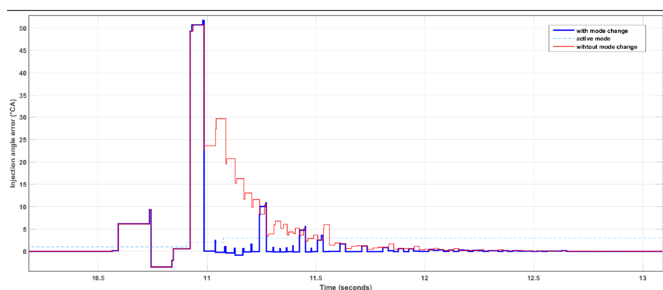


Figure 5. Angular error in the injection caused by scheduling delays of the AVR task: error with fixed execution times (red) and with adaptive excution (in blue).

VI. Related Work

The presentation of the task model in which engine control tasks are implemented with a variable computational requirements for increasing speeds is in [4],

These tasks are also referred to adaptive variable-rate (AVR). Analyzing the schedulability of tasks sets consisting of both periodic and AVR tasks is a difficult problem that has been addressed by several authors under various simplifying assumptions, under both fixed priority scheduling [5]–[7] and Earliest Deadline First (EDF) [8]–[10]. Other authors proposed methods for computing the exact interference [11] and the exact response time [7] of AVR tasks under fixed priority scheduling. It has been shown [10] that, given the large range of possibile interarrival times of an AVR task, fixed priority scheduling is not the best choice for engine control systems since, while EDF exhibits a nearly optimal scheduling performance. Based on this fact, Apuzzo et al. [12] provided an operating system support for AVR tasks under the Erika Enterprise kernel [13].

All the papers considered above, however, focused on analyzing the schedulability of task sets consisting of periodic and AVR tasks, without any concern on engine performance. A performance-driven design approach has been addressed in [14] for finding the transition speeds that trigger the mode changes of an AVR task.

A very large number of projects target the evaluation of scheduling policies and the analysis of task implementations. A necessarily incomplete list includes Yartiss [15], ARTISST [16], Cheddar [17], and Stress [18].

Finally, TrueTime [19] is a *freeware*[1] Matlab/Simulink-based simulation tool that has been developed at Lund University since 1999. It provides models of multi-tasking real-time kernels and networks that can be used in simulation models for networked embedded control systems. TrueTime is used by many research groups worldwide to study the (simulated) impact of lateness and deadline misses on controls. In TrueTime, the model of task code is represented by *code functions* that are written in either Matlab or C++ code. Several research works investigate the consequences of computation (scheduling) and communication delays on controls. An overview on the subject can be found in [20].

References

[1] L. Guzzella and C. H. Onder, *Introduction to Modeling and Control of Internal Combustion Engine Systems*. Springer-Verlag, 2010.

[2] F. Cremona, M. Morelli, and M. D. Natale, "Tres: A modular representation of schedulers, tasks, and messages to control simulations in simulink," in *Proc. of the 31st ACM Symposium on Applied Computing (SAC 2016)*, Pisa, Italy, April 4-8, 2016.

[3] L. Palopoli, G. Lipari, L. Abeni, M. D. Natale, P. Ancilotti, and F. Conticelli, "A tool for simulation and fast prototyping of embedded control systems," in *LCTES/OM*, S. Hong and S. Pande, Eds. ACM, 2001, pp. 73–81.

[4] D. Buttle, "Real-time in the prime-time," in *Keynote speech at the 24th Euromicro Conference on Real-Time Systems*, Pisa, Italy, July 12, 2012.

[5] J. Kim, K. Lakshmanan, and R. Rajkumar, "Rhythmic tasks: A new task model with continually varying periods for cyber-physical systems," in *Proc. of the Third IEEE/ACM Int. Conference on Cyber-Physical Systems (ICCPS 2012)*, Beijing, China, April 2012, pp. 28–38.

[6] R. I. Davis, T. Feld, V. Pollex, and F. Slomka, "Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling," in *Proc. 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, Berlin, Germany, April 2014.

[1] http://www3.control.lth.se/truetime/LICENSE.txt

[7] A. Biondi, M. D. Natale, and G. Buttazzo, "Response-time analysis for real-time tasks in engine control applications," in *Proceedings of the 6th International Conference on Cyber-Physical Systems (ICCPS 2015)*, Seattle, Washington, USA, April 14-16, 2015.

[8] G. Buttazzo, E. Bini, and D. Buttle, "Rate-adaptive tasks: Model, analysis, and design issues," in *Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE 2014)*, Dresden, Germany, March 24-28, 2014.

[9] A. Biondi and G. Buttazzo, "Engine control: Task modeling and analysis," in *Proc. of the International Conference on Design, Automation and Test in Europe (DATE 2015)*, Grenoble, France, March 9-13, 2015, pp. 525–530.

[10] A. Biondi, G. Buttazzo, and S. Simoncelli, "Feasibility analysis of engine control tasks under EDF scheduling," in *Proc. of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*, Lund, Sweden, July 8-10, 2015.

[11] A. Biondi, A. Melani, M. Marinoni, M. D. Natale, and G. Buttazzo, "Exact interference of adaptive variable-rate tasks under fixed-priority scheduling," in *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*, Madrid, Spain, July 8-11, 2014.

[12] V. A. A. Biondi and G. Buttazzo, "OSEK-like kernel support for engine control applications under EDF scheduling," in *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016)*, Vienna, Austria, April 11-14, 2016.

[13] "Erika enterprise: an OSEK compliant real-time kernel." [Online]. Available: http://erika.tuxfamily.org/drupal/

[14] A. Biondi, M. D. Natale, and G. Buttazzo, "Performance-driven design of engine control tasks," in *Proceedings of the 7th International Conference on Cyber-Physical Systems (ICCPS 2016)*, Vienna, Austria, April 11-14, 2016.

[15] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, M. Qamhieh *et al.*, "Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms," in *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2012, pp. 21–26.

[16] D. Decotigny and I. Puaut, "Artisst: an extensible and modular simulation tool for real-time systems," in *Object-Oriented Real-Time Distributed Computing, 2002.(ISORC 2002). Proceedings. Fifth IEEE International Symposium on*. IEEE, 2002, pp. 365–372.

[17] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *ACM SIGAda Ada Letters*, vol. 24, no. 4. ACM, 2004, pp. 1–8.

[18] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Stress: A simulator for hard real-time systems," *Software: Practice and Experience*, vol. 24, no. 6, pp. 543–564, 1994.

[19] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén, "How does control timing affect performance?" *IEEE control systems magazine*, vol. 23, no. 3, pp. 16–30, 2003.

[20] K. J. Astrom and B. Wittenmark, "Adaptive control," in *Prentice Hall*, 2016.

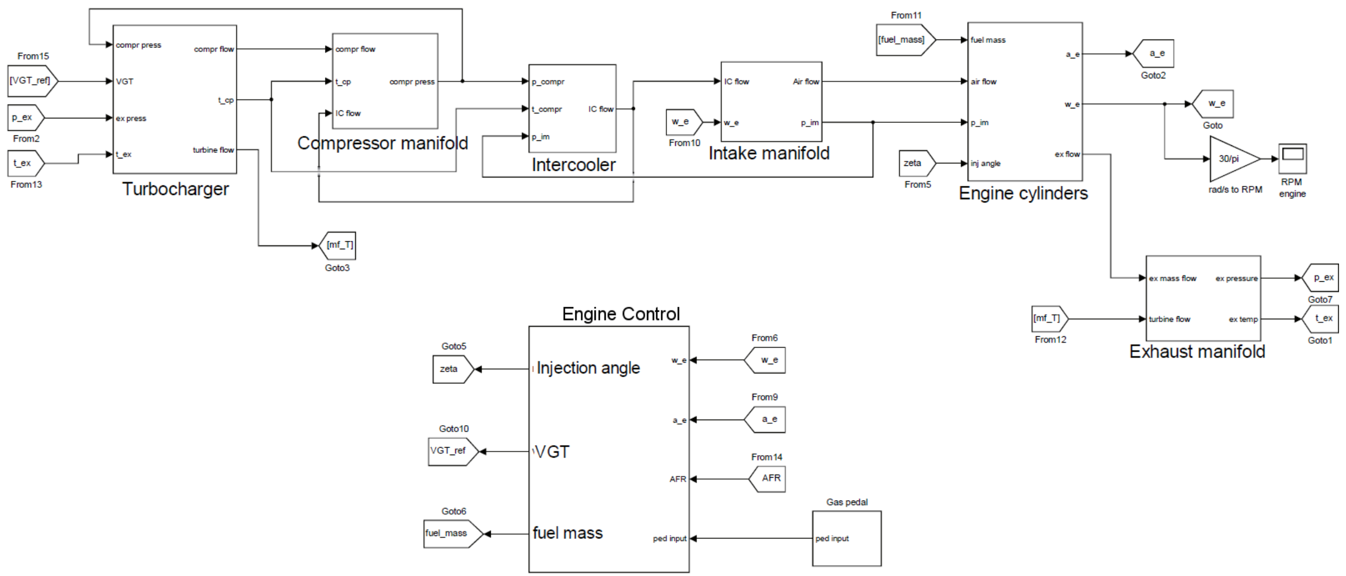


Figure 3. Engine control model in Simulink.

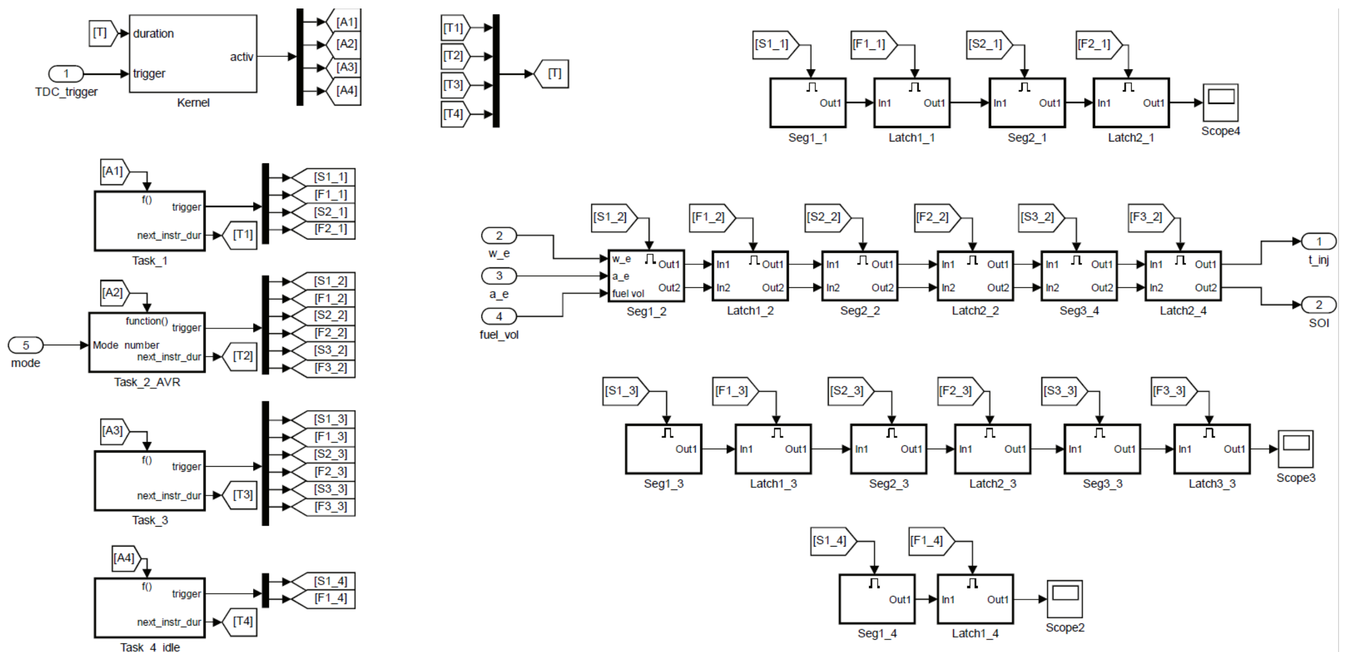


Figure 4. Task model in TRES.

Model Interpretation for an AUTOSAR compliant Engine Control Function

Sakthivel Manikandan Sundharam
University of Luxembourg
FSTC/Lassy
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg
sakthivel.sundharam@uni.lu

Sebastian Altmeyer
University of Amsterdam
CSA Group
Science Park 904
1098XH Amsterdam
altmeyer@uva.nl

Nicolas Navet
University of Luxembourg
FSTC/Lassy
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg
nicolas.navet@uni.lu

***Abstract*—Model-Based Development (MBD) is a common practice in the automotive industry to develop complex software, for instance, the control software for automotive engines, which are deployed on modern multi-core hardware architectures. Such an engine control system consists of different sub-systems, ranging from air system to the exhaust system. Each of these sub-systems, again, consists of software functions which are necessary to read from the sensors and write to the actuators. In this setting MBD provides indispensable means to model and implement the desired functionality, and to validate the functional, the non-functional, and in particular the real-time behavior against the requirements. Current industrial practice in model-based development completely relies on generative MBD, i.e., code generation to bridge the gap between model and implementation. An alternative approach, although not yet used in the automotive domain is model interpretation, the direct interpretation of the design models using interpretation engine running on top of the hardware. In this paper, we present a case study to investigate the applicability of model interpretation, in contrast to code generation, for the development of engine control systems. To this end, we model an engine cooling system, specifically the calculation of the engine-coolant temperature, using interpreted model based development, and discuss the benefits and low-lights compared to the existing code-generation practice.**

I. Introduction

Model-Based Development (MBD), also frequently referred to as Model-Driven Engineering (MDE), denotes the use of models as the main artifacts to drive the development of systems. It has been profoundly reshaping and improving the design of software-intensive embedded systems specifically. Traditionally, model-driven development (based on code generation) is deployed in the automotive industry. Code generation is used to generate code from a higher level model and create a working application.

As mentioned in [4], Model-Based Development is being used for series development by a majority of the automotive companies. Especially in development phases i.e., system design and coding, the model-based design is used extensively. As mentioned in [7], this kind of MBD used by automotive suppliers and car manufacturers is called *generative MBD*, since code and other artifacts are automatically generated from the model.

The other fundamental approach to achieve applications from models is *interpreted MBD*. Interpreted MBD can be seen as a set of platform independent models that are directly interpreted by an execution engine running on top of the hardware, with or without an operating system.

The fact that models can be directly executable helps a great deal as the development cycle time can be shortened; and there is no distortion between the model and what is executed. Though, to the best of our knowledge, the technique of model interpretation remains unexplored in the automotive domain, it can facilitate and speed up the development, deployment and timing verification of applications with real-time constraints running on potentially complex hardware platforms. Verification also can be done more easily as defects will be caught earlier in the process since there is no difference between the model and the executable program. In this paper, we present a case-study to evaluate how *interpreted MBD* can be applied to an automotive software development scenario.

This paper is structured as follows. In Section II, we explain the state of the industrial practice of automotive function development. Section III describes an AUTOSAR-compliant engine-coolant temperature calculation function used as case-study. In Section IV, we discuss our modeling approach, and Section V presents the case study. Finally, Section VI summarizes the results and discusses the case study. Section VII concludes the paper.

II. Automotive Function development - state of the practice

We explain the state-of-the-art of the development of an automotive function using an automotive engine management software system, which are commonly developed using a Model Based Development (MBD). The engine is controlled by an Electronic Control Unit (ECU) that contains engine functions for different sub-systems.

The requirements of the engine functions are specified in one of the Application Life-cycle Management (ALM) suites and traced until its realization as ECU. In ALM, different tools are integrated to develop and maintain the software. For example, IBM has an ALM suite called IBM Rational Team Concert (RTC) where Rational DOORS is the requirements management tool that captures all the functional and non-functional requirements. These requirements are analyzed further to design the engine function. Popular Model Based

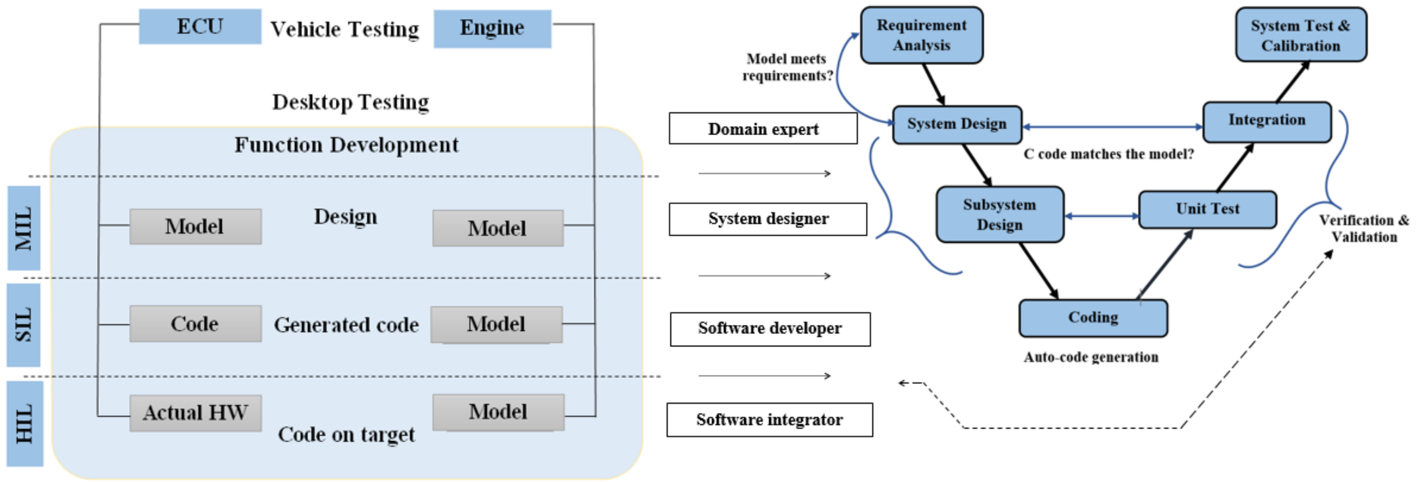


Fig. 1. Engine function development flow - Illustration of verification techniques, involved stakeholders and development phases.

Design (MBD) tools are MATLAB/Simulink (MLSL) from Mathworks, ASCET-MD from ETAS, and SCADE Suite from ANSYS. These industrial MBD tools further generate code for engine functions using code generators. Each engine control function is further (unit-) tested and integrated into the ECU.

Figure 1 shows the software function development flow practiced in the automotive industry. The system model of the engine captures the ideas and requirements. The model is an executable specification and can be simulated and rapid-prototyped to explore different design options. In the existing approach, the modeling environment is primarily used to describe the domain problem, in this case the engine function to be developed against the functional requirements. *Domain experts* and *software designers* are involved in this phase. The controller model is tested in a simulation environment (which includes the plant model, *i.e.* the engine) and this testing is called *Model-in-the-Loop* (MiL) testing to ensure that the model meets the requirements.

In the next step, the code is generated from the model using a code generator. Then, the code is verified under an engine model. This phase is referred to as *Software-in-the-Loop* (SiL) testing. *Software developers* are involved to test each engine function individually using unit testing. Next, the function is integrated with other existing engine functions in the integration phase by the *Software integrators*, typically a tier-one supplier. The complete engine software is then ported to the ECU hardware, which can be verified using a *Hardware-in-the-Loop* (HiL) testing system, such as PT-LABCAR, which realistically emulates vehicles I/Os.

In the current practice [3], the execution environment on the target is different from the execution within the modeling environment in terms of I/Os, scheduling and even in terms of generated code. Indeed, the target-generated code will be optimized towards the platform and thus be as efficient as possible. On the negative side, the build tool-chain must be available, and it takes a substantial amount of time to produce an executable program from the designed model (build time can require several 10s of minutes). Simulink and its block sets (like Simscape, Stateflow etc.) are examples for modeling environment and Embedded Coder is an example of the code generator for production code generation on a specific target processor. The generated code can be further customized to meet the requirements (e.g., with respect to safety). In the automotive software development, there is a high probability for mixed-mode development, where generated code is integrated with manually-developed functions.

III. AUTOSAR-COMPLIANT ENGINE FUNCTION

The engine cooling system is an important part of the vehicle. It is responsible for maintaining optimum operating temperature. The coolant is circulated through the engine block with the help of an electric water pump. The coolant will reduce the temperature of the engine block and then will run through the radiator equipped with a fan to remove waste heat.

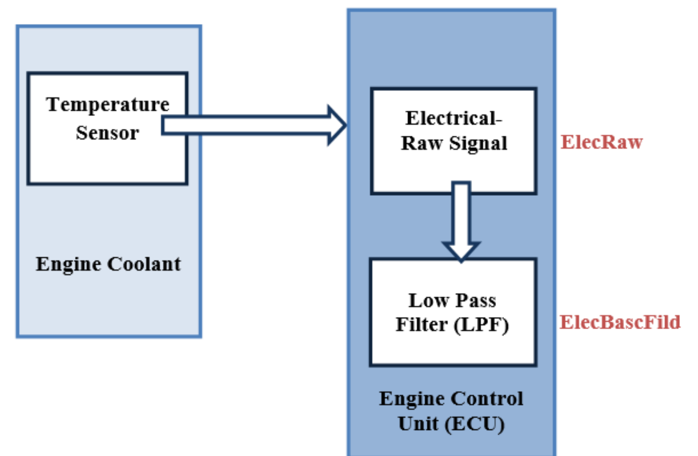


Fig. 2. Physical layout of an AUTOSAR compliant engine-coolant system function - Engine coolant temperature sensor connected to an ECU

Figure 2 shows the physical layout of the engine-coolant temperature calculation which is considered as the use case to

present our modeling approach. The engine-coolant temperature sensor plays an indispensable role in the engine cooling system. Precise information about the temperature is essential due to various reasons: the data are used by the engine control unit to adjust the fuel injection and ignition timing. Further, the temperature value is used to control the cold starting of the engine, to control the calculation of the fuel quantity, and to control the fan speed of the electric cooling radiator. This data is also used to provide readings of the coolant temperature gauge to the dashboard to protect the engine from over-heating.

The engine-coolant temperature sensor is connected to the engine ECU through an analog to digital pin. The electrical output is obtained from the sensor that monitors the temperature of the engine-coolant. As per AUTOSAR design pattern [2] catalogue for standard sensors, the overall system consists of 3 modules as depicted in Figure 3. *Sensor/Actuator Components* are special AUTOSAR software components which encapsulate the dependencies of the application on specific sensors or actuators. The AUTOSAR architecture takes care of hiding the specifics of the micro-controller (this is done in the micro-controller abstraction layer, MCAL, part of the AUTOSAR infrastructure running on the ECU) and the ECU electronics (handled by the ECU-Abstraction layer, also part of the AUTOSAR Basic Software).

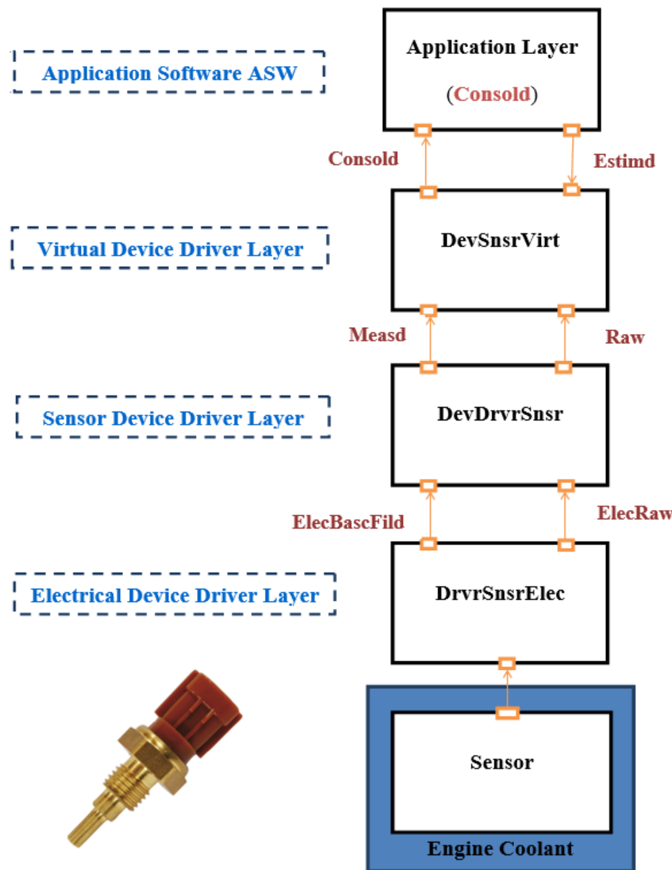


Fig. 3. AUTOSAR design pattern for a standard sensor

The architecture of the engine-coolant temperature calculation function involves 3 AUTOSAR software components:

Electrical Device Driver Layer (DrvrSnsrElec):
: The electrical value from the temperature sensor is read through the input pin and stored in the variable *ElecRaw*. The raw electrical signal (*ElecRaw*) is rugged against signal faults using the Low Pass Filter (*LPF*) and the filtered raw electrical signal (*ElecBascFild*) is obtained.

Sensor Device Driver Layer (DevDrvrSnsr):
: At this stage, the raw electrical signal is converted into its physical temperature value (*Raw*) using a lookup-table, where the corresponding value is provided. The temperature value of the filtered electrical signal (*ElecBascFild*) is also obtained from the lookup-table and is provided to the next layer.

Virtual Device Driver Layer (DevSnsrVirt):
: In this layer, the possible signal range check, electrical errors, cable interruption and sensor faults that may occur are identified. This is done in order that incorrect values from the sensor are not taken into account for the calculation in case of sensor malfunctioning. Other errors such as a cable interruption, short circuit to battery or sensor voltage saturation can also be detected and appropriate flags will be set:

- *ElecBascFildbit* - The electrical validity bit shows that the sensor raw value is electrical valid.
- *ElecBascFildbitCommon* - The common validity bit shows that the engine-coolant temperature as a whole is valid and can be transfered to the application Layer. Based on the temperature values calculated in this layer, the obtained temperature value (*Measd*) is compared with the estimated value (*Estimd*) from the application layer. This comparison determines the validity of the calculated value. If valid, the final temperature value (*Consld*) is sent to the application layer.

IV. Function development - Proposed approach

To the best of our knowledge, model interpretation for automotive function development has not been explored and experimented in the past. In case of model interpretation, a generic model-interpretation engine is implemented which executes the model of the engine function. As shown in Figure 4, the modeling environment includes the execution environment. Hence, the executable artifacts (*i.e.*, model and execution engine) are available within this environment. The model interpretation can be launched within the development environment or on a target platform. In the latter case, the interpretation can run on top of an OS or directly on the hardware. There are two possible interpretation modes: simulation and real-time. Simulation mode is suited for the use in the design phase, where execution should be as fast as possible, which implies that the activation frequencies of the processes

are not respected and they execute (conceptually) in zero time. Typically, executing in simulation mode is several orders of magnitude faster than in the real-time mode. Real-time mode is for the execution of the program with the actual desired temporal behavior of the application.

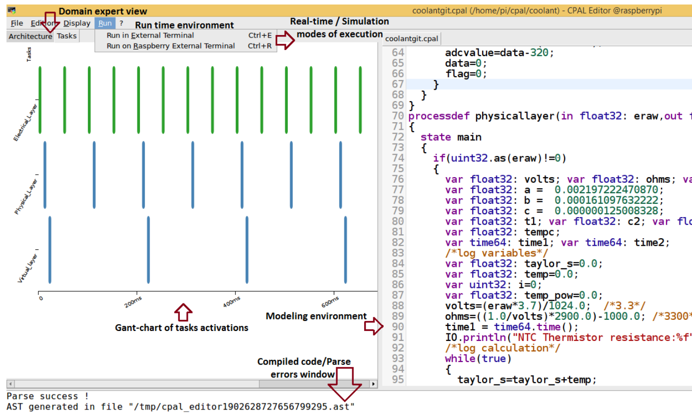


Fig. 4. An integrated environment, here the CPAL-Editor, with the code of the model, the Gantt chart of the processes activations and the possibility to execute the models in simulation and real-time mode both locally or on a target.

To ensure that simulation reflects the real-time behavior on the target platform, timing annotations (e.g., execution time latencies, jitters, etc) can be introduced in simulation mode. Those timing annotations can be derived from measurements on the target architecture, from WCET analysis and, possibly, by schedulability analysis if other software components can interfere with the function under development. Timing accurate simulation thus provides benefits to identify faults in design phase itself, earlier, thus than with the traditional design process.

As the model itself can be executed, no additional artifacts are needed, and, unlike in the traditional generative MBD, no target specific code is generated. Instead, the specifics of the platform are taken care by the interpretation engine. Further steps of the application development, such as compilation of source code to object code and the linking stage to produce the executable program, are not required.

V. A Case Study - Engine-coolant temperature calculation

The model of the engine-coolant system is developed in the CPAL (Cyber Physical Action Language, see [1, 6]), which is a new language to model, simulate, verify and program Cyber Physical Systems. CPAL[1] is a language jointly developed by our research group at the University of Luxembourg and the company RTaW. Many industrial use-cases are demonstrated [5] using CPAL in the past.

The model-based environment of CPAL consists of a single integrated development environment, i.e., the CPAL-Editor. The CPAL editor, combines the design, simulation, execution (both locally and on a target), visualization of the functional architecture and execution chronogram in one integrated environment. The model-interpretation engine is specific to the target platform. This interpretation engine can be executed on top of an operating system or without an operating system, the latter being called Bare-Metal Model Interpretation (BMMI). CPAL BMMI is available on the NXP Semiconductors Freedom-K64F, a low-cost development platform which is form-factor compatible with the Arduino R3 pin layout. The experiments in this study are performed on a Raspberry Pi equipped with a multi-core ARM Cortex-A7 processor operating at 900 MHz running Raspbian OS.

A typical engine-coolant temperature sensor can measure in the range −40°C to +150°C. In our case study, we have considered a Negative Temperature Coefficient (NTC) type sensor with an operating voltage as 3.3V. Figure 5 shows the experimental setup which aims to mimic the engine cooling system. The MCP3008 is an external ADC interface which is connected to the sensor. Since the sensor operates with the thermistor principle, a voltage divider circuit with 3.3V reference is added. ADC data from MCP3008 is communicated to the processor using the Serial Peripheral Interface (SPI). The sensor software component is modeled according to the AUTOSAR design catalog described in Section III. The speed of the electric fan is controlled based on the measured temperature.

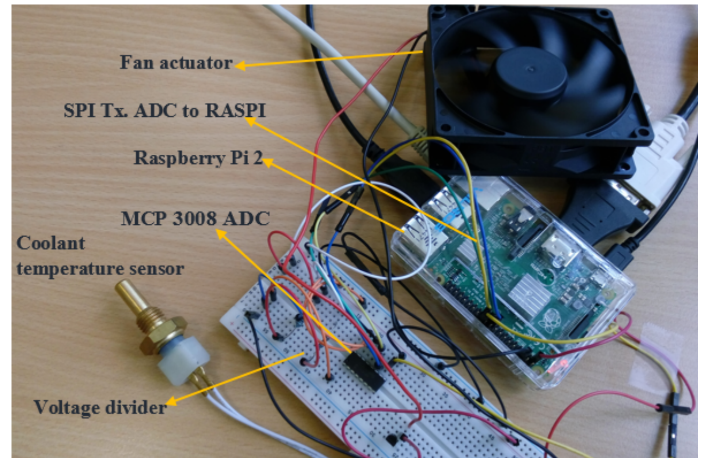


Fig. 5. Experimental set-up - Sensor interfacing to hardware

Out of the two possible CPAL execution environments (*i.e.*, bare-metal or hosted by an OS), we use the interpretation engine on top of an OS (Raspbian on Raspberry Pi) which can also execute in real-time, although with a lesser real-time predictability than the bare-metal implementation. The engine-coolant temperature is calculated by the sensor software component modeled in CPAL. Figure 6 shows the sample run-time environment where simulation and real-time execution are performed. Both interactive and non-interactive executions are possible. The interactive mode of execution is useful in program analysis and debugging. In interactive mode, the user has different execution options, such as a step-by-

[1]The CPAL documentation, graphical editor and the execution engine for Windows, Linux and Raspberry Pi platforms are freely available from http://www.designcps.com.

step execution, or uninterrupted execution for a pre-defined duration.

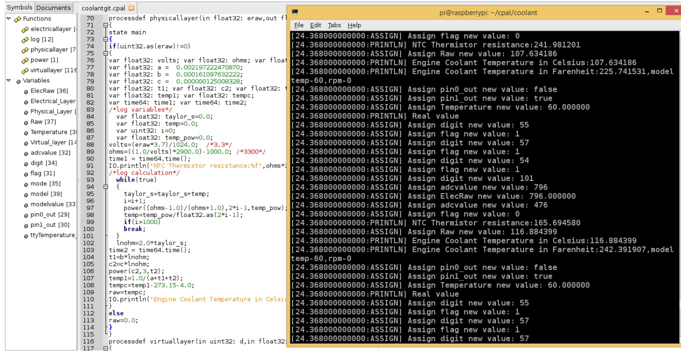


Fig. 6. CPAL model and execution environment under real-time mode

Since it is an interpretation-based execution environment, the user can list and change the values of global variables at run-time, as well as execute additional code statements. In non-interactive mode, the program is executed indefinitely or for a specified duration without requiring additional user inputs.

VI. Results and Discussions

From the case-study experience, we present our proposed development flow for function development. Figure 7 shows the development flow of model interpreted approach to develop an engine function.

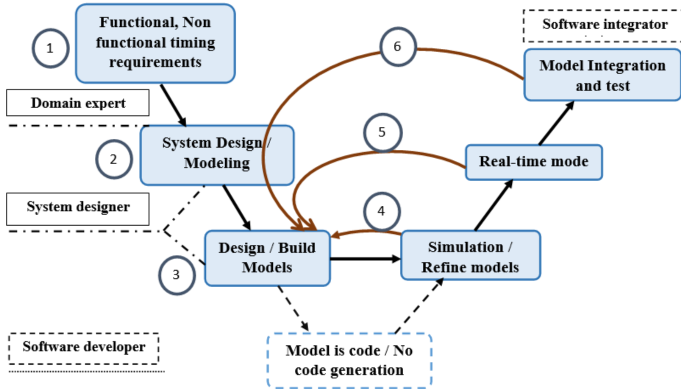


Fig. 7. Model interpreted engine function development flow - steps and stakeholders involved

a) Model interpreted development steps: In the first step all functional, non-functional including timing requirements of the engine function are collected. These are further analyzed by domain experts. The specifications are implemented in CPAL (step 2 - system design in Figure 7). During the development, as soon as the function model is updated the functional architecture, and other views created out of the model such as execution Gantt charts, are automatically updated too (step 3) which is done in the background along with the modifications. This allows the designer to immediately visualize and understand the effects of the changes made, without the need for building the executable and running it in debug mode. The latest version of the model is always available to execute, be it in simulation mode or real-time mode, locally or on a target. Typically performed once the simulation is satisfactory (step 4), the execution in real-time mode (step 5) helps the designer to assess the performances on the target, enabling rapid-prototyping. If simulation or execution in real-time mode highlights faults, the model is refined in an iterative process. From the development of the engine-coolant temperature calculation function, we here summarize the benefits and differences against the existing generative MBD approach.

b) Adapting to requirement changes is faster: The most important benefit of model interpretation is that changes in the model do not require an explicit regeneration/rebuild/retest/redeploy step. This shortens significantly the turnaround time and, in some scenarios, the overall change management process (how changes in the requirements are implemented). Although it is not available in CPAL yet, it would be possible for models to be updated at run-time, without the need to stop the running application, hence improving productivity. Also, since no artifacts are generated, the build times can be also reduced. Depending on the specific use case, an interpreter combined with model can even require less memory than generated code.

c) Finding failures in model is easier: Failures during the testing phase, after all modules have been integrated, expose problems that are clearly in the model, since the model itself is executed. Unlike with code generation, there is no need to trace back from the generated artifacts where the failure occurred in the model, which is often hard. On the other hand, debugging models at run time is possible. Since the model is available at run-time, it is possible to debug function models by stepping through them at run-time (e.g., we can add breakpoints at the model level). When debugging at model level is possible, domain experts can debug their own models (e.g., step-by-step) and adapt the functional behavior of an application based on this debugging. This can be very helpful when, for example, complex control or data-flows are involved.

d) Portability and hardware independence: Portability is another advantage of model interpretation. An interpreter in principle creates a platform independent target to execute the model. By rewriting only the hardware-specific components, it is possible to develop an interpreter which runs on multiple platforms, as it is the case for CPAL. In case of code generation, we need to make sure we generate code that is specific to the platform. In case of model interpretation, the interpreter handles the platform-specific adaptation.

A notable advantage of the model interpretation is that it hides the complexity of the hardware platform away from the programmer making it easier to configure the run-time environment and deploy the application. Indeed, easier deployment is an important difference. When code generation is used, we often see that we need to open the generated source code in an Integrated Development Environment (IDE) to analyze the program and build it from there to create the final application.

In case of BMMI, we just have to upload the model and reset the target, or, when the interpreter is hosted by an OS, execute it within the development environment or in command-line (possibly on a target through a script). Hence, it is much easier for domain experts to deploy and test an application, instead of only modeling it.

e) Benefits of single integrated environment: The important difference between interpreted approach and generative is that domain experts and software developers can work together around a single integrated environment and on a single model. As shown in Figure 8, the integrated modeling environment provides a graphical view of the architecture of the designed function model. This model can be used by domain experts for functional analysis and verification, and by software engineers to do function development and testing from day one on.

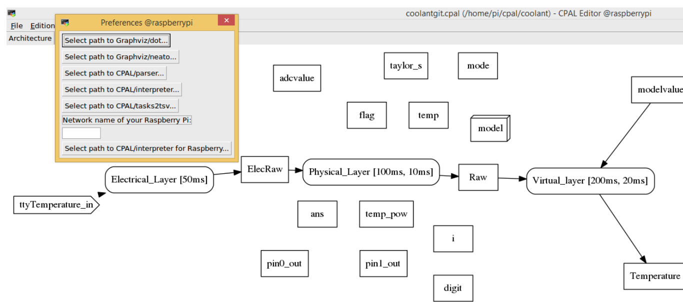


Fig. 8. Software architecture of the coolant temperature calculation

VII. Conclusions

Code generation is the standard practice in the industry for MBD of embedded systems, and this holds true in particular for engine function development. In this paper, we discuss a model-interpretation development flow that is exemplified with the development of an engine coolant temperature calculation by an AUTOSAR compliant software architecture. By comparison with the usual development chains relying on code-generation and based on the case-study, we discuss the benefits of model interpretation which includes simplicity, productivity and early-stage verification possibility, specifically in the time dimension. For instance, CPAL, the model-based development environment that we have chosen for our case study, already provides the basic mechanisms to offer timing-realistic simulation early in the design process. Our ongoing work is on a method to automate the derivation of the temporal quality-of-service required by a software module and, leveraging on model-interpretation, enforce it at run-time.

Although model-interpretation brings advantages, it is not going to cover all use-cases. The main reason is that model interpretation is intrinsically slower than compiled code. There are ways to mitigate this drawback in production code such as calling binary code from interpreted code (e.g., legacy code or specialized functions) or, possibly, selectively generating code for the computation-intensive portions of the model. Interpretation and code generation are often seen as two alternatives, not as a continuum. However, one may also imagine relying on model-interpretation, and benefits from the associated productivity gains, until the function/ECU meets all functional requirements, and then switch to code-generation for production code. This remains to be investigated in the future works.

Acknowledgment

This research is supported by FNR (Fonds National de la Recherche), the Luxembourg National Research Fund (AFR Grant n°10053122).

References

[1] S. Altmeyer, N. Navet, and L. Fejoz. Using CPAL to model and validate the timing behaviour of embedded systems. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Lund, Sweden, July 2015.

[2] AUTOSAR consortium. AUTOSAR design catalogue. http://www.autosar.org/fileadmin/files/releases/4-2/application-interfaces/general/auxiliary/AUTOSAR_TR_AIDesignPatternsCatalogue.pdf.

[3] M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu. Seamless model-based development: From isolated tools to integrated model engineering environments. *Proceedings of the IEEE*, 98(4):526–545, 2010.

[4] M. Broy, S. Kirstan, H. Krcmar, B. Schätz, and J. Zimmermann. What is the benefit of a model-based design of embedded software systems in the car industry? *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, page 310, 2013.

[5] L. Fejoz, N. Navet, S. M. Sundharam, and S. Altmeyer. Applications of the CPAL language to model, simulate and program cyber-physical systems. In *Demo Session of 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016)*, 2016.

[6] N. Navet, L. Fejoz, L. Havet, and S. Altmeyer. Lean model-driven development through model-interpretation: the CPAL design flow. In *Embedded Real-Time Software and Systems (ERTSS2016)*, January 2016.

[7] N. Tankovic, D. Vukotic, and M. Zagar. Rethinking model driven development: analysis and opportunities. In *Information Technology Interfaces (ITI), Proceedings of the ITI 2012 34th International Conference on*, pages 505–510. IEEE, 2012.

Evaluation of Mixed-Criticality Scheduling Algorithms using a Fair Taskset Generator

Saravanan Ramanathan, Arvind Easwaran
Nanyang Technological University, Singapore
Email: saravana016@e.ntu.edu.sg, arvinde@ntu.edu.sg

***Abstract*—The problem of scheduling mixed-criticality (MC) task systems is known to be NP-Hard, and as a consequence the performance of MC scheduling algorithms is frequently assessed using experimental evaluations based on randomly generated tasksets. It is therefore important to have a thorough understanding of all the parameters that impact the algorithms and a taskset generation procedure that is fair with respect to those parameters. Although there are a few popular taskset generators, there is no evaluation of the fairness properties of those generators. In fact, there is no existing study on identifying all the parameters that are relevant in the evaluation of MC scheduling algorithms. We address this shortcoming in this paper, and present a set of essential fairness properties for MC taskset generators. We also develop a new taskset generator and show that it satisfies those fairness properties. Finally, we evaluate the performance of multi-core MC scheduling algorithms using the generator, and provide new insights on the performance of those algorithms with respect to several taskset parameters.**

I. Introduction

Mixed-Criticality (MC) scheduling has received a lot of attention in the real-time literature ever since Vestal proposed the MC task model [1]. This is mainly because of the practical relevance of MC systems in safety-critical industries such as avionics and automotive. There have been several studies, both on single- as well as multi-cores, focusing on the design of scheduling algorithms for the MC task model; see [2] for review. One way to evaluate algorithm performance is analytical, wherein metrics such as speed-up bound [3] are derived. For MC systems it has been shown that the scheduling problem is NP-Hard [4]. Consequently, the only known analytical performance results are in terms of speed-up bounds.

Another mechanism to evaluate algorithm performance is experimental, wherein a *taskset generator* is used to generate a variety of MC task systems, and the algorithms are evaluated by testing their schedulability on these task systems. The driving principle behind such experimental evaluation is that as long as the set of generated task systems is "fair", meaning not biased in terms of the parameters used to define the task system, the resulting comparisons provide a fair assessment on the relative performance of the algorithms. Although, unlike speed-up bounds, these evaluations do not provide any analytical guarantees, they are being increasingly used in the evaluation of MC algorithms [5], [6], [7], [8]. This trend is because of two factors: 1) For many algorithms such as those based on heuristics or non-trivial schedulability tests (e.g., those derived from demand bound functions), it is extremely hard, if not impossible, to derive these speed-up bounds. 2) For algorithms with known speed-up bounds such as EDF-VD, either the bounds are not very tight as in constrained-deadline task systems, or the bounds are not representative of the performance of the algorithm in practice. For example, although EDF-VD has an optimal speed-up bound of $4/3$ for dual-criticality implicit-deadline task systems [9], its performance is shown to be relatively poor in experimental evaluations [5]. Thus, in the absence of tight analytical results on the performance of MC algorithms, it is important to design taskset generators that enable a fair experimental evaluation.

There have been few studies on taskset generators for MC systems ([5], [6], [7], [8], [10], [11]). Although they present different taskset generation algorithms, there is no work which methodically considers all the parameters that impact the performance of MC algorithms. As a consequence, there is neither any clear understanding of what constitutes a fair MC taskset generator, nor is there any discussion in these studies on the fairness properties of the generators themselves. Note that, when compared to non-MC systems, a much larger number of parameters affect the performance of MC algorithms. This is because tasks in MC systems have additional parameters such as resource utilization values at different confidence levels, and further these parameters are known to have a significant impact on algorithm performance. In this paper, we address this challenging problem by first presenting the principles that govern the fairness of a MC taskset generator. We then present a novel taskset generation algorithm for MC systems and show that it satisfies these principles. Similar to the UUnifast algorithm for single-core non-MC systems [12] and MRandFixedSum algorithm for multi-core non-MC systems [13], we believe that the taskset generation algorithm presented here can be used to experimentally evaluate MC algorithms in a fair manner. Thus, the contributions of this paper can be summarized as follows:

- We present the fairness properties (Section III-A) for any MC taskset generator based on all the parameters that affect the performance of MC scheduling algorithms. We identify some new parameters that influence schedulability, which were not considered in the existing generators.
- We propose a MC taskset generator that generates tasksets satisfying the above fairness properties (Section III-B).
- We present extensive experimental evaluation for multi-core MC scheduling algorithms with the proposed taskset generator (Section IV).

II. System Model

In this section we define our system model. We restrict our model to a dual-criticality system (namely LO and HI).

Tasks: We consider a sporadic taskset τ, in which each MC task τ_i is characterized by a tuple $(T_i,\chi_i,\vec{C}_i,D_i)$, where

- $T_i \in \mathbb{R}^+$ is the minimum release separation time,
- $\chi_i \in \{LO, HI\}$ is the criticality level,
- $\vec{C}_i \in \mathbb{R}^+$ is the vector of Worst-Case Execution Time (WCET) values - one for each criticality level. C_i^L and C_i^H are the LO- and HI-criticality WCET values respectively; we assume $C_i^L \leq C_i^H$ and,
- $D_i \in \mathbb{R}^+$ is the relative deadline; for implicit deadlines $D_i = T_i$ and for constrained deadlines $D_i \leq T_i$.

Taskset: We consider a dual-criticality sporadic taskset τ with n tasks, where a task τ_i represents an infinite number of job releases. LO- and HI-criticality utilization of a task τ_i is defined as $u_i^L \stackrel{\text{def}}{=} C_i^L/T_i$ and $u_i^H \stackrel{\text{def}}{=} C_i^H/T_i$ respectively. System-level normalized utilizations are defined as $U_L^L \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau_L} u_i^L/m$, $U_H^L \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau_H} u_i^L/m$ and $U_H^H \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau_H} u_i^H/m$, where m is the number of cores.

MC Modes: The system is said to be in LO-criticality mode or LO-mode if all the tasks $\tau_i \in \tau$ signal completion before exceeding LO-WCET. The system is said to be in HI-criticality mode or HI-mode if any HI-task $\tau_i \in \tau_H$ executes beyond its LO-WCET and signals completion before exceeding its HI-WCET. Mode switch is defined as the change in criticality level of the system from LO to HI. All LO-tasks are immediately discarded by the system at mode switch. We focus on the above MC model because this is the standard model in many studies on MC scheduling.

III. Fairness and Taskset Generator

In this section we describe the fairness properties that are essential for any MC taskset generator. We also describe our new taskset generator *MC-FairGen*, and compare its fairness properties against several existing generators.

A. Essential Fairness Properties

The performance of MC scheduling algorithms depend on several taskset parameters. Among them, the most important include task periods and deadlines, proportion of LO- and HI-criticality tasks, maximum individual task utilization, and system utilization parameters $|U_H^H - (U_H^L + U_L^L)|$, $|U_H^L - U_L^L|$ and $|U_H^H - U_H^L|$. The minimum required number of tasks in the system is $m+1$. The total utilization ($U_B = max(U_H^H, U_H^L + U_L^L)$) must range across all possible values. Thus, the essential fairness properties can be summarized as follows.

1) *Period*: Task periods must be chosen from a wide range and should have an appropriate distribution that is not biased. One way to achieve this is by choosing periods using uniform or log-uniform distribution. It has been shown that fixed-priority algorithms perform well when periods are chosen using log-uniform distribution [13].
2) *Deadline*: Task deadlines, in the case of constrained-deadline tasksets, must also be drawn from an appropriate distribution that is not biased. For example, drawing deadline values from a uniform distribution between C_i^H (or C_i^L) and T_i is one way to achieve this.
3) *Criticality*: The percentage of HI-criticality tasks in a taskset must also have an appropriate unbiased distribution (e.g., uniform across the scale from 0 to 100). The performance of algorithms (such as *criticality-aware* partitioning [7]) tend to vary when there are very few LO- or HI-criticality tasks in a taskset. Therefore it is essential to consider the boundary cases for this parameter.
4) *Maximum Task Utilization*: Maximum individual task utilization, $max\{max_i(u_i^L), max_i(u_i^H)\}$, must be fairly distributed across the range $(0, 1]$.
5) *System Utilization*: The normalized utilizations of a MC taskset include U_H^H, U_H^L and U_L^L. The three important parameters related to these utilizations are *total utilization difference* ($|U_H^H - (U_H^L + U_L^L)|$), *LO-mode utilization difference* ($|U_H^L - U_L^L|$) and *HI-criticality utilization difference* ($|U_H^H - U_H^L|$). Most of the algorithms tend to perform relatively poorly as these parameters increase in value. It is therefore essential that these three parameters are fairly distributed across the range $[0, 1]$.
6) *Independence of Parameter Distributions*: To gain further insights into behaviour of MC algorithms, beyond what could be obtained from the overall schedulability evaluations, it is necessary to evaluate them against specific parameters independent of remaining parameters. Therefore, it is essential that for each parameter, the remaining parameters are fairly distributed across possible values.

In the past, studies have used other parameters such as 'criticality factor' ([14], [5], [15], [6], [8], [11]) and 'number of tasks' [14] in their generator. Criticality factor is defined as the ratio of HI-mode to LO-mode utilization of a HI-task. Varying the criticality factor indirectly impacts the *HI-criticality utilization difference* ($|U_H^H - U_H^L|$). This variation in utilization difference is captured by the *system utilization* property. Further, by fixing the criticality factor, the maximum task utilization $max(u_i^H)$ is restricted as a function of u_i^L. It is therefore reasonable to choose system utilization parameters rather than criticality factor. Whereas, varying the number of tasks impacts the individual utilization of tasks. This parameter is captured by the *maximum task utilization* property.

Extension to Multi-Criticality: These properties can also be generalized to multi-criticality systems. The *period* and *deadline* properties remain the same. Extending *criticality* property to multi-criticality requires all possible values of task criticality to be considered for each criticality level. Extending the *maximum task utilization* property is quite straightforward; it needs to consider task utilizations across all the criticality levels. To extend the *system utilization* property one needs to consider all combinations of system utilization differences. For example, the property on *total utilization difference* ($|U_H^H - (U_H^L + U_L^L)|$) needs to be expanded to consider the utilization difference between all pairs of adjacent criticality levels.

Algorithm 1 MC-FairGen

Input: m,u_{min},u_{max}
Output: Taskset τ

```
1: for U_H^H ∈ [0.1, 0.2, ..., 1.0] do
2:     for U_H^L ∈ [0.05, 0.15, ..., U_H^H] do
3:         for U_L^L ∈ [0.05, 0.15, ..., 1 − U_H^L] do
4:             for P_H ∈ [0.1, 0.2, ..., 0.9] do
5:                 Minimum required total HI-tasks, N_min^H = ⌈(U_H^H ∗ m/u_max)⌉
6:                 Minimum required total LO-tasks, N_min^L = ⌈(U_L^L ∗ m/u_max)⌉
7:                 Minimum required total tasks, N_min = max(m + 1, ⌈(N_min^H/P_H)⌉, ⌈(N_min^L/(1 − P_H))⌉)
8:                 Total tasks, N = uniform[N_min, 10 ∗ m]
9:                 Total HI-tasks, N_H = max((P_H ∗ N), N_min^H)
10:                Total LO-tasks, N_L = N − N_H
11:                ∀i ∈ N, the period T_i = uniform[5, 100]
12:                HI-task HI-utilizations {u_i^H} = MRandFixedSum(U_H^H ∗ m, N_H, u_min, u_max)
13:                HI-task LO-utilizations {u_i^L} = BoundedUniform(U_H^L, U_H^H, m, N_H, u_min, {u_i^H})
14:                LO-task utilizations {u_i^L} = MRandFixedSum(U_L^L ∗ m, N_L, u_min, u_max)
15:                ∀i ∈ N, the execution requirement C_i^L = u_i^L ∗ T_i
16:                ∀i ∈ N_H, the execution requirement C_i^H = u_i^H ∗ T_i
17:                ∀i ∈ N_L, the relative deadline D_i = uniform[C_i^L, T_i]
18:                ∀i ∈ N_H, the relative deadline D_i = uniform[C_i^H, T_i]
19:            end for
20:        end for
21:    end for
22: end for
```

B. MC-FairGen Taskset Generator

The taskset parameters considered in our generator are described as follows:

- Minimum and maximum individual task utilization u_{min} ($= 0.0001$) and u_{max} ($= 0.99$). u_{min} is required to guarantee all possible values for the percentage of HI-criticality tasks in a taskset. u_{max} is required to ensure a reasonable execution time for many schedulability tests, particularly those based on demand bound functions.
- $m \in \{2, 8\}$ denotes the total number of cores.

MC-FairGen is described in Algorithm 1. The minimum required total HI-tasks(N^H_{min}) and total LO-tasks(N^L_{min}) in the system is given by Steps 5 and 6 in Algorithm 1. The ceiling of utilization bound ensures individual task utilization to be ≤ 1. The division by u_{max} allows task utilizations to be bounded by u_{max}. The minimum required total tasks in the system N_{min} is given by Step 7, which ensures the percentage of HI-criticality tasks P_H. Further, it lower bounds the number of tasks in the system by $m+1$. The total number of tasks in the system is then drawn uniformly at random from $[N_{min}, 10 * m]$. The upper bound($10 * m$) on the number of tasks in the system is to allow for all possible values of P_H. HI-task HI-utilizations $\{u^H_i\}$ and LO-task utilizations $\{u^L_i\}$ are obtained using *MRandFixedSum* algorithm [13].

HI-task LO-utilizations $\{u^L_i\}$ are obtained using *BoundedUniform* shown in Algorithm 2. *BoundedUniform* sorts the $\{u^H_i\}$ values in descending order, and for each u^H_i it assigns u^L_i subject to two conditions: (1) sum of the total allocated u^L_i and total minimum remaining u^L_i do not exceed the utilization bound($m * U^L_H$) and (2) each $u^L_i \leq u^H_i$.

Algorithm 2 BoundedUniform

Input: $U^L_H, U^H_H, m, N_H, u_{min}, \{u^H_i\}$
Output: $\{u^L_i\}$

```
1: Sort {u_i^H} in decreasing order
2: U_L^rem = U_H^L ∗ m
3: U_H^rem = U_H^H ∗ m
4: N_H^rem = N_H − 1
5: for u_i ∈ {u_i^H} do
6:     U_H^rem = U_H^rem − u_i
7:     u_i^L = uniform(max(u_min, U_L^rem − U_H^rem), min((U_L^rem − (N_H^rem ∗ u_min)), u_i))
8:     N_H^rem = N_H^rem − 1
9:     U_L^rem = U_L^rem − u_i^L
10: end for
```

C. Fairness Properties of MC-FairGen

1) *Period, Deadline and Criticality*: MC-FairGen explicitly considers these fairness properties in the taskset generation process. All the three parameters are drawn from uniform distribution.
2) *Maximum Task Utilization*: Given system utilization values and number of tasks, *MRandFixedSum* draws task utilization values uniformly from the given range [13]. Since we consider all possible combinations of system

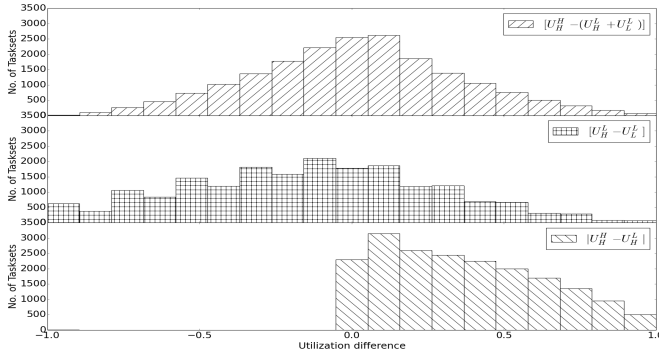
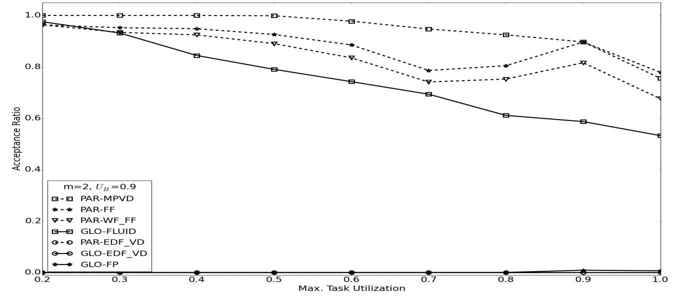
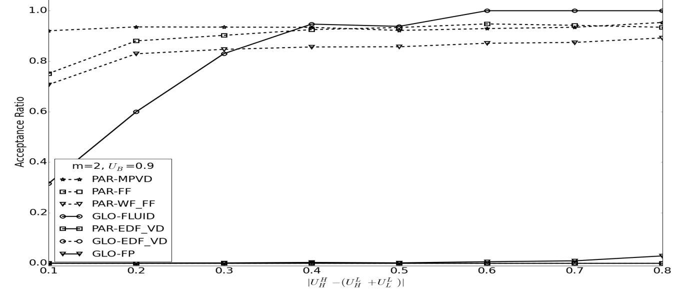


Fig. 1: Utilization distribution of MC-FairGen



(a) Maximum Individual Task Utilization



(b) Total Utilization Difference ($|U_H^H - (U_H^L + U_L^L)|$)

Fig. 2: Varying taskset parameters

utilization values, the resulting tasksets have a folded normal distribution for $max\{max_i(u_i^L), max_i(u_i^H)\}$ with mean (ν) 1.0 and standard deviation (σ) 0.46. We propose classifying the tasksets into two equal-sized classes based on the value of maximum task utilization. That is, tasksets with maximum task utilization no more than $\nu(1 - 3\sigma/4) = 0.655$ would be categorized into the "small" class, and those with value greater than this bound would be categorized into the "large" class. Figure 2a shows the performance of MC algorithms for values of maximum task utilization. We can observe that the variation in performance is not uniform across the parameter values; the performance drop is significant for larger values when compared to smaller values. This is consistent with the classification presented above; algorithm performance is more or less stable when maximum task utilization values are in the small class, and only decrease when these values are in the large class. Hence, based on this classification, we can claim that MC-FairGen satisfies the fairness property for this parameter.

3) *System Utilization*: Figure 1 shows the distribution of three system utilization differences for our generator. It can be seen that all the distributions are either normal or folded normal, as in the case of maximum task utilization. Therefore, we again classify each of these parameters into two classes, "small" and "large", as above. The cut-off values for this classification are as follows: 0.2 for $|U_H^H - (U_H^L + U_L^L)|$, 0.2 for $|U_H^L - U_L^L|$ and 0.35 for $|U_H^H - U_H^L|$. To verify that this classification is reasonable in terms of ensuring fairness with respect to the existing algorithms, we present the variation in schedulability as a function of $|U_H^H - (U_H^L + U_L^L)|$ in Figure 2b. As can be observed, the variation in performance is significant when the parameter value is less than 0.2, and minimal when the parameter value is greater than 0.2. That is, the variation *is not uniform* across this parameter. Although we do not present figures for the other two parameters due to lack of space, similar results have been observed. Thus, based on this classification of the system utilization parameters, we can claim that MC-FairGen satisfies the corresponding fairness property.

4) *Independence of Parameter Distributions*: To evaluate the schedulability performance of an algorithm against a particular parameter, it is necessary to negate the impact of all the other parameters. Let us consider the parameter $|U_H^H - (U_H^L + U_L^L)|$ whose metrics are shown in Table I. 50.41% of the tasksets are in the small class and 49.59% of the tasksets are in the large class with respect to this parameter. In each of these two classes, the distribution of tasksets for the remaining parameters are also well distributed. Note that for task periods, deadlines and criticality distribution, since we choose them independently using uniform distribution, they would also be fairly distributed in these two classes. Similar metrics have been observed for the remaining parameters as well, but we do not present them here for brevity. Thus, we can conclude that MC-FairGen also satisfies the *independence of parameter distribution* property.

Discussion on uniform distribution: In MC systems, it is extremely challenging, if not impossible, to have a uniform distribution across system utilization and maximum task utilization parameters. This is due to the constraints between the parameters.

The three system utilizations U_H^H, U_H^L and U_L^L characterize a MC system. Any valid MC system should satisfy the following two conditions: 1) $U_H^H \geq U_H^L$ and 2) $U_H^L + U_L^L \leq 1$. Satisfying the above two conditions restricts the range of values for some utilizations. Given an U_H^H value, U_H^L is bounded by U_H^H, and given an U_H^L value, U_L^L is bounded by 1-U_H^L. Say, we want to have a uniform distribution for the $|U_H^H - (U_H^L + U_L^L)|$ parameter. Lets fix U_H^H for a given $|U_H^H - (U_H^L + U_L^L)|$ value.

TABLE I: Total System Utilization Difference $|U_H^H - (U_H^L + U_L^L)|$

Parameter	Classification	% of Tasksets	Classification				Classification	
			Small	**Large**	**Small**	**Large**	**Small**	**Large**
			$\|U_H^L - U_L^L\|$		$\|U_H^H - U_H^L\|$		$max(max(u_i^H), max(max_i^L))$	
$\|U_H^H - (U_H^L + U_L^L)\|$	**Small**	50.41	43.65	56.35	51.46	48.54	51.15	48.85
	Large	49.59	45.36	54.64	43.10	56.90	48.51	51.49

Then we have two choices when picking $(U_H^L + U_L^L)$. Picking U_H^L or U_L^L decides the other parameter. One thing to consider here is that U_H^L value is restricted by U_H^H. This in turn restricts the U_L^L value, thereby affecting the distribution of the other two parameters $|U_H^L - U_L^L|$ and $|U_H^H - U_H^L|$. Therefore, it is reasonable to consider a normal distribution for the parameters rather than a uniform distribution, particularly given the performance variation of existing scheduling algorithms.

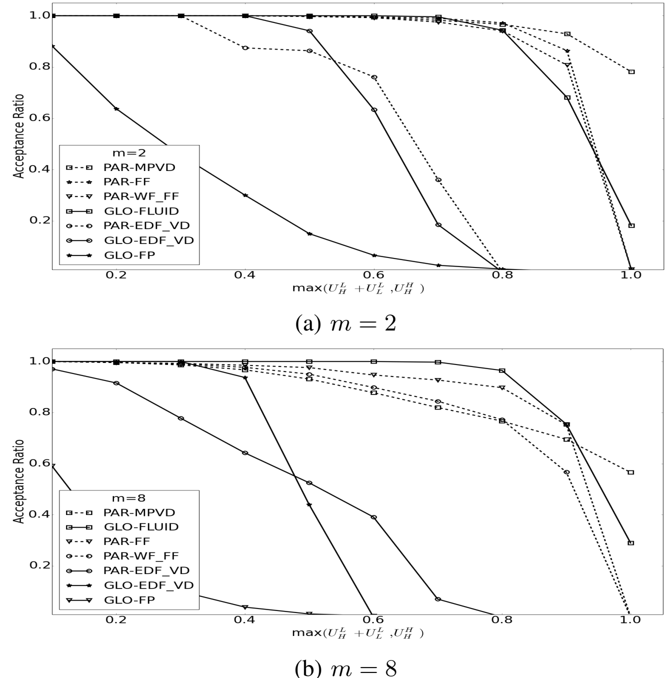
D. Comparison of Existing Generators

In this section we evaluate the existing MC taskset generators in terms of the fairness properties presented in Section III-A. We classify the existing set of generators into two major categories. The group of generators that consider the same utilization bound for both LO- and HI-mode utilizations ($U_H^L + U_L^L$ and U_H^H) fall under the first category. The group of generators that consider independent utilization bounds for LO- and HI-mode fall in the second category (denoted as class D). We further classify the first category into three classes (denoted as A,B and C) based on their taskset properties.

All the existing generators consider the *period* and *deadline* property. Class A generators [8] do not consider the *maximum task utilization* property. They have the property that all the generated tasksets are confined to small system utilization values. Like Class A, Class B generators ([14], [5], [6], [15], [11]) also do not satisfy the *maximum task utilization* property. Unlike Class A however, the generated tasksets of these class of generators are not confined to small system utilization values. Class C generators ([10], [16]) consider all the fairness properties except the *system utilization* properties through a set of different experiments. However, these class of generators have a high taskset discard ratio when the utilization bounds are small. Class D [7] is a reasonable generator for MC systems because it considers independent utilization bounds for LO- and HI-mode utilization. It however considers a fixed number of HI-criticality tasks in the generation process, and hence does not satisfy the *criticality* property. None of the above generators satisfy the *system utilization* properties. Thus, it is reasonable to conclude that none of the existing generators adhere to all the fairness properties listed in Section III-A.

IV. Experiments and Results

In this section we evaluate the schedulability performance of multi-core MC scheduling algorithms using MC-FairGen. These include global fpEDF [16], partitioned EDF_VD [16], global fixed-priority [15], global fluid [10], an extension of GREEDY [5] with first-fit packing strategy [7], another



(a) $m = 2$

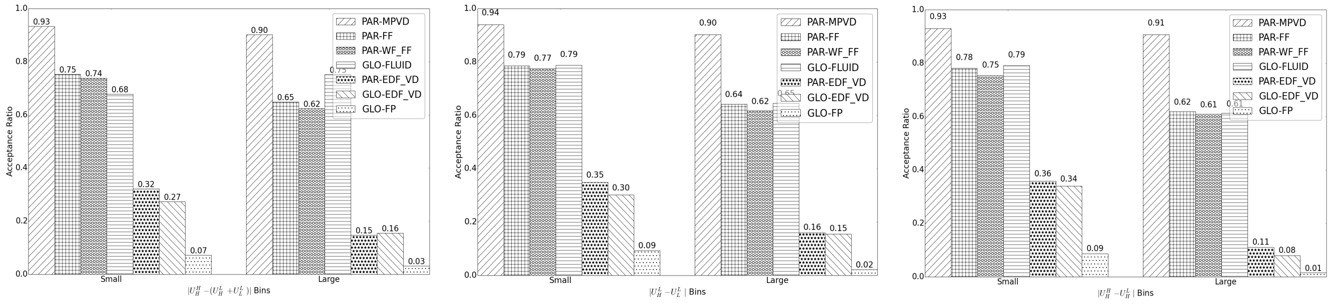
(b) $m = 8$

Fig. 3: Overall Schedulability (implicit,uniform)

extension of GREEDY with both worst-fit and first-fit strategy [7] and MPVD with heavy low-critical task aware allocation [8] represented as GLO-EDF_VD, PAR-EDF_VD, GLO-FP, GLO-FLUID, PAR-FF, PAR-WF_FF and PAR-MPVD respectively. All the results presented are for implicit deadline task systems with uniform distribution of task periods and deadlines. Similar results were obtained for constrained deadlines. We also evaluated for log-uniform distribution of periods and deadlines, and found that there was not much variation in the performance of the algorithms except for FP scheduling. We therefore do not present them here for brevity.

In Figure 3 we present the overall schedulability of the algorithms. We plot the acceptance ratios of the algorithms i.e., fraction of schedulable tasksets, versus total utilization U_B varying over $m \in \{2, 8\}$. Each data point corresponds to at least 5000 tasksets. For $m = 2$, the partitioned demand bound function (DBF) based tests perform better than the other algorithms as shown in Figure 3a. The results obtained are consistent with the results from previous studies except for the partitioned algorithms [10].

When normalized utilization nears 1.0, the performance of PAR-FF and PAR-WF_FF algorithms drop significantly compared to PAR-MPVD. The reason is that the partitioning



(a) Total Utilization $|U_H^H - (U_H^L + U_L^L)|$ (b) LO-mode Utilization $|U_H^L - U_L^L|$ $(m = 2)$ (c) HI-Crit. Utilization $|U_H^H - U_H^L|$ $(m = 2)$

Fig. 4: System Utilization Difference Distribution (implicit,uniform)

heuristics of these algorithms fail to successfully allocate the tasks. In case of PAR-FF, the tasks are allocated using first-fit strategy independent of its criticality. The problem with the first-fit is that the task utilizations are not balanced among the cores. In case of PAR-WF_FF, the HI-criticality tasks are allocated first using worst-fit approach and then the LO-criticality tasks are allocated using first-fit approach. The problem with this approach is that when there are heavy low-critical tasks in the system, it fails to get allocated to the core. Whereas, in case of PAR-MPVD, due to heavy LO-critical task aware partitioning and WF_FF bin packing approach, it performs well.

The performance of PAR-MPVD shown here is contradicting to the one presented in [11]. PAR-MPVD is known to perform better when there are heavy LO-critical tasks. As the generator in [11] generates tasksets only in low utilization ranges, it negatively affects the performance of PAR-MPVD.

To provide further insights on how algorithms perform with respect to specific parameters, we also present the performance results varying individual parameters. For brevity, we only present the schedulability results based on varying system utilization parameters i.e., $|U_H^H - (U_H^L + U_L^L)|$, $|U_H^L - U_L^L|$ and $|U_H^H - U_H^L|$ in Figure 4. All the algorithms perform well in the first class, where the three parameter values are small, and perform poorly when the values become large. GLO-FLUID algorithm performs well when $|U_H^H - (U_H^L + U_L^L)|$ is large as it mainly optimizes HI-mode execution, and performs poorly when $|U_H^L - U_L^L|$ or $|U_H^L - U_L^L|$ becomes large. All DBF based tests have more impact on $|U_H^L - U_L^L|$ and $|U_H^H - U_H^L|$ parameters when compared to $|U_H^H - (U_H^L + U_L^L)|$.

V. Summary

Taskset generators are an important tool in the evaluation of MC scheduling algorithms, mainly due to the hardness of these algorithms and the lack of quantifiable metrics such as speed-up bounds. In this paper we identified the factors that affect the schedulability of MC scheduling algorithms and presented the fairness properties that govern any MC taskset generator. We also proposed a new generator called MC-FairGen capable of generating tasksets that satisfy the fairness properties. We evaluated the performance of multi-core MC algorithms using the proposed generator. These evaluations have provided some new insights on how individual taskset parameters affect the existing algorithms, and could be used to develop improved algorithms in the future.

References

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium (RTSS), 28th IEEE International*, Dec 2007.

[2] A. Burns and R. I. Davis. (2013) Mixed Criticality Systems - A Review. http://www-users.cs.york.ac.uk/burns/review.pdf.

[3] B. Kalyanasundaram and K. Pruhs, "Speed is as powerful as clairvoyance [scheduling problems]," in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, Oct 1995.

[4] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *Computers, IEEE Transactions on*, vol. 61, no. 8, Aug 2012.

[5] P. Ekberg and W. Yi, "Bounding and shaping the demand of mixed-criticality sporadic tasks," in *Real-Time Systems (ECRTS), 24th Euromicro Conference on*, July 2012.

[6] A. Easwaran, "Demand-based scheduling of mixed-criticality sporadic tasks on one processor," in *Real-Time Systems Symposium (RTSS), 34th IEEE International*, Dec 2013.

[7] P. Rodriguez, L. George, Y. Abdeddaim, and J. Goossens, "Multi-criteria evaluation of partitioned edf-vd for mixed-criticality systems upon identical processors," in *Workshop on Mixed Criticality Systems (WMC)*, December 2013.

[8] C. Gu, N. Guan, Q. Deng, and W. Yi, "Partitioned mixed-criticality scheduling on multiprocessor platforms," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2014.

[9] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *Real-Time Systems (ECRTS), 24th Euromicro Conference on*, July 2012.

[10] J. Lee, K.-M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee, "MC-Fluid: Fluid Model-Based Mixed-Criticality Scheduling on Multiprocessors," in *Real-Time Systems Symposium (RTSS), 35th IEEE International*, Dec 2014.

[11] J. Ren and L. T. X. Phan, "Mixed-criticality scheduling on multiprocessors using task grouping," in *Real-Time Systems (ECRTS), 27th Euromicro Conference on*, July 2015.

[12] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, may 2005.

[13] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, July 2010.

[14] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *Real-Time Systems Symposium (RTSS), 32nd IEEE International*, Nov 2011.

[15] R. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *Real-Time Systems (ECRTS), 24th Euromicro Conference on*, July 2012.

[16] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, vol. 50, no. 1, 2014.

Dynamic criticality management with ARTEMIS

Olivier CROS
LIGM / Université Paris-Est
Bat Copernic - 5, bd Descartes
77454 Champs sur Marne, France
olivier.cros0@gmail.com

Geoffrey EHRMANN
LACSC
37 quai de Grenelle
75015 Paris, France
geoffrey.ehrmann@gmail.com

Laurent GEORGE
LIGM / Université Paris-Est, ESIEE
2 Boulevard Blaise Pascal
93162 Noisy-le-Grand, France
lgeorge@ieee.org

***Abstract*—In this work, we propose to detail the mixed-criticality integration inside our network simulator ARTEMIS. The objective here is to propose a solution to manage and simulate concrete criticality level changes inside network infrastructures, in order to focus on a network topology reconfiguration w.r.t to critical and non-critical messages evolutions. Through a transmission time computation model based on a probabilistic approach, we propose a solution to generate flowsets integrating mixed-criticality, in order to simulate the scheduling of these flowsets through different topologies.**

I. Introduction

A. About real-time simulation

In strongly constrained industrial domains like spacecraft, public transports or aircraft, reliability and performances are two fundamental objectives which imply defining strict timeliness constraints to prevent system failures. It seems obvious for everyone to imagine the huge human disaster represented by an aircraft network crash at landing, not to mention the financial impact of such events.

As a matter of fact, new protocols and architectures in real-time networks must be certified before being deployed on industrial structures. These protocols have to be analyzed, verified and tested to be proved reliable and safe to be implemented. But operating the tests directly on real physical systems can appear to be very costly. As a conclusion, these real tests should done when most of the protocol has already been validated. That is why, in order to prepare and run performances and reliability tests replacing some of the physical tests, we need to define simulation tools.

For the real-time network simulation, we propose a dedicated network simulator. This simulator is called Another Real-Time Engine for Message-Issued Simulation (ARTEMIS).

B. Related work

ARTEMIS has already been presented in [1], [2] as an open-source user-oriented real-time network simulation tool. Its architecture was similar to real-time multicores and multiprocessors schedulers architectures like Cheddar [3] (a modular framework for schedulability analysis), SchedMCore [4] (toolbox for multicore simulation), and SimSo [5] (an open-source tool designed for multiprocessor context).

There also exists different network simulators, which are more oriented to industrial context. We can mention NS [6] for global network simulation or OmNET++ [7] for dimensioning and performances purposes. Concerning Real-Time (RT) networks architectures, there also exists different simulators to observe and manage specific network architectures : CANoe [8] for Controller Area Network (CAN) or the work presented in [9] for Avionics Full DupleX switched ethernet (AFDX).

ARTEMIS is a RT network simulation tool, providing schedulability analysis for network topologies. Based on a generic component-oriented model (see [1]), the purpose of ARTEMIS is to propose the integration of mixed criticality constraints inside network topologies. This integration was partially detailed in [2] but the mixed criticality management model presented was rather incomplete. In this work, we propose to detail a more dynamic and configurable mixed criticality model integration inside ARTEMIS, and we detail the technical solutions we have done to represent and manage mixed criticality inside a simulation environment.

C. Contributions

The architecture of ARTEMIS (described in [10]), is organized around a set of external modules (grapher, generators) based on a scheduling simulation core. Based on the work presented in [10], we integrated in ARTEMIS core two main models for mixed criticality management. First, we designed centralized and decentralized criticality management to store and share the criticality level among all the nodes of a topology. Then, in order for the core to act independently and to simulate criticality change events scenarios (not just depending on user actions), we designed a new message generation model inside the core. That is what we detail below in III and IV.

The integration of these new protocols implied to change a part of the generation and scheduling model of ARTEMIS, dedicated to the criticality management inside each node. The modular architecture of ARTEMIS allowed us to design a dedicated part for criticality management, without requiring to modify the input or output data formalization. We also reinforced the design and conception fundamentals by improving the Graphical User Interface (GUI) for a better user experience. We added new functionalities for web-oriented and distributed context, to make ARTEMIS a sharable tool designed to be installed on public web servers. We detail this in II.

We propose to test different potential schedulability analysis results and to evaluate the impact of mixed criticality integration inside different network topologies. This is showed through different simulations, detailed in V. We now describe the different improvements inside ARTEMIS global architecture.

II. Web-oriented architecture

A. Global architecture

ARTEMIS' Graphical User Interface (GUI) is the link between users and simulator's kernel. It has to interpret messages between users and kernel in order to make possible the communication between these two entities. To build a simulation, users must configure a network through the GUI, then the interface sends data to the kernel as XML files. The kernel runs the simulation and returns XML files containing simulation results. Once XML files are parsed, GUI displays results as XML logs or graphs.

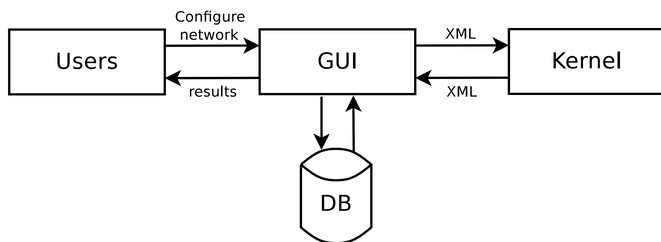


Fig. 1. ARTEMIS architecture

To configure a simulation, users have first to create a topology by using the topology generator or create it manually. Then users have to define all the components of this network, namely the nodes and the links. A node is defined by a name, a scheduling policy, an automatic generated network address and a transmission rate.

Once the topology is created, users can create messages which will define the network behaviour. Users can create a new message manually by configuring the message path, the Worst-Case Transmission Time (WCTT), the period and the offset. The messages creation is more detailed in [2].

Each parameter related to a simulation is saved in a MySQL database. When users click on "Run", GUI generates the XML files after getting data from the database. There are 4 XML files to run a simulation:

- "network.xml" that contains the network topology. It contains a list of nodes with all their attributes (ID and name), and the other nodes to which they are linked to.
- "config.xml" contains the whole configuration of the network, namely the simulation time, the latency and the mixed-criticality management model.
- "graphconfig.xml" contains the name of the graph and the parameters related to graph management.
- "messages.xml" contains the messages to be sent. It lists all the messages and their attributes that we defined previously.

These files are sent to the kernel, which will perform the simulation before returning XML files with graph results, GUI displays these results as a scheduling graph.

B. Web distribution

ARTEMIS is a web-oriented tool. This choice has been made in order to make it easy to install and to use. Using web interface makes it independent from any operating system, which allows us to spread the tool to a large public. The main purpose of the GUI is to be as intuitive as possible. As a matter of fact, ARTEMIS is designed for everyone, which includes non-developers or students who need to be guided, so interface has been designed to be ergonomic, fluid and clear for users; web programming allows a fluid and clear utilization of the tool. Thanks to AJAX architecture, the system answers quickly to users commands, which make the navigation comfortable, and the CSS language enables us to make a clean and sleek visual.

C. Exporting results

In order to improve user experience of the tool, ARTEMIS now integrates a simulation manager. It adds to ARTEMIS a bunch of new functions to manage simulations. Each user can now create its own simulations, export or import them to different platforms in order to increase the reusability of the different simulation configurations built.

The export function produces a ZIP archive containing the input XML files required to build the simulation. Every archive exported by ARTEMIS can also be imported. Importing a simulation triggers the creation of a new simulation and the automatic configuration of it, by using data from the selected ZIP archive. All informations are saved in the database. Then the simulation is ready to be run.

These functions are essential for ARTEMIS. It makes the tool portable and user-oriented by allowing sharing and communication between users and simulation contexts. These new functions allows a user to create dedicated topologies of variable sizes and to propose different messages sets configurations in order to operate benchmarking and performances comparisons on different contexts.

The simulation identification in ARTEMIS is based on session identification and unique identifier association for each simulation configuration : each simulation is unique, and belongs to a specific user. It allows us to improve the portability of ARTEMIS architecture, specially in contexts designed for multi-user utilization, which were the fundamental goal of the web architecture of ARTEMIS.

III. Flowset generator

In order to be able to simulate concrete network scheduling scenarios through ARTEMIS, we define a flowset generator connected to the kernel. Currently, RT simulators propose tasksets generators based on the UUnifast algorithm [11] to build the different tasksets needed for scheduling analyses.

In our work, we adapted current taskset generation algorithms to network context. The purpose of ARTEMIS is to propose scheduling scenarios integrating mixed-criticality in real-time networks. We adapted the current models to generate flowsets mixing messages of different criticality levels. That is the point we propose to detail in the following section.

A. UUnifast for network context

Basically, UUnifast [11] is a taskset generation algorithm. Its purpose is to generate a set of n periodic or sporadic tasks, associated to a global load l. Each task of the generated set is characterized by two properties : a Worst-Case Execution Time (WCET), and a period (or minimum inter-arrival time, in the case of a sporadic task). For each task τ_i, we note C_i its WCET and T_i its period. We define the maximum duration T_{max} of the period, based on the duration of the simulation. The generation process is based on 4 different steps :

- First, we generate a random value r_i, based on a uniform law $\mathcal{U}$, with the following expression :

  $$r_i = \mathcal{U}(log(T_{min}), log(T_{max} + T_g))$$

  We assume that all the generated values of r_i are in the interval $[log(T_{min}), log(T_g+T_{max})]$. More details on the computation of r_i were given in [11].
- Then, we compute the period T_i of the task τ_i. This value of T_i is indexed on r_i, according to the expression :

  $$T_i = \left\lfloor \frac{e^{r_i}}{T_g} \right\rfloor * T_g$$

  This bounds the generated period with the value T_{max}. This expression is based on the time granularity of the system, noted as T_g.
- Next, based on a uniform law $\mathcal{U}$ of support 0 and 1, we generate a random utilization u_i for the task. This utilization represents the individual load of the task. It is computed by the cumulative activation of jobs from the task during the time interval $[0; T_{max}]$.
- Finally, we compute the WCET C_i of the task, by computing $C_i = u_i * T_i$.

At the end of the generation of each task, we compare the value of the targetted load l and the value of cumulative utilizations $u = \sum_{k=1}^{n}(u_i)$. If we have $u = l$, the taskset is characterized as correct. In the other case, we discard the taskset and generate a new one.

The problem represented by this method concerns the discarding process. As a matter of fact, this discarding process tends to increase the number of generation loops to run and, as a result, to increase the generation time needed by the algorithm. Inside ARTEMIS, we propose to modify this process in order to reduce the number of discarded tasksets.

Discarding a taskset comes from the point that the cumulated utilization u tends to exceed or lower the value of l. The computation of each value of u_i is based on a uniform law $\mathcal{U}(0, 1)$. As a matter of fact, the generation process tends to generate flowsets with cumulated utilizations which are out of bounds, implying to discard the generated flowset.

The solution we propose was to bound the uniform law $\mathcal{U}$ in order for the global generated utilization u to be centered around the value of l.

As we target a global load value of l for a flowset of size n, we generate utilizations which have an average load equal to $\frac{l}{n}$. Thus, the law $\mathcal{U}$ is characterized by a specific variance v which can be modified to adjust the results of the generating algorithm. The higher the value of v, the more heterogeneous the generated flowset in terms of utilizations, but the more we tend to increase the number of discarded tasksets. We can vary the value of v depending on the accuracy and heterogeneity we target in the generated taskset. For basic simulations, v is set between 0.05 and 0.06 in ARTEMIS core.

B. Load computation

In multicore and multiprocessor RT scheduling analyzers, the generation of a taskset is based on a targeted global utilization represented by the taskset. Depending on the network architecture, a generated taskset utilization can exceed 1 but the utilization on each node is less than 1 .

In network context, this global utilization has been replaced by the global load l. One naive approach would be to establish a strict parallel between the global utilization of a taskset and the global load l. In fact, as there is not message transmission or paths computation in processor context, the individual utilization $\frac{C_i}{T_i}$ of a task τ_i represents its direct impact on the system in terms of utilization. On the opposite, the individual load $\frac{C_i}{T_i}$ of a flow v_i in a network is not its direct impact : as each message from the flow will be transmitted once in each switch of its path, the real impact of one flow in terms of traffic depends on its path.

C. Mixed-criticality integration

The presented flowset generator generates flows of different criticality levels. It implies defining two different constraints: first, we have to clearly define a protocol to decide which message belongs to which criticality level. Then, for each message, we have to precise the WCTT of the message for each criticality level it belongs to. We detail these two steps below.

In order to decide which message belongs to which criticality level, we first based our work on the assumption made in [12] for the WCTT of a message sent in different criticality modes. If we suppose k different criticality levels $\gamma_1, ..., \gamma_{k-1}, \gamma_k$, we assume for n flows that :

$$\forall i \in [1; n], \forall q, r \in [1; k], q < r \implies C_i^{\gamma_q} \leq C_i^{\gamma_r} \quad (1)$$

This hypothesis corresponds to the case where increasing the criticality level of a flow leads to send more information. If the maximum criticality level of a flow v_i is γ_q, all the WCTT of v_i will be lower or equal to $C_i^{\gamma_q}$. Given this hypothesis on the different criticality levels of a network, we defined a criticality rate C_{rate} in the network. For each flow generation, we compute a probability p_{rate} included in $[0; 1]$. Once computed, for each criticality level γ_q, we check if $p_{rate} < (C_{rate})^{q-1}$. If that is the case, we generate a dedicated value for $C_i^{\gamma_q}$. If not, we set $C_i^{\gamma_q} = -1$. This hierarchical structure is convenient to mixed-criticality models as it was presented in [13].

IV. Mixed-criticality models

A. Transmission time computation models

Integrating mixed-criticality in ARTEMIS means that the different virtualized topologies created through the tool must be able to manage criticality levels and criticality level switches. This implies to be able to trigger specific events occuring in a criticality level switch. According to previous works on mixed-criticality in networks [10], we assume that two different events can trigger a criticality switch in ARTEMIS :

- Either the user statically designed a criticality switch at a specific time. In that case, the user specifies the level to switch to and the time at which it occurs. It is called the static model.
- Either a message exceeds its WCTT at a specific level according to a configurable law. If we suppose a flow v_i composed of two WCTT C_i^{LO}, C_i^{HI} for two Low (LO) and High (HI) criticality levels, this event corresponds to a time where a message from v_i exceeded the WCTT C_i^{LO}. In that case, necessarily, it implies that v_i has to be considered as occuring a criticality switch to HI

The first model was introduced in ARTEMIS and has already been discussed in [2] . In order for ARTEMIS core to be able to manage the second case (low-critical level WCTT exceeding), it means that the messages generator model has to be able to generate messages exceeding their low-critical WCTT. We propose to detail here the modifications we add to integrate inside ARTEMIS message generator in order to take into account this new generation model.

Inside ARTEMIS core, we defined several $\gamma_1, ..., \gamma_{k-1}, \gamma_k$ criticality levels (minimum 1). Each flow v_i is designed with a specific WCTT $C_i^{\gamma_j}$ for each criticality level j. In the case where the flow does not belong to any criticality level except the lowest one γ_1 (non-critical level), we note $\forall j > 1, C_i^{\gamma_j} = -1$. As a matter of fact, each flow v_i is defined with a set of WCTT noted as $C_i^{\gamma_1}, ..., C_i^{\gamma_{k-1}}, C_i^{\gamma_k}$.

In order to generate potential criticality switch triggering events, generating a message from a flow v_i implies to generate not only a specific message transmission time between the message Best Transmission Time (BTT) and $C_i^{\gamma_1}$, but a message transmission time which is included between the message BTT and its highest WCTT (attached to the highest criticality level the flow v_i belongs to). In order to integrate this mixed criticality model, we integrated inside ARTEMIS different probabilistic models to generate messages of different transmission times, each transmission time associated to a specific criticality level.

We have to keep in mind that basically, ARTEMIS has been designed for worst case analysis. Hence, for each generated transmission time of a message, we round it to the closest highest corresponding WCTT w.r.t. a criticality level. This model allows us to keep a worst-case evaluation of delays in the end-to-end transmission delay computation of the different flows in the network.

1) Linear model: The linear model proposes to generate a transmission time which value is based on a linear probability law. The generated time is computed from two bounds : the flow BTT (noted as B_i) and the message highest WCTT, belonging to a criticality level γ_j (notes as $C_i^{\gamma_j}$). The probability of generating a specific transmission time t is estimated as follows :

$$\mathcal{P}(t) = \begin{cases} 0 \quad \text{if } t \leq B_i \\ \frac{T_g}{C_i^{\gamma_j} - B_i} \quad \text{if } B_i \leq t \leq C_i^{\gamma_j} \\ 0 \quad \text{if } t > C_i^{\gamma_j} \end{cases}$$

2) Strict model: The strict model is based on the assumption that a message transmission time is necessarily equal to one of its WCTT. As a matter of fact, the strict model consists in picking one transmission time among all potential values in $C_i^1, ..., C_i^{j-1}, C_i^j$. ,

If we suppose that the flow v_i belongs to $\gamma_1, ..., \gamma_{j-1}, \gamma_j$ criticality levels, we can express its probability model as :

$$\mathcal{P}(t) = \begin{cases} \frac{1}{j} \quad \text{if } t = C_i^u, u \in [1; j] \\ 0 \quad \text{if } not \end{cases}$$

3) Gaussian-based models: In this model, we define a uniform law $\mathcal{U}$ which is used as a base to compute each transmission time of each message. This uniform law is defined by two parameters : its center c and its deviation d. We note the expression as $\mathcal{U}(c, d)$. In order to define different transmission time computation models, we can adjust both values of c and d. Their role is described as follows :

- The lower the value c, the higher the probability for the transmission time of a message from flow v_i to be equal or close to its BTT. On the contrary, the higher the value of c, the higher the probability to have a generated transmission time close to the highest WCTT of v_i.
- The deviation is used to define the probability of a generated transmission time to be far from c. The higher the deviation, the more heterogeneous the successive generated transmission times.

A complete basic uniform law can generate WCTT which are beyond the bounds B_i and $C_i^{\gamma_j}$. To avoid this, we integrate bounds inside the generator, implying to regenerate a transmission time if the previous one was not between the bounds. This allows us to propose reliable generation models which will not generate out of bounds transmission times or non coherent flowsets.

B. Mixed-criticality switches management

In order to be compliant with Mixed-Criticality (MC) management models proposed in [10], we integrated protocols to manage MC inside switched Ethernet networks. Based on our previous work [10], we integrated first the centralized approach that relies on a global clock synchronization. The purpose of this centralized protocol is to guarantee, through a reliable multicast (implemented in ARTEMIS core), the consistancy of the criticality level of the network in all the nodes at any time. We integrated the centralized MC management protocol

in ARTEMIS taking into account the network clock accuracy provided by a clock synchronization protocol.

The centralized protocol implies to switch the criticality level to high levels in nodes even if they do not transmit high-critical flows. This induces a loss of non-critical traffic transmissions. In order to answer to this problem, we also integrated an alternative protocol inside ARTEMIS, based on a distributed and independant MC management protocol among nodes, called the decentralized protocol. This approach does not require a global clock synchronizatio protocol.

In ARTEMIS, we integrated the potential to manage these two modes. The first mode (centralized) was the fundamental one and has already been discussed in [2]. The decentralized MC management imposed to split the criticality management from the global core time management.

We integrated inside ARTEMIS CoreScheduler a new module responsible for criticality management : the CriticalityManager. This module allows us to manage a criticality level table (for centralized protocol) and an independant criticality level value for each node (for decentralized protocol). The CoreScheduler is, among all, responsible for critical switches events and criticality table integration. These concepts were detailed in [2].

The CriticalityManager's role is to store all the different criticality switches and WCTT overuns in the network, in order to associate them with corresponding criticality level switches, either locally in a node (decentralized approach) or in the global topology (centralized approach). Its architecture is detailed in figure

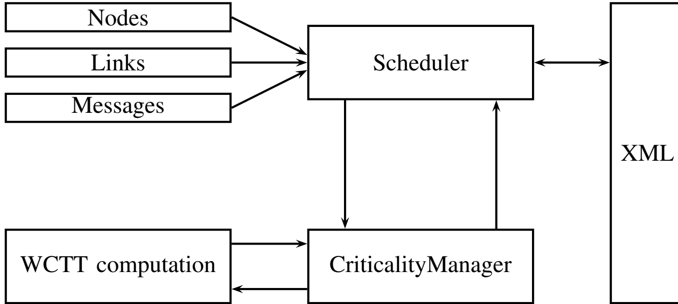


Fig. 2. CriticalityManager architecture in ARTEMIS core

At any time, each node controls if the CriticalityManager triggered a specific criticality switch for this node at a given time. As a matter of fact, the CriticalityManager is a criticality switch engine allowing the criticality level of each node to stay consistant and reliable at any moment of the simulation.

V. Simulation Results

In order to illustrate the dynamic detection of mixed criticality switches, we defined a simple topology composed of 4 switches and a set of end-systems,. We defined also a set of flows. In the different simulation environments detailed below, we defined a topology and a set of flows as described in figure 3.

The different flows parameters are detailed in the table below. We ran the simulations for a dual (LO, HI) criticality

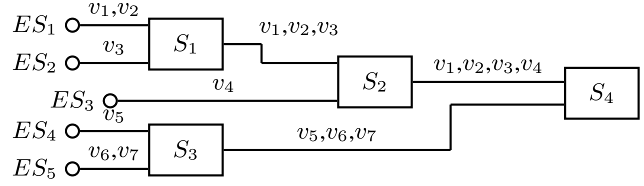


Fig. 3. Simulation environment (topology and flows)

level network. We ran a first set of simulations to compare the different linear, strict and gaussian models detailed in III.

v_i	T_i	C_i^{LO}	C_i^{HI}
v_1	80	5	8
v_2	50	4	-
v_3	80	4	8
v_4	60	3	-
v_5	70	4	7
v_6	80	5	-
v_7	50	3	-

First, we ran a simulation in dynamic centralized mode (see figure 4). Given this model, we can observe that a WCTT overun in LO mode was detected for flow v_3 is ES_2 at $t = 82\mu s$. It means that, at this time, the criticality level switches from LO to HI. As a matter of fact, the system detected that the message from ES_2 exceeded its LO-WCTT and sent a criticality switch event to the CriticalityManager. Figure 4 also shows that the transmission time generator is able to generate different transmission times (corresponding to different WCTT) for the same flow.

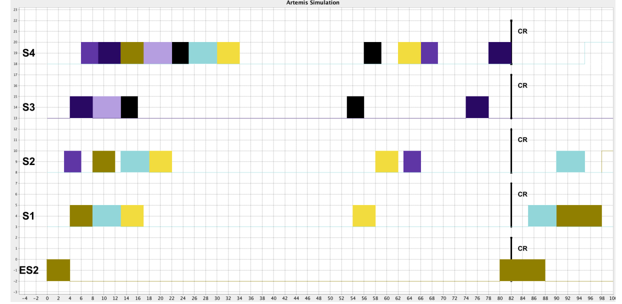


Fig. 4. Simulation in ARTEMIS with dynamic centralized mode

This first simulation is an implementation and proof of concept of MC switches management inside ARTEMIS core. This simulation shows the reliability of the centralized model and that, when a criticality switches happens, the consistancy of the criticality level in all nodes is maintained by the CriticalityManager.

In order to illustrate the decentralized MC management mode, we ran another simulation with the same parameters, but with decentralized protocol. We obtained the results showed in figure 5. We observe in this figure that each node in the path of v_1 detects its LO-WCTT overun. These detections occurs at different times on the different S_1, S_2, S_4 switches, respectively at $t = 1\mu s$, $t = 9\mu s$ and $t = 17\mu s$, indexed on the simulation parameters.

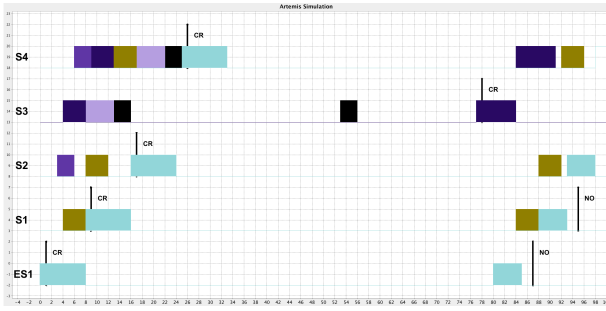


Fig. 5. Simulation in ARTEMIS with dynamic decentralized mode

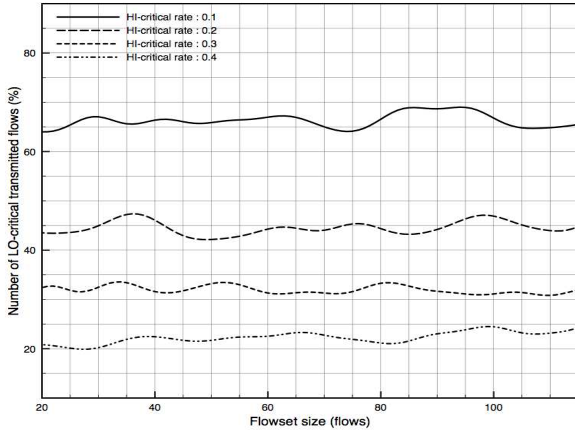


Fig. 6. LO-critical messages transmitted during HI mode

The delay to wait before switching back to LO mode is automatically set to the longest period of all flows in the topology. If no message in HI mode is received by a node during taht period, a node switch back to LO mode. We observe that the node S_1 switches back to LO-mode at $t = 96\mu$s (80μs after the end of transmission of the last HI-critical message).

As it was predicted, the decentralized network allows nodes which are not currently transmitting HI traffic to stay in LO mode. This allows us to transmit a higher amount of LO-critical traffic. In order to illustrate the number of LO critical messages which can be transmitted during both centralized and decentralized approaches, we ran a set of simulations computing the number of correctly transmitted LO messages depending on the amount of HI messages in the network.

We generated 40 different random flowsets, for a number of flowset ranging from 20 to 115. This allows to cover a various set of possible network traffic modelizations. In order to illustrate the impact of HI-critical flows, we generated these simulations for different LO to HI ratios (equal to the number of LO flows devided by the number of HI flows) ranging from 0.1 to 0.4. The results shown in figure 6 shows that, during HI-critical phases, we can assure the transmission of LO-critical messages from 20 % to 70 %, which represents a clear gain in terms of Quality Of Service (QoS).

As a conclusion, we can observe that the criticality level switches impacts the network behavior in terms of QoS, but both presented protocols assure the reliability of HI-critical messages transmission

VI. Conclusion and Perspectives

In this paper, we showed the new mixed criticality management models integrated inside the last version of ARTEMIS. Integrating different criticality protocols allows us to integrate non-critical traffic management in the mixed-criticality simulations. The integration of centralized and distributed mixed criticality switch models allows us to propose a wide range of simulation contexts. This integration makes ARTEMIS specifically designed for reliability and performances tests and practices in the domain of real-time networks simulation. ARTEMIS comes with a taskset generation tool supporting mixed criticality in the context of switched Ethernet Networks.

As a further work, we will propose an on-line downloading platform for ARTEMIS.

References

[1] L. G. X. L. Olivier Cros, Frédéric Fauberteau, “Simulating real-time and embedded networks scheduling scenarios with artemis,” in *WATERS'14*, 2014.

[2] L. G. Olivier Cros, “Mixed-criticality management of networked real-time systems with artemis simulator,” in *WATERS'15*, 2015.

[3] L. N. L. M. F. Singhoff, J. Legrand, “Cheddar: a flexible real time scheduling framework,” in *The Special Interest Group on Ada (ACM's SIGAda) 2004*, 2004.

[4] M. C. Mesonero, “Environnement de développement d'applications multipériodiques sur plateforme multicœur. la boîte à outil schedmcore,” 2012.

[5] M. Chéramy, P.-E. Hladik, and A.-M. Déplanche, “Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms,” in *Proc. of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, ser. WATERS, 2014.

[6] D. Mahrenholz and S. Ivanov, “Real-time network emulation with ns-2,” in *Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications, 2004. DS-RT 2004.*, 2004.

[7] A. Varga, *OMNeT++ user guide*, 2014.

[8] “Automotive can network response time analysis with variable jitter,” in *Mechatronics 2004 : 9th Mechatronics Forum International Conference*, 2004, pp. 785–794.

[9] T. M. Rodrigo Coelho, Mark Szczepanski and G. Fohler, “A web based monitoring tool for afdx networks,” in *WATERS'15*, 2015.

[10] X. L. Olivier Cros, Laurent George, “A protocol for mixed-criticality management in switched ethernet networks,” in *Workshop on Mixed-Criticality in Real Time Systems Symposium, WMC-RTSS'16*, 2016.

[11] R. I. D. Paul Emberson, Roger Stafford, “Techniques for the synthesis of multiprocessor tasksets,” in *Workshop on Analyzing Tools and methodologies for Embedded and Real-Time Systems(WATERS'10)*, 2010.

[12] S. Vestal, “Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance.” in *Real Time Systems Symposium(RTSS'07)*, 2007, pp. 239–243.

[13] A. Burns and R. Davis, *Mixed criticality systems: A review*. Department of Computer Science, University of York, 2013, vol. Tech. Rep.

NTGEN: a Network-on-Chip Traffic Generator toolkit for latency analysis

Ermis Papastefanakis[†‡], Laurent George[‡], Xiaoting Li[*], Ken Defossez[†]
[*]ECE Paris, 75015 Paris, France
[†]Thales Communications and Security, 92622 Gennevilliers, France
[‡]Université Paris-Est, LIGM / ESIEE, Champs sur Marne, France
Email: ermis.papastefanakis@thalesgroup.com, laurent.george@univ-mlv.fr, xiaoting.li@ece.fr, ken.defossez@thalesgroup.com

***Abstract*—Characterizing Networks-on-Chip (NoCs)-based Systems-on-Chip (SoCs) involves running many tests in software simulated as well as in hardware emulated environments. Tests help characterizing a platform and give metrics that can concern many different aspects. Each metric provides useful information for qualitative or quantitative conclusions. In this paper, we present a new tool called NTGEN that covers all the chain of actions for characterising latency on a Field Programmable Gate Array (FPGA) NoC-based platform. The toolkit, can be used for generating traffic scenarios that can be automatically launched. It helps manipulating as well as analysing the results in order to represent them into meaningful information.**

Keywords—Network-on-chip, toolkit, traffic generator, NTGEN, latency.

I. Introduction

Advances in semiconductor fabrication technologies allow chip manufacturers to include more processing cores in each new generation of products. Scalability has been an issue that needed to be addressed and the NoC paradigm was proposed as a solution to this problem [1]. It is now adopted in more and more designs [2],[3].

The effort to continue increasing performance in computer systems is leading to the adoption of highly parallel and heterogeneous platforms. Through parallelism we can distribute calculations to different Processing Elements (PEs) inside a manycore SoC and expect to obtain a high data flow rate. In addition, heterogeneous accelerators, adapted for handling specific workloads contribute to better exploit the parallel nature of such architectures.

The role of the interconnect in parallel platforms is vital in respect to their performance. As the number of PEs is steadily increasing the traditional interconnect technologies were not able to maintain scalability and started becoming a bottleneck. The NoC approach is based in the concept of taking mechanisms from computer networks and bringing them inside a SoC. NoCs suggest using a modular architecture of routers to handle the communication of PEs. This implies that in a NoC we find familiar concepts like routers and network interfaces transformed in respect to the constraints found inside a SoC ecosystem. We also find concepts such as routing algorithms, arbitration, flow control and buffering.

Initially NoCs were proposed as an alternative that would solve the scalability limits, however this quickly evolved. Researchers have been exploring NoC architectures to evaluate their properties and capabilities in other aspects that are important to SoCs. Some examples are reconfiguration, security, fault tolerance and determinism.

Since future generations of platforms for time-critical or safety critical tasks will be most likely NoCs-based, research on NoC architecture has produced concepts that are oriented to these domains. Through this exploration emerges the need for tools to validate models at a high level of detail and precision. Since such thorough validation is time consuming there is an interest for these tools to be optimised in order to perform those tasks as efficiently as possible and at the same time demand minimum human intervention and supervision.

In what concerns manycore architectures with real-time capabilities, sufficient testing needs to be performed in order to correlate theoretical results with the system model. Exhaustive tests might be necessary to cover all the potential scenarios of a use case while multiple use cases need to be considered in order to validate the system's timeliness.

Transaction-Level Modeling (TLM) allows to validate a model from a functionality perspective very fast and also provides insights in performance and timeliness as shown in [4]. However, Register Transfer Level (RTL) modeling provides more accurate results and is synthesizable which allows to also obtain realistic information on power consumption, chip surface, clock distribution etc. These advantages of RTL simulation come with the heavy cost of a rather high simulation time and high development effort to describe the hardware model. The first can be solved through emulation which allows to run a model at speeds that are three to four orders of magnitude higher than simulation. In this case the RTL model is created using a Hardware Description Language (HDL) and since emulation works on real hardware (FPGAs) it allows us to obtain a more realistic model. The downside with HDLs is the complexity in describing and debugging a model which in simulation can be more intuitive and flexible if there is a necessity to make changes.

Concerning benchmark tools, we can cite [5] where the authors present a Generic Mixed Criticality Benchmark (GMCB) providing task execution times, task criticality levels, communication patterns, and message sizes. In Mälardalen benchmarks [6], a benchmark is proposed to characterize the Worst-Case Execution Time (WCET) of applications. For timing analysis, the TACLebench [7] is a benchmark that can be used.

In the context of NoC, the MCSL benchmarks [8] can also be used to experiment classical signal processing ap-

plications including a H264 decoder, Fourier transforms, and Reed-Solomon encoders/decoders. With this benchmark, it is possible to define flow parameters (message size, paths) and the execution time of flows in the NoC. These applications define execution times. It is also possible to generate statistical traffic and use recorded traffic patterns. We are not aware of an open source toolkit such as the one presented in this paper

Our contribution: We propose a Noc Traffic GENerator (NTGEN) Verilog module that allows to produce flows (source-sink) in a NoC and a toolkit that handles this procedure by creating random test scenarios that cover the space we want to explore. The toolkit also allows us to automate the execution of all the tests, store and post-process the results. It finally also provides visualization of the results designed specifically to allow the user to add customized views for aspects of interest.

In section II, we describe the platform we use and provide information on its flow profiles. In section III we express the requirement for the toolkit and afterwards in section IV we proceed to detail the implementation. In section V we present the challenges we faced. Finally, we reach our conclusion in section VI and finish the paper by proposing our future work in section VII

II. NoC ARCHITECTURE

A. Platform

We consider a system based on a 4x4 2D mesh NoC where each node consists of a router, a Network Interface (NI) and an Intellectual Property (IP) element. Even though this is the platform that the toolkit was conceived for we considered that the effort to make it compatible with other platforms is small and was not be a limiting factor.

Each **router** R_{xy} possesses five links, four located at the edges **N**orth, **E**ast, **W**est, **S**outh (NEWS), used to connect with neighbor routers and the fifth is used to connect with the **L**ocal (L) IP$_{xy}$. At every input there is a buffer with the capacity of containing four **fl**ow control dig**its** (flits). A crossbar along with an arbiter are handling packet transmission (XY dimension routing) and flow control (Stop&Go). Virtual Channels (VCs) are not implemented and this means that a packet cannot bypass another packet that is already in an input buffer.

An illustration of a router R_{xy} is given in Figure 1.

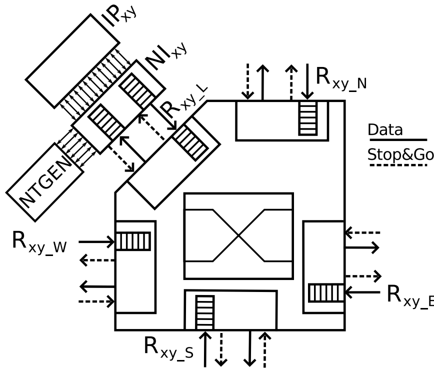


Fig. 1. Architecture of a NoC router R_{xy}

The **NI** is in charge of serializing and de-serializing packets. Packets are then split into smaller size flits in order to travel in the NoC. When an IP element makes a request for a memory location, the NI will encapsulate that into a packet, split it into flits and send them one by one to the appropriate router. When they reach their destination, the local NI will reassemble the packet, de-serialize it and forward it to the IP element that will handle the request. The same applies for the response.

A packet containing a memory response can take up 64 bytes of data and is split in 8 flits of 8 bytes each. The NI will add a header flit containing routing information making a total of 9 flits. A packet containing a request can be as small as one flit.

The **IP** element is the end point in the NoC. It can be a PE, memory, General Purpose Processor (GPP) or a peripheral (slave or a master). Consequently not all nodes in a NoC can initiate traffic.

This platform is implemented in Verilog and is able to synthesize on a FPGA (Xilinx Virtex-7). Measurements can be taken through a cycle accurate simulator (Xilinx Vivado) or through traces of the FPGA output stream passing through gigabit ethernet.

B. Network model

We consider n periodic flows transmitted in the NoC. A periodic flow τ_i sends packets respecting two parameters: 1) the period T_i which is the temporal interval between the arrival of two consecutive packets, and 2) the maximum transmission time C_i which is the maximum time to transmit all the flits of a packet on a router. In addition, the Average-Case Traversal Time (ACTT) and Worst-Case Traversal Time (WCTT) represent the average and worst-case time it took packets to traverse their path.

Due to the dimension-order X-Y routing, each packet of flow τ_i follows a static path denoted $\mathcal{P}_i$ which is composed of the source and destination IPs as well as the input ports of routers along this path. The first buffer of the source IP is denoted $first_i$, while the last buffer of the destination IP is denoted $last_i$. Then the path of flow τ_i is represented by $\mathcal{P}_i = \{first_i, ..., last_i\}$.

We consider one diffusion path in the network which means that when packets of different flows join one path, they do not leave this path until they are transmitted to the same destination (source-sink model). A real use case that illustrates this concept can be found in memory hierarchies where the last level, a common bottleneck in Multi-Processor Systems-on-Chip (MPSoCs), is the Random Access Memory (RAM). This assumption also comes from Avionics Full DupleX switched Ethernet (AFDX) network flows and is related to our previous work in [9].

III. TOOLKIT REQUIREMENTS

In order to be able to characterize the platform mentioned above and thoroughly validate new features we proceed by defining the requirements of the toolkit.

We took as input two use cases: a comparison between two arbitration schemes (First-in First-out (FIFO) and Round-Robin) as well as in validating our past work in [9]. Although both use cases were used to define requirements, in this paper we present how the toolkit was used to develop measurements

and visualization concerning the first use case of arbitration comparison. The context of this paper being to present the toolkit, we do not present the results of the use case here.

In Figure 2, we consider the NoC described in section II with 7 flows $\tau_1 \ldots \tau_7$ reaching one destination and where each IP is indexed with the coordinates of its router. The solid lines represent the paths that join to reach the destination node. We focus on flow τ_1 following the path $\mathcal{P}_1 = \{\mathrm{IP}_{00}, R_{00_L}, R_{01_W}, R_{11_N}, R_{21_N}, \mathrm{IP}_{21}\}$. The paths of the other flows are:

$$
\begin{aligned}
\mathcal{P}_2 &= \{\mathrm{IP}_{03}, R_{03_L}, R_{02_E}, R_{01_E}, R_{11_N}, R_{21_N}, \mathrm{IP}_{21}\} \\
\mathcal{P}_3 &= \{\mathrm{IP}_{20}, R_{20_L}, R_{21_W}, \mathrm{IP}_{21}\} \\
\mathcal{P}_4 &= \{\mathrm{IP}_{22}, R_{22_L}, R_{21_E}, \mathrm{IP}_{21}\} \\
\mathcal{P}_5 &= \{\mathrm{IP}_{23}, R_{23_L}, R_{22_E}, R_{21_E}, \mathrm{IP}_{21}\} \\
\mathcal{P}_6 &= \{\mathrm{IP}_{30}, R_{30_L}, R_{31_W}, R_{21_S}, \mathrm{IP}_{21}\} \\
\mathcal{P}_7 &= \{\mathrm{IP}_{32}, R_{32_L}, R_{31_E}, R_{21_S}, \mathrm{IP}_{21}\}
\end{aligned}
$$

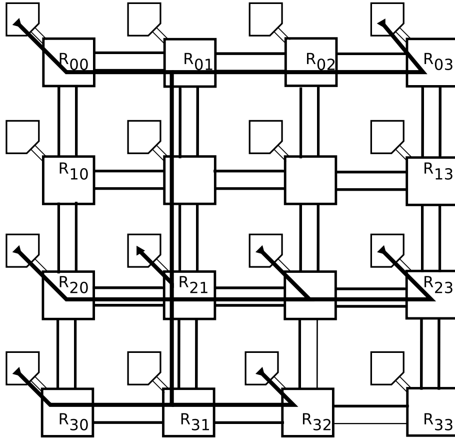


Fig. 2. NoC example of the case study

The FPGA used for the measurements already has an interface that allows us to send commands and recover traces using the gigabit ethernet port. Basic error detection is also implemented allowing to react when frames are lost. An NTGEN instance at each IP element should be addressable and modeled in a way that would make it possible to receive commands at runtime. In addition all NTGEN modules can be synchronized and start their transmission simultaneously.

Concerning the scenario generation, we wanted to be able to specify the desired average link utilization of the output node (here the link R_{21} to IP_{21}) as well as the number of traffic sources and obtain random periods to create such a pattern. Automated deployment is available based on the list of scenarios generated with minimum supervision. This means that reinitialising the NoC at the beginning of each test is also necessary. Assuming results are obtained we are able to process them and filter the information that is necessary. Eventually for the visualisation phase we exploit a framework allowing to plot the results.

During all this process we also have non-functional requirements such as performance and resource usage that should be contained in order to make this toolkit compatible with an average computer. In addition the development languages should be chosen based on their capacity to provide reusability, readability and ease of maintenance. In the same context, standard data formats should be chosen avoiding customized structures that would pose a limit to compatibility. Finally the toolkit should have a modular architecture that would make it portable to other NoC models and would allow anyone to replace or add modules to improve its functionality.

IV. Toolkit implementation

Following the requirements specified we propose a toolkit described in Figure 3. On the left we have the FPGA which is connected through an Ethernet cable to a host computer on the right. On the FPGA we have the synthesized bitstream of the NoC architecture along with NTGEN which is present in each node. On the host side we have all the software elements that the toolkit consists of.

A. Traffic Generator Implementation

NTGEN, as illustrated in Figure 1 is connected to the NI similarly to an IP element. Once configured, NTGEN can inject 9-flit long packets to the NI destined to any of the other nodes in the network. The corresponding NTGEN in the destination node will receive the packet and send a message through the Ethernet interface to the host detailing the packet's transmission. The length of the packet was intentionally chosen to represent a cache line transfer. However, if supported by the NoC model longer packets can be generated in order to represent for example Direct Memory Access (DMA) modules. This would reduce the number of messages sent to the host as we would have a lower packet per flit ratio. When synthesized, NTGEN takes around 200 Look Up Tables (LUTs) which makes it light and capable to scale as the number of nodes might increase.

B. Scenario generation tools

For the host side of the toolkit, the starting point is the random generation of periods [10] for each of the source nodes in a scenario. The input information is the number of source nodes, the desired utilization and the total number of random scenarios that are going to be generated. The algorithm in [10] is called to generate a set of periods and also calculate the necessary amount of cycles (one hyperperiod equal to the least common multiple of flow periods) that each scenario needs to run. This file is then taken by a script that aggregates multiple lists of different utilizations, source and destination nodes based on user input provided to the script. The final file generated is passed to the command interface that can interpret it and launch each of the scenarios sequentially. It will configure NTGEN in each of the source nodes using a library that transforms each line of the list to commands for the FPGA. This tool also manages the initialization of the FPGA before each scenario, the launch of each scenario as well as its termination when the hyperperiod is reached.

At the same time, the response interface (the counter-part of the control interface) receives traces from the FPGA and performs a preliminary post-processing that allows to filter a big part of the information and keep the necessary parts that are then saved in a file. This file is taken by the visualisation tool whose goal is to transform the raw information into objects. Then we can easily exploit them to produce visualization

graphs that can then be used to obtain an insight on the temporal behavior of the model.

We can understand that the user is required to interact with the toolkit mainly at the early stages as we tried to keep the configuration parameters simple. If necessary, we can obtain a finer configuration granularity by intervening inside the scripts.

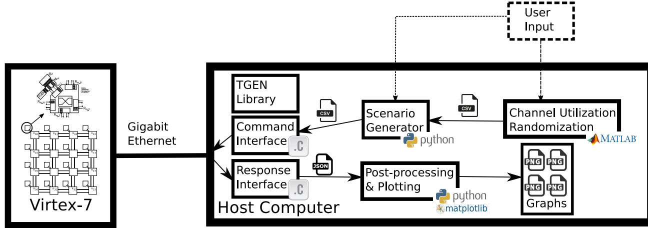


Fig. 3. Toolkit software architecture

C. FPGA interfacing

Concerning the command and response mechanisms that manage the tests on the FPGA and store the results, we can see a top-level state diagram in Figure 4. Both interfaces are launched and reach an idle state. The user launches the command to execute test scenarios and the command interface resets and configures NTGEN. The NoC model possesses a reset mechanism that enables the command interface to put it in an initial state that is consistent and allows us to perform each test with fixed initial conditions. After NTGEN is configured for the first scenario the command interface passes at a ”wait state” during which it is polling a pipe that is established for communication between the two interfaces. When the scenario is over, the response interface will send a message in that pipe and the command interface will reset the FPGA and configure NTGEN to continue with the next scenario. In this current version of the toolkit, managing errors is not supported, in the case that a packet is lost, the response interface will send a message in the pipe and both programs will exit.

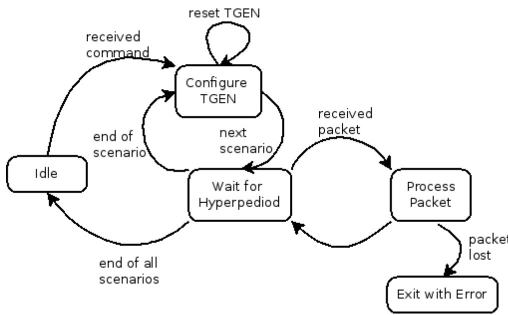


Fig. 4. Toolkit state machine

D. Visualisation

When results have been stored, we can use the visualisation script in order to plot the information into graphics (using matplotlib library). That way, we can get a better understanding on the behavior of the model in addition to having raw information.

The plotter of the toolkit provides mainly pre-coded graphics that: will automatically be plotted based on the results (see below) and allow the verification of the precision of the tested scenarios. The capability for the user to code his/her own graphs that can visualize additional aspects of the results.

In figures 5, 6, 7, 8, 9, 10, we present some of the graphs that are provided by default in the toolkit.

In figure 5 we can see all the scenarios executed and the worst case latency in each. In figures 6, 7 we have all the scenarios grouped in relation to the link utilization at the destination node. In figure 6 we see the average time it takes packets to traverse the NoC. In figure 7 we have the worst case of the worst traversal time of the scenarios with the same link utilization. This will allow us to perform side to side comparison between two features, being arbitration, routing etc. In figure 8 we obtain the histogram for a specific node so that we can observe its behavior. Finally in figures 9, 10 we have information on the precision of the tests we performed. This can be used to easily to verify that the traffic generation does indeed produce correct utilization patterns.

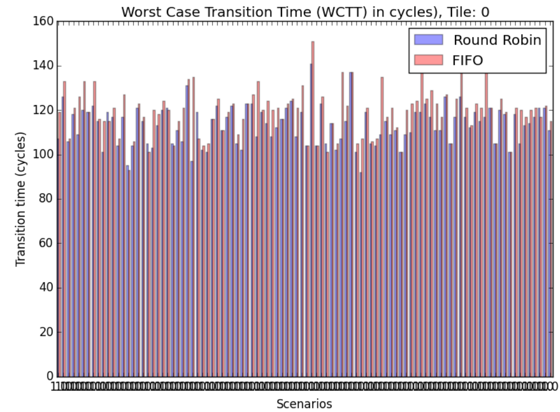


Fig. 5. Overview of all scenarios

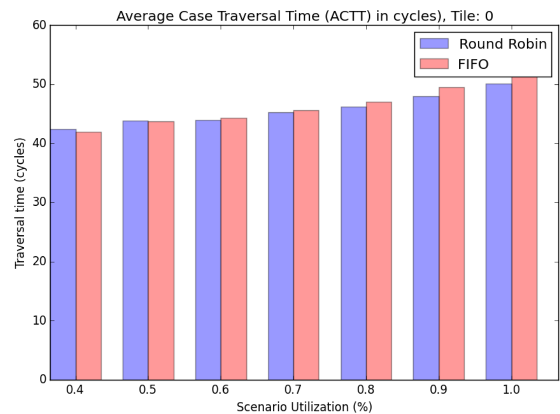


Fig. 6. Bar graph for the ACTT

V. CHALLENGES

A. Implementation of HDL traffic generator model

HDL is quite different from software development as it requires to keep the model synthesizable and to always take into account the way the synthesis tool generates the

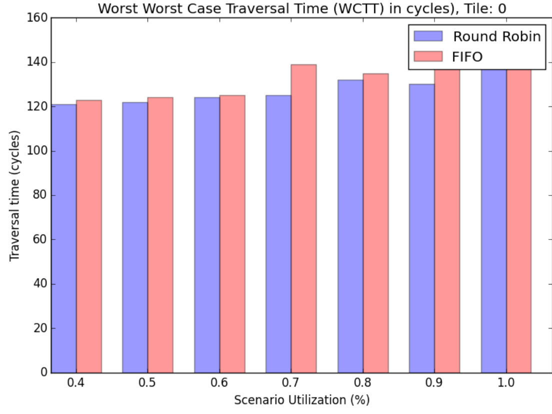


Fig. 7. Bar graph for the WCTT

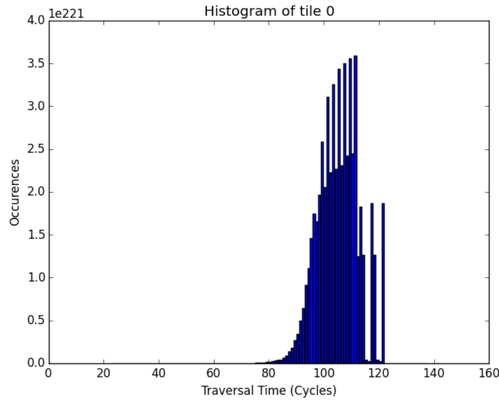


Fig. 8. Distribution of Traversal Time of node IP_{00}

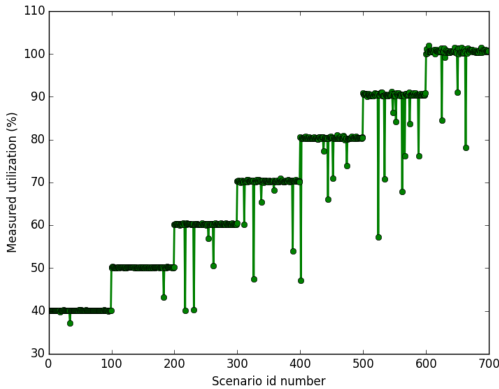


Fig. 9. Precision validation between theoretical and measured values for each scenario

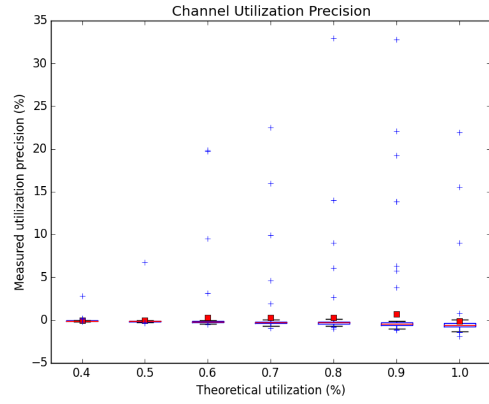


Fig. 10. Relative precision validation for each utilization set of scenarios

bitstream based on the code. For example, a modulo counter implemented using the % symbol (usual in languages such as C) will use much more resources compared to implementing the module through a register. Validation can be challenging as we can choose between two worlds, cycle accurate simulation or emulation on FPGA, each having its own advantages. With simulation we can have a very high degree of visibility of the model's behavior, monitor or alter any part of it and pause the simulation at any point. The downside is that simulation is very slow and can become inefficient in long scenarios. Emulation on FPGA can be significantly faster as the model is running in real-time. However, synthesizing the bit-stream can take long (30-120 minutes) and needs to be repeated at every modification. As a result emulation is more suitable for long scenario validation while simulation is very efficient in short fine grained validation as well as for the development/debugging phase.

B. Implementation of scenario generator

The challenges in scenario generation relate to producing enough scenarios to cover all the channel utilization needed for testing during a sufficient duration (at least one hyperperiod). At first we needed to produce random scenarios with a specific bandwidth output. This was achieved by using Randfixedsum algorithm [10] to generate each source's transmission period. The goal was to obtain periods that would cause the link at the destination node to have a desired average utilization rate. In addition, the total duration of each scenario, defined by its hyperperiod should not be too large to test it. As a result from all the scenarios generated, we only keep the ones that do not exceed 1 billion cycles.

C. Implementation of communication software with the FPGA

The main challenge here relates to performance and more specifically to the reception of packets from the FPGA. The objective is to be able to process and store packets fast enough to avoid dropping them due to congestion. Depending on the channel utilization, the FPGA would output packets to the host computer at different rates. For higher utilization scenarios the bandwidth would reach as much as 600 MBit/s and storing this information would generate files that as much as 8GBytes per scenario. Considering that we would need to execute multiple

tests of hundreds of scenarios, archiving the information would limit the capacities in testing. As a result there would have to be some processing at the reception of each packet in order to keep the useful parts of the information and reduce storage requirements.

A new challenge emerges here as the amount of processing resources is limited by the packet reception rate. The processing time should not exceed the time between packets as it will end up filling the input buffer to a point that packets will eventually be dropped. To resolve this issue various mechanisms were implemented. Firstly the CPU/memory trade-off was shifted and the memory used for the input buffer was maximised in order to be able to absorb more packet pressure giving the possibility to the CPU to complete the processing. Secondly, the processing task was attached to a specific core; avoiding migrations to other cores would optimize the resource usage. Finally, to ensure that there would be no other tasks sharing the core with the processing task, creating context switching and preemption overhead, we masked the core making it solely available to the toolkit.

Through this challenge we are able to identify that the major factor of scalability for the toolkit is the destination node. Since messages to the host computer are sent each time a packet is received, it is obvious that scaling to more cores or adding virtual channels will not affect the volume of data we receive at each scenario. However adding more destination nodes will have an impact, but again only in high bandwidth scenarios. At this case slowing the operational frequency of the FPGA is a rather feasible solution that will add more emulation time for intense scenarios but remain more efficient than simulation and keep the tool scalable. Another possibility is to offload part of the post-processing to the FPGA and receive information that, requiring little or no processing, is ready to be stored and exploited. This solution needs to be evaluating its complexity in implementation and maintenance. However, given the gain of processing in FPGAs it can have a very high potential.

Secondary challenges involve the FPGA communication protocol stack that we implemented in order to provide a layers of abstraction, making the toolkit able to be adapted to other platforms.

D. Implementation of data analysis and representation

The JSON library in Python opens data files and parses all the content into memory in a single operation. This results in the creation of multiple objects holding information in RAM that is not immediately necessary but still occupies memory space. In fact for a rather small amount of results the memory occupied by object was so big that made us realize that the analysis would be impossible for the full scenarios list. This was another reason that made processing necessary before storing the data to disk. Currently the scenario file size after partially processing results, allows Python's JSON parser to cope without problems. However, in order to anticipate for future scenarios yielding more voluminous results a more solid solution needs to be implemented.

VI. CONCLUSION

In this paper we propose a new toolkit called NTGEN for a FPGA NoCs platform performance analysis. This toolkit can be used to perform tests to validate platform features and functionalities and characterize the latency of flows sent through the NoC. NTGEN can automatically generate traffic scenarios and perform analysis to present them in useful valuable information.

VII. FUTURE WORK

At first in order to take an orientation for a more mature simple set of tools, we envision to converge the different programming languages and file formats. Secondly, we plan to enable support for traffic generation and visualization for more than one destinations. This way the toolkit will be able to handle more use cases and take a more general character. Thirdly, there is an interest to be able to generate realistic traffic patterns in addition to random ones. We intend to look into this subject so that for example, we can record real applications and replay them to simulate traffic.

Another interesting aspect would be to support results acquired from simulation in addition to emulation. The effort is minimal and it will allow to use the toolkit for small scenarios that still need visualization.

Finally, in the long term we would like to provide a Graphical User Interface (GUI) making the toolkit easier to use. Scenario generation will be easier, as well as following the progress of testing scenarios. In addition, by being able to chose subsets of data through a user interface will make managing the visualisation of the results much faster.

REFERENCES

[1] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*. IEEE, 2001, pp. 684–689.

[2] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications." in *HPEC*, 2013, pp. 1–6.

[3] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell, "Programming the adapteva epiphany 64-core network-on-chip coprocessor," *International Journal of High Performance Computing Applications*, p. 1094342015599238, 2015.

[4] L. S. Indrusiak, J. Harbin, and O. M. Dos Santos, "Fast simulation of networks-on-chip with priority-preemptive arbitration," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 4, p. 56, 2015.

[5] J. Harbin, T. Fleming, L. S. Indrusiak, and A. Burns, "Gmcb: An industrial benchmark for use in real-time mixed-criticality networks-on-chip," in *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[6] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks: Past, present and future," in *OASIcs-OpenAccess Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[7] "Taclebench website," http://www.tacle.eu/index.php/activities/taclebench, 2016, [Online; accessed 26-May-2016].

[8] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang, "A noc traffic suite based on real applications," in *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*. IEEE, 2011, pp. 66–71.

[9] E. Papastefanakis, X. Li, and L. George, "Deterministic scheduling in network-on-chip using the trajectory approach," in *Real-Time Distributed Computing (ISORC), 2015 IEEE 18th International Symposium on*. IEEE, 2015, pp. 60–65.

[10] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, 2010, pp. 6–11.

Challenge Solutions

Calculating Latencies in an Engine Management System Using Response Time Analysis with MAST

Juan M. Rivas, J. Javier Gutiérrez, Julio L. Medina and Michael González Harbour
Software Engineering and Real-Time Group, University of Cantabria, Spain.
{rivasjm, gutierjj, medinajl, mgh}@unican.es

***Abstract*—This paper reports solutions to the 2016 edition of the Formal Methods and Timing Verification (FMTV) challenge. The challenge requests calculating latencies in a complex engine management system, of which an Amalthea model is provided. We propose solving the challenge using MAST, which is a real-time systems model and also a suite of tools for schedulability analysis and optimization. The efforts to solve the challenge are mainly focused on translating the Amalthea model into the MAST model. Then, response time schedulability analysis tools are used. We discuss the strengths and limitations of our approach, and present the results obtained. Finally, we report the time needed to understand and complete the challenge. The solutions are available to the public in electronic form to facilitate their assessment by the community.**

***Keywords*— Amalthea; MAST; engine management system; real-time, response-time analysis.**

I. Introduction

This paper presents a solution to the 2016 FMTV Challenge [1] which asked calculating tight end-to-end latency bounds in a complex engine management software composed of a number of cause-effect chains. The system is provided as an Amalthea [2] model.

We propose the verification of this system by applying response time analysis (RTA) inside the MAST [3][4] analysis suite. Accordingly, the first effort that must be undertaken is to define an Amalthea to MAST model transformation path. Once an equivalent MAST model is generated, the MAST analysis tool can be used to calculate latencies, using common response-time analysis techniques, such as the offset-based analysis [5]. Using MAST enables the application of complex mathematical formulation to perform the response time analysis on an easy to understand high level abstraction model. This approach requires: (1) the correct interpretation and transformation of the provided model, (2) the selection of the most appropriate and less pessimistic analysis technique, and (3) the correct interpretation of the results provided by the tools.

The paper is organized as follows. Section II describes the MAST environment focusing on the most relevant elements used to solve the challenge. Section III deals with the interpretation of the provided Amalthea model, and how it is modelled using MAST. Section IV proposes an analysis for event chains. In Section V, the challenge results are presented. Finally, Section VI presents the conclusions of this work.

This work has been funded in part by the Spanish Government under grant number TIN2014-56158-C4-2-P (M2C2).

II. MAST tool suite

The MAST environment provides an open source set of tools to perform schedulability analysis and optimization of real-time systems [4]. These tools operate on systems described using the MAST model [3], which is key to our solution of the challenge. This model is aligned with MARTE (Modeling and Analysis of Real-Time Embedded systems) [6], a standard of the Object Management Group (OMG) for modeling and analysis of real-time and embedded systems.

A. The MAST model

The MAST model follows an event driven approach, and assumes a real-time distributed system with multiple processing resources (CPUs and communication networks). The system is composed of distributed end-to-end flows, which are released by periodic, sporadic or aperiodic sequences of external events. The relative phasing of the activations of different end-to-end flows is assumed to be arbitrary. An end-to-end flow is composed of a sequence of steps, which represent the execution of a thread in a processor, or the transmission of a message through a network. Each release of an end-to-end flow causes the execution of one instance of its sequence of steps. Each step is released when the preceding one in its end-to-end flow finishes its execution. We

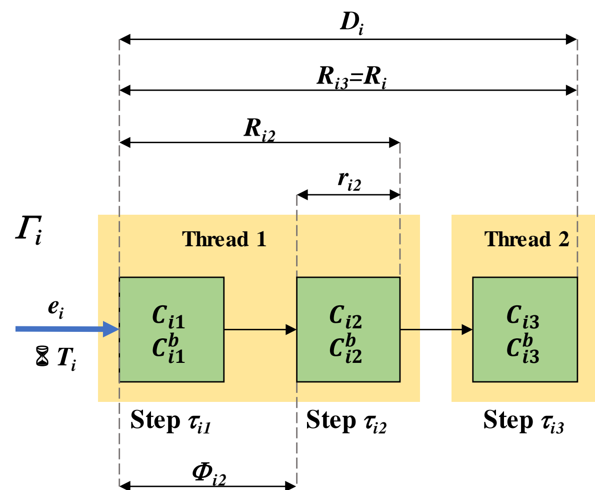


Fig. 1. Example of a simple MAST end-to-end flow with three steps.

assume that the steps are statically mapped to processing resources. The model also allows mutual exclusion synchronization in the processors.

Fig. 1 shows an example of an end-to-end flow (Γ_i) with three steps (τ_{i1}, τ_{i2}, τ_{i3}), each executing in a different processing resource PR_k. The end-to-end flow is released by the arrival of the external event e_i. This external event has a period T_i, which can also represent the minimum inter-arrival time of a sporadic arrival pattern. Steps can have an initial offset (Φ_{ij}) associated, which is the minimum imposed release time of the step, relative to the arrival of the external event. Each step has a worst-case execution time (WCET) C_{ij}, and a best-case execution time (BCET) C_{ij}^b.

MAST supports Fixed Priorities (FP) and Earliest Deadline First (EDF) scheduling. The timing requirements that we consider are end-to-end deadlines (D_i), which must be met by the completion of the last step in the end-to-end flow, relative to the arrival of the external event. The deadlines can be larger than the periods.

As a result of the response time analysis, each step τ_{ij} has a worst-case response time (or an upper bound of it) R_{ij}, and a best-case response time (or a lower bound of it) R_{ij}^b. These response times are relative to the arrival of the external event (global response times). The worst-case response time of an end-to-end flow (R_i) is the worst-case response time of its last step. The system is said to be schedulable if the worst-case response times of the end-to-end flows are lower or equal to their end-to-end deadlines ($R_i \leq D_i$).

The completion time of the steps can vary for different activations. As a consequence, the step activation time also varies. For a step τ_{ij}, we define its release jitter (J_{ij}) as its worst-case variation in activation times. The jitter is taken into account by the analysis techniques.

B. *MAST analysis tools*

To solve the challenge, we use the response-time analysis techniques included in MAST [4] on the equivalent MAST model generated from the Amalthea model. MAST implements several analysis techniques that can be applied to an FP system with end-to-end flows, ranging from the holistic analysis, to various offset-based techniques [4].

Of particular interest for this work is the Offset-Based Analysis with Precedence Relationships [5] (*offset_based_approx_w_pr* in MAST). This technique supports steps with offsets, and is capable of reducing the pessimism in the results by eliminating scenarios that would be impossible when taking into account the precedence relationships inside end-to-end flows. This characteristic is particularly helpful with end-to-end flows that don't traverse different processing resources, as it will be the case in this challenge.

Additionally, MAST can also perform sensitivity analysis by calculating the system slack, which, if positive, is defined as the percentage by which the execution times of all the steps in the system may be increased while still keeping the system schedulable. If negative, the system slack corresponds to the percentage by which WCET's would have to be decreased to make the system schedulable. Similarly, slacks for each processor can be calculated too.

MAST provides global worst-case and best-case response times of the steps in the system. For a part of this challenge we will need local response times of the steps. While these are not usually provided by MAST, we have modified the tool so it could handle local response-times too, according to [7] taking into account offsets. Then, we define local worst-case response times (r_{ij}), and local best-case response times (r_{ij}^b) as upper and lower bounds, respectively, on the completion times of steps, relative to their own local activations (see Fig. 1). This custom version of MAST will be made available in addition to the transformation and generated models.

III. AMALTHEA TO MAST MODEL TRANSFORMATION

The 2016 FMTV Challenge provides an Amalthea model of a full blown engine management system. The complexity of the system is made apparent just by looking at the model file, which has approximately 71000 lines. In this section, we will describe how we interpret the Amalthea model, and how the equivalent MAST description of the engine management system is created. While Amalthea defines a vast meta-model supporting many types of elements and use-cases, we will limit our transformation to the elements relevant for this challenge.

Amalthea tasks represent the schedulable elements in the model. For the case of the challenge, they have the following characteristics:

- Tasks are activated by periodic or sporadic stimuli with minimum inter-arrival times. Stimuli are assumed to have arbitrary phasing (property "Clock" of the stimuli is undefined). Timing constraints are given as deadlines that the tasks must meet. In this case, deadlines are equal to the periods (tasks must finish before their next activation).
- Tasks are statically assigned to a core, and are scheduled with a fixed priority policy. Tasks can be preemptive (they can preempt any lower priority task at any moment), or cooperative (they can preempt lower priority cooperative tasks only at the termination of runnables). In the provided model, cooperative tasks always have lower priority than preemptive tasks.
- Each Amalthea task in the model executes a sequential list of Runnables. Each Runnable is composed of three sequential stages: (1) label (memory) read accesses, (2) execution of instructions in the assigned processing core, and (3) label (memory) write accesses. Some Runnables don't write or read from memory.

We interpret Amalthea tasks as MAST end-to-end flows, in which each runnable is transformed into a MAST step. For sporadic Amalthea tasks, the resulting MAST end-to-end flow will be periodic, with a period equal to the minimum inter-arrival time. This interpretation is only correct for flows with offsets within the periods [8]. Since in the Amalthea model the flow deadlines are within the periods so are the step offsets. If the offsets were larger than the periods, the MAST flows would need to be sporadic and the worst-case response times would be larger. The deadline of the Amalthea task is directly used as the

end-to-end deadline of its corresponding MAST end-to-end flow.

MAST lacks a specialized element to model memories. Additionally, it also doesn't implement any mechanism to model the blocking of a processor while it is accessing a memory, thus disallowing us to model memory as a general purpose device. With these limitations in mind, we will model the memory accesses as execution times added to the MAST steps, accounting for the worst-case and best-case costs of accessing the memory. The worst-case cost of accessing a label pessimistically assumes that every core is accessing that memory at the same time. Therefore, if we consider that only the global memory is used (second question of the challenge), the worst-case cost of accessing a label is 4*9 cycles. Similarly, the best-case cost of accessing a label assumes that no other core is in the queue for that memory, so this value is just 9 cycles (no contention).

Accordingly, in the runnable to MAST step transformation, the worst-case execution time of the step (C_{ij}) is calculated as the sum of two elements: (1) the execution time of the upper bound of the number of instructions of the runnable, and (2) the worst-case cost of accessing the labels. If a runnable accesses N labels (read and/or write), the worst-case cost would be N*4*9 cycles if we assume that only global memory is used. Likewise, the best-case execution time (C_{ij}^{b}) of the step is calculated as the sum of the lower bound of the instructions of the runnable, and the best-case cost of accessing the labels (N*9 cycles).

Additionally, we also take into account the blocking effect in a thread accessing the memory due to a label being accessed by a lower priority thread in the same core, even though this is almost negligible. This is modeled by including in each core a shared resource protected by the Immediate Ceiling protocol that is accessed by each step during 9 cycles. This produces one blocking of 9 cycles to each higher priority thread, which is the intended effect.

Fig. 2 depicts the transformation of a simple Amalthea task (Fig. 2a) into a MAST end-to-end flow (Fig. 2b). If memory accesses are ignored, as stated in the first question of the challenge, the executions times of the resulting MAST steps only include the execution times produced by the instructions.

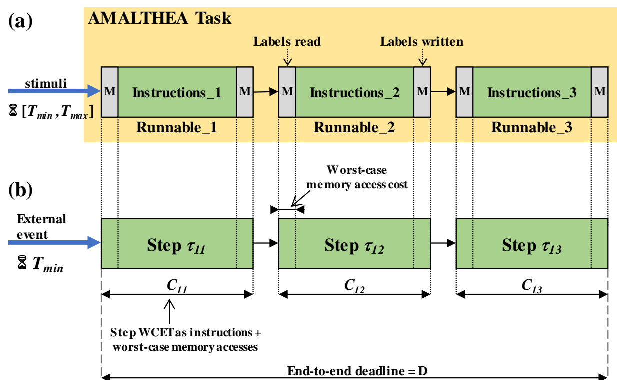


Fig. 2. (a) Example of a simple Amalthea task with three Runnables, and (b) its MAST end-to-end flow equivalent used in this work

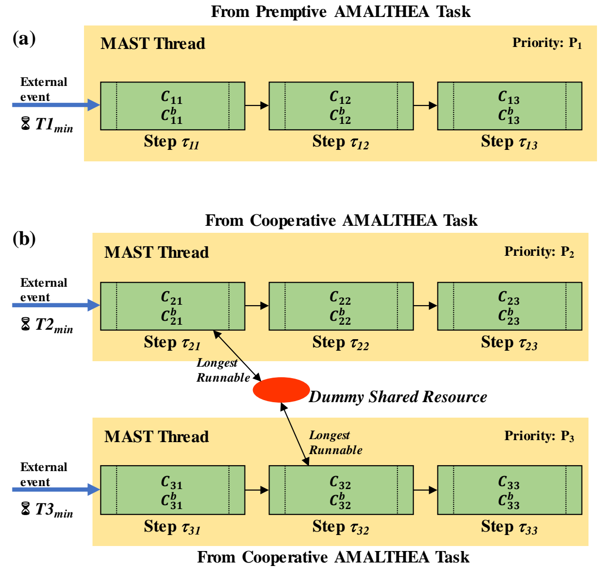


Fig. 3. Equivalent Amalthea tasks as MAST end-to-end flows, for (a) preemptive , and (b) cooperative Amalthea tasks.

MAST supports non-preemptive tasks, but they cannot be preempted by any task. This is not aligned with the behavior of Amalthea cooperative tasks, which can be preempted by preemptive tasks. To model cooperative tasks, we will take into account that in the worst-case scenario, these tasks will be blocked by an amount equal to the longest cooperative runnable with lower priority. In MAST we can induce this blocking adding a dummy shared resource that is used by the longest runnable of each cooperative task. MAST automatically finds the longest possible blocking that affects each task. Fig. 3a depicts a MAST end-to-end flow transformed from a preemptive Amalthea task, while Fig. 3b shows the transformation of two Amalthea cooperative tasks.

IV. Analysis of Event Chains

We interpret event-chains as a latency model for non-consecutive runnables communicating via shared memory. The first runnable in the event-chain writes a result in a label. Then the next runnable in the chain reads this label, process it, and writes its result in another label, and so on. Runnables in an event-chain can belong to the same Amalthea task or not. Even though MAST does not support this kind of "virtual" end-to-end flows, it provides results that can be used to calculate bounds for the best and worst-case latencies of the event-chains.

We distinguish two types of event-chains: event-chains that stay in the same Amalthea task; and event-chains that traverse different Amalthea tasks. Each kind requires a different formulation to calculate the end-to-end latencies.

A. Event-chains that traverse different Amalthea tasks

Fig. 4 shows the MAST equivalent model of a simple event-chain that traverses three Amalthea tasks. This is the behavior that follows *EffectChain_2* and *EffectChain_3* event-chains in the challenge. Let us use the simple example shown in Fig. 4 to

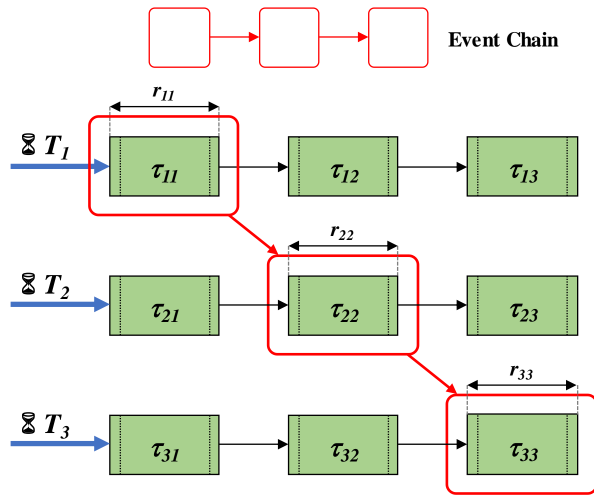


Fig. 4. Interpretation of an event-chain traversing different MAST end-to-end flows.

explain how to formulate the latencies for this kind of event-chain.

The worst-case latency of the event-chain (L) comprises the sum of the worst-case local response times of the steps in the chain (r_{ij}), and the periods of all the end-to-end flows but the first one. The periods should be added because in the worst-case situation it is assumed that at the time a label is written, the next runnable in the chain has just executed, so the chain cannot continue until the next period. For sporadic stimuli, the period added must be its upper bound. Similarly, the best-case latency (L^b) is calculated by summing the best-case local response times (r_{ij}^b). In this case periods are not added, because the best case is built when a label is read immediately after the previous runnable in the chain updated its value. The formulation for the worst and best case latencies for the event-chain shown in Fig. 4 is formalized as follows:

$$L = r_{11} + T_2 + r_{22} + T_3 + r_{33}$$

$$L^b = r_{11}^b + r_{22}^b + r_{33}^b$$

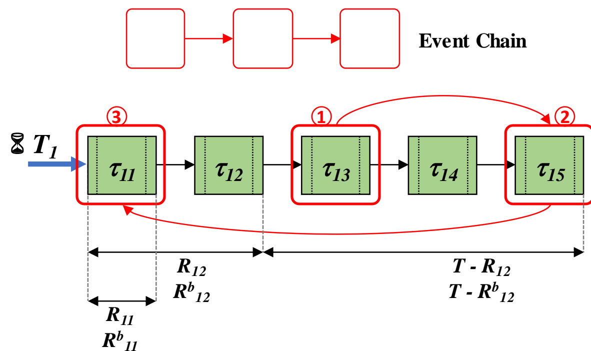


Fig. 5. Interpretation of an event-chain going backwards in the same MAST end-to-end flow.

B. *Event-chains that go back in the same Amalthea task*

Fig. 5 shows the MAST equivalent model of a simple event-chain that traverses the same Amalthea task backwards. This is the behavior of *EffectChain_1* in the challenge. For this kind of event-chains it is trivial to see that to go backwards, the chain requires an additional activation of the Amalthea task.

Using the simple example shown in Fig. 5 as reference, for this type of event-chains the worst-case latency (L) occurs when the first label in the chain is read as soon as possible (R_{12}^b), so the chain has to wait the maximum amount of time until the next activation of the end-to-end flow. Then, the event-chain must wait for the worst-case completion time of step τ_{11} (R_{11}). Since the end-to-end flow must finish before its next activation, the response time of step τ_{15} is irrelevant in this calculation. The total worst-case latency for this type of event-chain is formalized with the following equation:

$$L = (T_1 - R_{12}^b) + R_{11}$$

Likewise, the best-case latency (L^b) of the event-chain occurs when the first label is read as late as possible (R_{12}) and step τ_{11} finishes as soon as possible (R_{11}^b). The best-case latency for these kind of event-chains can be calculated with the following formula:

$$L^b = (T_1 - R_{12}) + R_{11}^b$$

V. Evaluation

To transform the provided Amalthea model to MAST we developed an *ad-hoc* tool written in Java, consisting on less than 400 lines of code. This tool reads the challenge model using the Eclipse EMF framework [9], and builds an equivalent MAST model piece by piece using the interpretations described in Section III. The transformation of the given Amalthea model to MAST takes approximately 10 minutes, most of which are spent by the EMF framework loading the Amalthea model. The generated MAST model has approximately 23000 lines.

We proceed to solve the questions raised in the challenge, that is, to calculate end-to-end latencies that are as tight as possible. The challenge doesn't explicitly specify which are the end-to-end latencies that must be calculated. We provide end-to-end latencies for the Amalthea tasks (since they all have timing requirements), and for the event-chains described in the model. The analysis technique used has been the Offset-Based Analysis with Precedence Relationships [5]. This is the less pessimistic technique for end-to-end flows that only traverse one processor. The analysis tool takes from 1 to 5 minutes to execute, depending on the utilization of the system. The calculations of the slacks took up to 2 hours, since they involve iterative executions of the analysis tool.

In a first attempt to get analytical worst-case latencies, we used the upper bounds of the number of instructions of the runnables as the WCET of the MAST steps. The total utilization of that system goes above 100%. Using response time analysis in such situation automatically yields unbounded (infinite) worst-case response times. While utilizations over 100% can be handled by other techniques (e.g., simulators), they are not appropriate when applying response time analysis. After knowing that all upper-bounds in the original Amalthea model

can never occur at the same time, and not having the realistic models for each relevant real-time situation, we decided to consider two scenarios: Scn-ACET, and Scn-WCET.

In Scn-ACET, the worst-case execution times of the steps are calculated using the mean value of the number of instructions of the runnables. In Scn-WCET the worst-case execution times of the steps are calculated with the upper bound of the number of instructions (as described in Section III). In both scenarios we calculate latencies for different CPU clock frequencies, from the default 200Mhz and above (233Mhz, 266Mhz, etc.), until the timing requirements in the system are met. We essayed common CPU frequencies only. Additionally, for each analyzed case, we also calculate the system slack, and the slack of each core.

A. Ignoring Memory Accesses

Table I shows the results when memory accesses are ignored. Shadowed cells indicate tasks that don't meet their deadlines. We can see that for Scn-ACET, 200Mhz is enough to make the system schedulable, with a system slack of 9.77%. If the clock frequency is increased to 233Mhz, system slack increases to 27.73%. For Scn-WCET, schedulability is achieved at 300 Mhz, with a system slack of 8.98%. If we observe the slack in each core, we can see that CORE1 is always the most constrained (lowest positive slack). This is to be expected, as this core has higher utilization among all cores.

B. Adding Memory Accesses, using Global Memory Only

We repeat the process, but this time considering the memory accesses. As a reminder, the memory accesses are modelled as additional WCET of the steps, considering the worst-case cost of accessing each label. The results are shown in Table II. As can be expected, the core utilizations now increase compared to the case without memory accesses (Table I). The increase in utilization is between 3% and 12%, depending on the core. As a consequence, there is a system-wide increase in latencies too.

In this situation, Scn-ACET is not schedulable at 200 Mhz (Angle_Sync task misses its deadline in its worst-case). In this scenario, schedulability is achieved at 233 Mhz, with a system slack of 13.67%. On the other hand, Scn-WCET is schedulable at 300 Mhz, although with a marginal system slack of just 0.78%. At 333 Mhz, this system slack increases to 11.72%.

C. Re-mapping Labels

The final question of the challenge asks for an optimization of the label-to-memory mapping to minimize the latencies. MAST does not provide a model for mapping memories, so we propose a reasonable solution. We identify that the majority of the labels are only accessed from a single core. As a first step, we map those labels into their local memories. Now the problem is reduced to determining where to map the labels shared by more than one core.

TABLE I. END-TO-END LATENCIES (MILLISECONDS.) AND SLACKS (%), **IGNORING MEMORY ACCESSES.**

	Scn-ACET		**Scn-WCET**			
	200 Mhz	***233 Mhz***	***200 Mhz***	***300 Mhz***	***333 Mhz***	***D***
CORE0 Util. (%)	71.47	61.35	97.02	64.68	58.27	
CORE1 Util. (%)	88.38	75.86	133.57	89.05	80.22	
CORE2 Util. (%)	71.36	61.26	106.85	71.24	64.18	
CORE3 Util. (%)	77.19	66.25	117.94	78.62	70.83	
System Slack (%)	9.77	27.73	-27.34	8.98	21.09	
CORE0 Slack (%)	31.08	52.89	-98.44	45.12	60.52	
CORE1 Slack (%)	10.29	28.46	-98.44	9.35	21.2	
CORE2 Slack (%)	40.37	63.58	-98.44	40.37	55.66	
CORE3 Slack (%)	29.1	50.21	-98.44	26.56	40.37	
Angle_Sync	5.54	3.86	∞	5.59	4.58	**6.66**
ISR_1	0.03	0.02	∞	0.02	0.02	**9.5**
ISR_10	0.02	0.02	∞	0.02	0.02	**0.7**
ISR_11	1.45	1.23	∞	1.29	1.16	**5**
ISR_2	0.04	0.03	∞	0.04	0.03	**9.5**
ISR_3	0.06	0.05	∞	0.05	0.05	**9.5**
ISR_4	0.50	0.43	∞	0.46	0.41	**1.5**
ISR_5	0.21	0.18	∞	0.19	0.17	**0.9**
ISR_6	0.23	0.20	∞	0.21	0.19	**1.1**
ISR_7	1.21	0.86	∞	0.90	0.81	**4.9**
ISR_8	0.75	0.63	∞	0.66	0.59	**1.7**
ISR_9	2.46	1.48	∞	2.20	1.39	**6**
Task_1000ms	31.18	17.63	∞	31.14	18.63	**1000**
Task_100ms	31.01	17.48	∞	30.97	18.47	**100**
Task_10ms	7.72	6.62	∞	7.86	7.08	**10**
Task_1ms	0.52	0.45	∞	0.51	0.46	**1**
Task_200ms	31.09	17.55	∞	31.05	18.55	**200**
Task_20ms	9.55	7.95	∞	9.78	8.81	**20**
Task_2ms	0.29	0.25	∞	0.27	0.24	**2**
Task_50ms	12.77	9.91	∞	12.99	11.46	**50**
Task_5ms	0.93	0.80	∞	0.89	0.80	**5**
EffectChain_1 (*L*)	12.63	12.25	∞	12.67	12.40	
EffectChain_2 (*L*)	25.23	23.10	∞	25.44	23.86	
EffectChain_3 (*L*)	63.38	60.72	∞	63.85	62.44	

TABLE II. END-TO-END LATENCIES (MILLISECONDS.) AND SLACKS (%), **INCLUDING MEMORY ACCESSES, USING GLOBAL MEMORY ONLY**

	Scn-ACET		**Scn-WCET**			
	200 Mhz	***233 Mhz***	***200 Mhz***	***300 Mhz***	***333 Mhz***	***D***
CORE0 Util. (%)	73.75	63.31	99.30	66.20	59.64	
CORE1 Util. (%)	99.21	85.16	144.41	96.27	86.73	
CORE2 Util. (%)	76.40	65.58	111.89	74.59	67.20	
CORE3 Util. (%)	86.10	73.91	126.85	84.57	76.19	
System Slack (%)	-2.34	13.67	-32.81	0.78	11.72	
CORE0 Slack (%)	-98.44	48.47	-98.44	41.92	57.57	
CORE1 Slack (%)	-1.92	14.20	-98.44	1.18	12.21	
CORE2 Slack (%)	-98.44	52.89	-98.44	33.80	48.47	
CORE3 Slack (%)	-98.44	34.50	-98.44	17.87	30.41	
Angle_Sync	6.95	4.85	∞	6.60	4.96	**6.66**
ISR_1	0.03	0.03	∞	0.03	0.02	**9.5**
ISR_10	0.03	0.02	∞	0.02	0.02	**0.7**
ISR_11	2.26	1.27	∞	1.32	1.19	**5**
ISR_2	0.05	0.04	∞	0.04	0.04	**9.5**
ISR_3	0.07	0.06	∞	0.06	0.05	**9.5**
ISR_4	0.52	0.45	∞	0.47	0.42	**1.5**
ISR_5	0.22	0.19	∞	0.20	0.18	**0.9**
ISR_6	0.24	0.21	∞	0.22	0.20	**1.1**
ISR_7	1.25	0.89	∞	1.09	0.83	**4.9**
ISR_8	0.78	0.65	∞	0.67	0.61	**1.7**
ISR_9	2.53	2.15	∞	2.27	1.44	**6**
Task_1000ms	33.91	19.32	∞	33.03	19.64	**1000**
Task_100ms	33.55	19.02	∞	32.74	19.37	**100**
Task_10ms	8.61	7.39	∞	8.45	7.62	**10**
Task_1ms	0.58	0.50	∞	0.54	0.49	**1**
Task_200ms	33.71	19.15	∞	32.87	19.49	**200**
Task_20ms	11.21	8.79	∞	11.15	9.22	**20**
Task_2ms	0.32	0.27	∞	0.29	0.26	**2**
Task_50ms	13.63	11.42	∞	13.57	11.96	**50**
Task_5ms	0.97	0.84	∞	0.92	0.83	**5**
EffectChain_1 (*L*)	12.93	12.52	∞	12.87	12.59	
EffectChain_2 (*L*)	26.17	23.89	∞	26.07	24.67	
EffectChain_3 (*L*)	64.20	62.20	∞	64.40	62.91	

In our pessimistic approach for modeling the memory accesses, even if just one label in the local memory is accessed from a non-local core, every label in that local memory would be impacted. For example, consider a local memory with labels that are accessed from two cores: the local core and a non-local core. In this case, and regardless of from which core the memory is accessed, the worst-case cost assumes that both cores are accessing the memory at the same time, and thus that cost for reading or writing any of its labels would be 1 cycle + 9 cycles = 10 cycles.

To preserve the advantage of local memory accesses, we map into global memory every label shared among different cores. Therefore, local labels are assured to be accessed without contention (1 cycle access only), and the worst-case cost for shared labels is modelled as in Section III; that is, assuming that all cores are accessing global memory at the same time (a cost of 4*9 cycles for each label access). Table III shows the slacks and latencies obtained using this mapping, which confirms that the new mapping improves the results. It is also worth noting that with this new mapping, the results are closer to the case ignoring memory accesses (Table I), than to the case in which all labels are mapped to the global memory (Table II).

TABLE III. END-TO-END LATENCIES (MILLISECONDS.) AND SLACKS (%), **RE-MAPPING LABELS TO LOCAL AND GLOBAL MEMORIES**

	Scn-ACET		**Scn-WCET**			
	200 Mhz	***233 Mhz***	***200 Mhz***	***300 Mhz***	***333 Mhz***	***D***
CORE0 Util. (%)	71.98	61.78	97.53	65.02	58.57	
CORE1 Util. (%)	92.14	79.09	137.33	91.56	82.48	
CORE2 Util. (%)	72.28	62.04	107.77	71.84	64.72	
CORE3 Util. (%)	79.85	68.54	120.6	80.4	72.43	
System Slack (%)	5.08	22.66	-29.3	5.86	17.58	
CORE0 Slack (%)	30.41	51.98	-98.44	44.31	60.52	
CORE1 Slack (%)	5.75	22.94	-98.44	6.19	17.87	
CORE2 Slack (%)	38.11	61.52	-98.44	38.86	54.72	
CORE3 Slack (%)	24.72	45.12	-98.44	24.12	37.38	
Angle_Sync	5.78	4.50	∞	5.75	4.71	**6.66**
ISR_1	0.03	0.02	∞	0.02	0.02	**9.5**
ISR_10	0.02	0.02	∞	0.02	0.02	**0.7**
ISR_11	1.47	1.24	∞	1.30	1.17	**5**
ISR_2	0.04	0.03	∞	0.04	0.03	**9.5**
ISR_3	0.06	0.05	∞	0.05	0.05	**9.5**
ISR_4	0.51	0.44	∞	0.46	0.41	**1.5**
ISR_5	0.21	0.18	∞	0.19	0.17	**0.9**
ISR_6	0.23	0.20	∞	0.21	0.19	**1.1**
ISR_7	1.22	0.87	∞	1.07	0.81	**4.9**
ISR_8	0.76	0.63	∞	0.66	0.60	**1.7**
ISR_9	2.47	1.49	∞	2.23	1.39	**6**
Task_1000ms	31.63	17.89	∞	31.43	18.81	**1000**
Task_100ms	31.42	17.71	∞	31.24	18.64	**100**
Task_10ms	7.98	6.85	∞	8.04	7.24	**10**
Task_1ms	0.54	0.47	∞	0.52	0.47	**1**
Task_200ms	31.52	17.80	∞	31.34	18.73	**200**
Task_20ms	9.68	8.31	∞	9.86	8.88	**20**
Task_2ms	0.30	0.25	∞	0.27	0.24	**2**
Task_50ms	12.93	10.84	∞	13.10	11.56	**50**
Task_5ms	0.94	0.80	∞	0.90	0.81	**5**
EffectChain_1 (*L*)	12.71	12.33	∞	12.73	12.46	
EffectChain_2 (*L*)	25.40	23.25	∞	25.56	23.97	
EffectChain_3 (*L*)	63.53	60.83	∞	63.95	62.52	

VI. CONCLUSIONS

This paper provides general guidelines to transform an Amalthea timing model into a MAST equivalent model that can be used in the MAST Analysis Tool Suite. Using them, response time analysis has been applied to calculate worst case latencies of tasks in a complex engine management system.

To understand the Amalthea model, we relied on the documentation of the tool [2], and the document describing the challenge [1]. While the basics of the model (e.g., tasks and runnables) can be easily understood with these materials, special elements of the model such as the event-chains required additional inquiries in the workshop forum. The total amount of time needed to completely digest the model can be approximated to about 12-14 hours divided in several days. Once the model was understood, the process of building the Amalthea to MAST transformation in Java required approximately 5 man-hours to a person familiar with MAST and EMF. The workspace used in this paper can be downloaded from [10].

The paper answers the three main questions of the challenge, (1) providing latencies when memory accesses are ignored, (2) providing latencies when all labels are mapped to the global memory, and (3) finding a new optimized mapping. Safer CPU frequencies as well as indicators of the most loaded tasks and cores in the system are provided. The main weakness we identify in our proposal is its pessimism in the modelling of global memory accesses. It uses an upper bound that cannot occur in reality. This is done to overcome the limitations of MAST which does not currently model the memory and the blocking of the processor while the memory is accessed. These two shortcomings have flagged interesting developments that we will explore in the future.

REFERENCES

[1] 2016 Formals Methods and Timing Verification (FMTV) challenge, co-located with the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS). https://waters2016.inria.fr/challenge/

[2] AMALTHEA: An Open Platform Project for Embedded Multicore Systems, http://www.amalthea-project.org/

[3] M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and J.M. Drake Moyano, "MAST: Modeling and Analysis Suite for Real Time Applications," Proceedings of 13th ECRTS conference, Delft, The Netherlands, IEEE Computer Society Press, pp. 125-134, June 2001.

[4] MAST web-page, http://mast.unican.es/

[5] J. C. Palencia and M. González Harbour, "Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems," Proceedings of the 20th Real-Time Systems Symposium, IEEE Computer Society Press, pp 328-339, December 1999.

[6] Object Management Group, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems," 2011 OMG Document, v1.1 formal/2011-06-02.

[7] J.C. Palencia, J.J. Gutiérrez, and M. González Harbour. "On the Schedulability Analysis for Distributed Hard Real-Time Systems," Proc. of the 9th Euromicro Workshop on Real-Time Systems, pp. 136-143, June 1997.

[8] J. C. Palencia. "Análisis de planificabilidad de sistemas distribuidos de tiempo real basados en prioridades fijas", Phd Thesis, University of Cantabria, July 1999.

[9] Eclipse Modeling Framework (EMF), https://eclipse.org/modeling/emf/

[10] Amalthea workspace used for this solution: www.istr.unican.es/members/rivasjm/workspace_fmtv16_public.zip

A Novel Analytical Technique for Timing Analysis of FMTV 2016 Verification Challenge Benchmark

Junchul Choi, Donghyun Kang, and Soonhoi Ha
Department of Computer Science and Engineering, Seoul National University, Seoul, Korea,
Email: {hinomk2, kangdongh, sha}@iris.snu.ac.kr

***Abstract*—In this paper, we present solutions to FMTV 2016 verification challenges, combining the response time analysis and schedule time bound analysis. The worst case response time of a task is computed by the conventional response time analysis while the end-to-end latency of a cause-effect chain is conservatively estimated by considering the schedule time bounds of associated runnables. Three separate challenges are discussed in order. The proposed technique is first explained to address the first challenge that ignores the memory latency. For the second challenge, we estimate the memory access latency by computing the maximum possible arbitration delay with arrival curve analysis. Finally, we propose a heuristic algorithm that determines the mapping of data labels to optimize the end-to-end latency.**

I. Challenge Model and Terminologies

We first review the Amalthea performance model [1] of the benchmark, making some assumptions for unclear explanation in the provided problem specification [2][3].

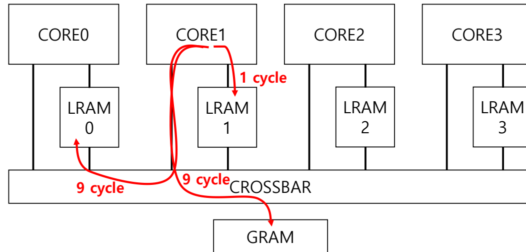


Fig. 1. Microcontroller architecture used in the challenge

The provided Amalthea model contains a hardware model of a simplified microcontroller architecture with four symmetric cores as shown in Fig. 1. Each core A_i^C has its own local memory A_i^L. A crossbar network is used for the interconnection among cores and a global memory A^G.

A task τ_i is a basic mapping unit onto a core and task-to-core mapping is given. The core τ_i is mapped to is denoted by m_i. A task is invoked either periodically or sporadically. I_P and I_S denote a set of periodic tasks and a set of sporadic tasks, respectively. The minimum and the maximum initiation interval are specified for each task τ_i and are denoted as p_i^l and p_i^u. If τ_i is a periodic task ($\tau_i \in I_P$), p_i^l is equal to p_i^u which means that the initiation interval becomes the period. All tasks are simultaneously initiated at the system activation time. The basic timing requirement for task τ_i is to finish execution before its deadline denoted by d_{τ_i}. Since implicit deadline model is assumed, deadline d_{τ_i} is equal to p_i^l.

A task τ_i consists of a set of runnables $\{r_{i,j} \mid 1 \leq j \leq |\tau_i|\}$ where runnable $r_{i,j}$ is an unit of execution and $|\tau_i|$ means the number of runnables in the task. Runnables in a task are executed sequentially on the mapped core in the ascending index order. The lower and the upper bound of execution time of $r_{i,j}$, denoted $c_{i,j}^l$ and $c_{i,j}^u$, are specified assuming that code is executed directly from core-exclusive flashes without contention. Note that memory access delay is not included in the execution times. The runnables are assumed to read all required data at the beginning of their execution and write back the results after execution is completed. We assume that when a runnable attempts to access a memory, no preemption is allowed until the resource request is processed. 1250 runnables are specified in the provided model.

A distinct priority is assigned to each task for the fixed-priority scheduling. We assign each task a unique index in the descending priority order; task τ_i has a higher priority than τ_j if $i < j$. A task τ_i is scheduled by either preemptive or cooperative fixed priority scheduling policy. S_P and S_C denote a set of preemptive tasks and a set of cooperative tasks, respectively. A task $\tau_i \in S_P$ can preempt lower priority tasks at any time, whereas a task $\tau_i \in S_C$ can preempt lower priority cooperative tasks at the boundary of runnable executions [4]. There are 21 tasks and preemptive tasks have higher priorities than cooperative tasks in the provided model.

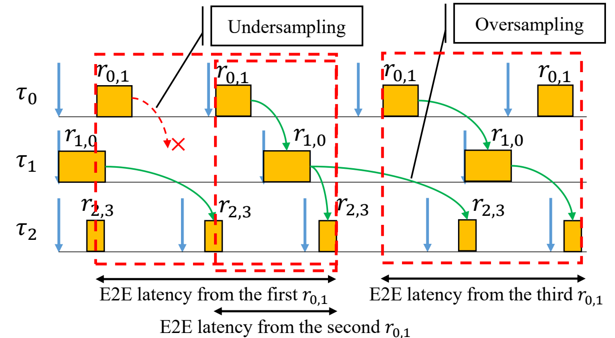


Fig. 2. End-to-end latency of an example cause-effect chain

A cause-effect chain CEC_i defines a chain of runnables that are connected by read/write dependency with labels. Note that there are no cyclic dependencies between tasks within a cause-effect chain. Due to the potential different task periods, data may get lost (undersampling) or get duplicated (oversampling).

We assume an end-to-end latency of a cause-effect chain as the maximum time duration between the first input that may be undersampled and the first output generated from the corresponding or later input. This semantic is as same as the *reaction time constraint* of the AUTOSAR [5]. Fig. 2 shows end-to-end latencies from three stimulus runnable instances in an example cause effect chain $\{r_{0,1}, r_{1,0}, r_{2,3}\}$. Since we are concerned about reaction time, the second $r_{1,0}$ instance is regarded as the reaction of the first $r_{0,1}$ instance. The third $r_{2,3}$ instance is the first response to the second $r_{1,0}$ instance so that final reaction of the first $r_{0,1}$ instance is generated by the third $r_{2,3}$ instance. Three cause-effect chains are specified in the provided model.

Data is specified by a set of labels: each size is less than the memory transfer size 32bits. Memory arbitration model is assumed differently in each challenge as follows:

- **Challenge 1: calculate tight end-to-end latencies ignoring memory accesses and arbitration**
  All read/write accesses to labels take zero time so that only runnable execution times affect the end-to-end latencies.
- **Challenge 2: calculate tight end-to-end latencies including memory accesses and arbitration**
  All labels are assumed to be stored in the global memory. Read and write accesses have symmetric memory access times. When accessing the global memory, crossbar transfer takes 8 cycles and access to global memory takes 1 cycle. When there is a contention at global memory, the accesses are assumed to be arbitrated according to the FIFO policy.
- **Challenge 3: optimize end-to-end latencies by mapping the labels among the local and global memories**
  We can map a label in a local memory whose access latency is 1 cycle. We assume that local memory size is limited. Local memories are also arbitrated according to the FIFO policy.

For all challenges, we aim to conservatively estimate the upper bound of response time of each task τ_i, denoted as L_{τ_i}, and end-to-end latency of cause-effect chain CEC_i, denoted as L_{CEC_i}, as tightly as possible.

II. Proposed Solution Technique for Challenge 1

Since memory access delay is ignored in challenge 1, we compute the worst-case response time of a task τ_i, considering the execution times only.

A. End-to-end latency of a preemptive task

If a higher priority task is released during the execution of a preemptive task $\tau_c \in S_P$, it is preempted by all runnables in the higher priority task. Thus we can formulate the upper bound of the latency between the release time of a runnable $r_{c,i}$ to the finish time of a runnable $r_{c,j}$, denoted $UBL^f(r_{c,i}, r_{c,j})$ where $1 \le i \le j \le |\tau_c|$, as follows using the response time analysis:

$$UBL^f(r_{c,i}, r_{c,j}) = \sum_{k=i}^{j} c^u_{c,k} + \sum_{\tau_h \in hp(\tau_c)} \left(\left\lceil \frac{UBL^f(r_{c,i}, r_{c,j})}{p^l_h} \right\rceil \cdot \sum_{k=1}^{|\tau_h|} c^u_{h,k} \right) \quad (1)$$

where $hp(\tau_c) = \{\tau_h | m_h = m_c, c > h\}$ is a set of higher priority tasks. Then the estimated end-to-end latency of a preemptive task τ_c becomes $L_{\tau_c} = UBL^f(r_{c,1}, r_{c,|\tau_i|})$.

B. End-to-end latency of a cooperative task

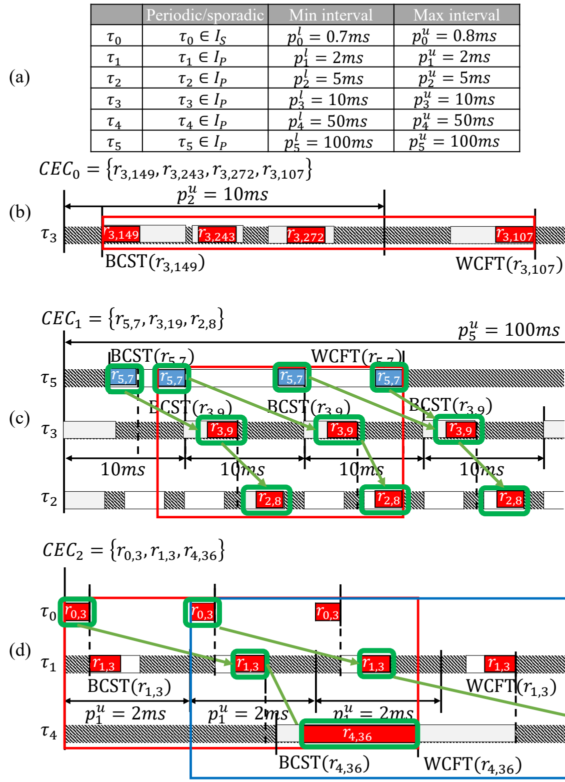
For a cooperative task τ_c, the release of τ_c can be blocked by at most one runnable execution of a lower priority task mapped on the same core. Higher priority cooperative tasks released after the start time of a runnable $r_{c,j}$ have no effect on the finish time. We formulate the upper bound of the latency between release time of a runnable $r_{c,i}$ to start time of a runnable $r_{c,j}$, denoted $UBL^s(r_{c,i}, r_{c,j})$ where $1 \le i \le j \le |\tau_c|$ as follows:

$$UBL^s(r_{c,i}, r_{c,j}) = \left(i = 1 ? \max_{r_{l,k} \in \cup lp(\tau_i)} c^u_{l,k} : 0 \right) + \sum_{k=i}^{j-1} c^u_{c,k} + \sum_{\tau_h \in hp(\tau_c)} \left(\left\lceil \frac{UBL^s(r_{c,i}, r_{c,j}) + 1}{p^l_h} \right\rceil \cdot \sum_{k=1}^{|\tau_h|} c^u_{h,k} \right) \quad (2)$$

where $lp(\tau_c) = \{\tau_l | m_l = m_c, c < l\}$ is a set of lower priority tasks mapped on the same core. The first, second, and third terms indicate the maximum blocking from a lower priority task, the sum of maximum execution times of runnables, and the maximum preemptions from higher priority tasks, respectively. Blocking delay is zero when $i \ne 1$ since any lower priority task cannot start after the first runnable starts. Note that $UBL^s(r_{c,i}, r_{c,j}) + 1$ is used in the third term to include the higher priority tasks released between the finish of the $(j-1)$-th runnable and the start of the (j)-th runnable. Then $UBL^f(r_{c,i}, r_{c,j})$ can be estimated as follows:

$$UBL^f(r_{c,i}, r_{c,j}) = \left(i = 1 ? \max_{r_{l,k} \in \cup lp(\tau_i)} c^u_{l,k} : 0 \right) + \sum_{k=i}^{j} c^u_{c,k} + \sum_{\tau_h \in \in hp(\tau_c) \cap S_P} \left(\left\lceil \frac{UBL^f(r_{c,i}, r_{c,j})}{p^l_h} \right\rceil \cdot \sum_{k=1}^{|\tau_h|} c^u_{h,k} \right) + \sum_{\tau_h \in hp(\tau_c) \cap S_C} \left(\left\lceil \frac{UBL^s(r_{c,i}, r_{c,j})}{p^l_h} \right\rceil \cdot \sum_{k=1}^{|\tau_h|} c^u_{h,k} \right) \quad (3)$$

All requests of higher priority preemptive tasks within $UBL^f(r_{c,i}, r_{c,j})$ are accounted in the third term while the requests of higher priority cooperative tasks after $r_{c,j}$ starts are excluded. Then the estimated worst-case response time of a cooperative task τ_c becomes $L_{\tau_c} = UBL^f(r_{c,1}, r_{c,|\tau_i|})$.



	Periodic/sporadic	Min interval	Max interval
τ_0	$\tau_0 \in I_S$	$p_0^l = 0.7ms$	$p_0^u = 0.8ms$
τ_1	$\tau_1 \in I_P$	$p_1^l = 2ms$	$p_1^u = 2ms$
τ_2	$\tau_2 \in I_P$	$p_2^l = 5ms$	$p_2^u = 5ms$
τ_3	$\tau_3 \in I_P$	$p_3^l = 10ms$	$p_3^u = 10ms$
τ_4	$\tau_4 \in I_P$	$p_4^l = 50ms$	$p_4^u = 50ms$
τ_5	$\tau_5 \in I_P$	$p_5^l = 100ms$	$p_5^u = 100ms$

Fig. 3. End-to-end lantency computation of three example cause-effect chains. A white box indicates the schedule time bound of a runnable while a red or a blue box indicates an execution time of the runnable.

C. End-to-end latency of a cause-effect chain

In this section, we compute the end-to-end latency of a cause-effect chain. Before explaining the latency computation, we define two variables $BCST(r_{c,i})$ and $WCFT(r_{c,i})$ which mean a lower bound of start time of $r_{c,i}$ and an upper bound of finish time of $r_{c,i}$ respectively. $WCFT(r_{c,i})$ is formulated as $WCFT(r_{c,i}) = UBL^f(r_{c,1}, r_{c,i})$. Since a cooperative task is not blocked by a low priority task in the best case, $BCST(r_{c,i})$ for either a preemptive task or a cooperative task can be formulated in the same way:

$$BCST(r_{c,i}) = \sum_{k=1}^{i-1} c_{c,k}^l + \sum_{\tau_h \in hp(\tau_c)} \left(\left\lceil \frac{\max(0, BCST(r_{c,i}) - \delta_h + 1)}{p_h^u} \right\rceil \cdot \sum_{k=1}^{|\tau_h|} c_{h,k}^l \right) \tag{4}$$

where $\delta_h = p_h^u - \sum_{k=1}^{|\tau_h|} c_{h,k}^l$. For each higher priority task, the maximum initiation interval and the minimum execution times are considered to compute the minimum interference.

A cause-effect chain is defined by a sequence of runnables that have read/write dependency over a label between each pair of runnables. Fig. 3 shows three example cause-effect chains and the activation patterns of five tasks are summarized in Fig. 3 (a). A cause-effect chain CEC_0 in Fig. 3 (b) consists of four runnables in the same task τ_3. In this case, we have to analyze how many task instances are involved in the chain. If the (i+1)-th runnable of the chain has a smaller index than the i-th runnable, labels written by the i-th runnable will be read by the (i+1)-th runnable in the next task instance. Hence the number of the instances involved in the chain is computed by counting how many times runnable indices decrease in the task sequence. In Fig. 3 (b), two task instances are involved in the chain since index decrease appears only once in the chain ($r_{3,272} \rightarrow r_{3,107}$). If one task instance covers the cause-effect chain, the end-to-end latency can be computed as $UBL^f(r_{c,b}, r_{c,e})$ where $r_{c,b}$ and $r_{c,e}$ are the first and the last τ_c runnables in the chain. Otherwise, the worst-case end-to-end latency becomes the distance from the BCST of the first runnable to the WCFT of the last runnable plus the task period multiplied by the count of index decreases in the chain, which gives $p_3^u + WCFT(r_{3,107})$ - $BCST(r_{3,149})$ for the example of Fig. 3 (b).

A cause-effect chain CEC_1 in Fig. 3 (c) consists of three runnables with different activation patterns. In this case, we consider the schedule time bound of the first runnable ($BCST(r_{5,7})$, $WCFT(r_{5,7})$) and examine all possible BCSTs of the second runnable $r_{3,19}$ that may appear after the first runnable. In the example of Fig. 3 (c), there are three possible BCSTs of $r_{3,19}$. If we consider a pair of runnables only, the worst-case scenario is that the second runnable starts just before the first runnable finishes and the label written by the first runnable is read by the second runnable at the latest in the next task instance. Based on this observation we define a set of starting points of the first runnable as shown in blue color in the figure. The set includes the schedule of the first runnable whose finish time coincides with a possible BCST of the second runnable as well as the earliest and the latest schedule within the schedule bound.

For the subsequent pair of runnables, for instance the second and the third runnables in the example of Fig. 3 (c), we need to consider the schedule time bound of the successor and the WCFT of the predecessor. If the WCFT of the predecessor lies in the schedule time bound of the successor, the label written by the predecessor should be read by the successor runnable at the latest in the next task instance. For each candidate starting point of the first runnable in the chain, the figure shows the longest cause-effect chain by green arrows where red and blue boxes mean the executions of runnables. Among all candidate starting points, we find one that gives the worst-case chain latency that is represented by a red bounding box in the figure, which corresponds to the second candidate starting point.

In this example, we consider a single runnable involved in each task. In case more than one runnable of the same task is included in the chain, we group them as a sub-chain. Then, a cause-effect chain consists of a sequence of sub-chains where each sub-chain consists of a set of runnables in the same task. If the worst-case latency of the sub-chain spans more than one task instance like the case of Fig. 3 (b), we need to consider only one starting point for the sub-chain for the second case.

The third case shown in Fig. 3 (d) is the case that the cause-

effect chain starts with a sporadic task: the first runnable in CEC_2 belongs to a sporadic task τ_0. Since the sporadic task may start anytime, we find the worst-case scenario in which the finish time of $r_{0,3}$ is aligned with the best case start time of the first $r_{1,3}$ instance. Then the end-to-end latency from $r_{0,3}$ to $r_{1,3}$ is bounded by $UBL^f(r_{0,3}, r_{0,3}) + p_1^u + WCFT(r_{1,3}) - BCST(r_{1,3})$. Note that we need to check only one starting point, which makes the finish time of the sub-chain be aligned with the best case start time of next sub-chain, unlike the case of periodic tasks in Fig. 3 (c). We repeat this computation for all task instances of the first periodic task in the chain within the hyper-period of tasks. In Fig. 3 (d), τ_1 is the first periodic task. If we repeat computation for all τ_1 instances, the maximum latency occurs with the third τ_1 instance since labels written by the third $r_{1,3}$ instance is missed by the first $r_{4,36}$ instance.

Algorithm 1 Algorithm to compute the end-to-end latency of a cause-effect chain

```
1: E2E ← 0, d_len ← 0
2: if the first sub-chain is in a sporadic task then
3:     d_len ←end-to-end latency of the first sub-chain
4:     while all sporadic sub-chains before the first periodic sub-chain do
5:         d_len ← d_len + (one period) + (WCFT of the last runnable) −
   (BCST of the first runnable)
6:     end while
7: end if
8: for all instances of the first periodic sub-chain within hyperperiod do
9:     find all candidate starting points of the first runnable
10:    for all candidate starting points do
11:        start ← (candidate starting point)
12:        end ←corresponding end point
13:        for all sub-chains after the first periodic sub-chain do
14:            if sub-chain is in a sporadic task then
15:                end ← end + (one period) + (WCFT of the last
   runnable) − (BCST of the first runnable)
16:            else
17:                end ←minimum WCFT among runnable instances whose
   BCST is no smaller than end
18:            end if
19:        end for
20:        E2E ← max(E2E, end − start)
21:    end for
22: end for
23: return E2E + d_len
```

Now we summarize the proposed technique for the estimation of the end-to-end latency of a cause-effect chain with Algorithm 1. At first, if the chain starts with sporadic tasks, we compute the end-to-end latency d_len of those sporadic sub-chains (lines 2-7). Then for the first periodic sub-chain, we examine all instances of the first periodic sub-chain within the hyperperiod of the chain. (lines 8-23). For each instance, we find all candidate starting points and compute the latency from the starting point to the end time of the chain (lines 9-21). If the chain starts with a sporadic task or a sub-chain that spans more than one task instance, we need to consider only one starting point which is the BCST of the runnable. Otherwise, we find all candidate starting points as Fig. 3 (c). From each starting point, we find the end point of the chain (lines 13-19).

III. Proposed Solution Technique for Challenge 2

In the second challenge, we consider the worst-case memory access delay in the latency computation. Since memory accesses are arbitrated according to the FIFO policy and a core is assumed to be blocked during memory access, one memory access may be delayed by at most three accesses (one per each core). Hence a naive way to find a conservative upper bound is to assume that each access experiences blocking by three queued accesses. To find a tighter bound of memory access delay, however, we analyze the maximum number of memory accesses issued by tasks in each core within any time window of size Δt by adopting the event stream model [6]. Then we can bound the number of memory accesses that are issued from remote cores. For example, if there are total 10 accesses during the worst-case response time of a task τ_i, L_{τ_i}, and all accesses are assumed to be blocked by three accesses, the total memory access delay will be $10 \cdot (8+3+1)$ cycles. If we know that some cores cannot issue more than 10 accesses within any time window of size L_{τ_i}, we can tighten the upper bound of memory access delay.

Since we aim to find the maximum number of accesses within a time window, we consider the lower bound of execution time and the lower bound of initiation interval in this section. For brevity, we define a variable $\mathcal{C}_i$ as the sum of best case execution times of all runnables in τ_i plus memory access delay without contention.

For a given time window of size Δt, we have to compute the maximum memory access requests from each core. To tackle this problem, several approaches that find an upper bound of the number of shared resource accesses within a time window have been proposed ([7], [8]). In this paper, we propose an improved technique by accounting for the scheduling pattern of tasks. For each core, we have to find out the task execution scenario that produces the maximum memory access requests within the time window. Since the number of task execution scenarios is enormously large, we consider the partitioning of the time window to tasks in the core. The partitioned time means the net execution time of a task. Note that a task may have multiple task instances in the time window that may not be continuous due to preemption or periodic appearance. Since the total execution time within a time window cannot exceed Δt, we check all combinations of task net execution times. For instance, suppose that there are two tasks in a core and $\Delta t = 3$. Then we check all possible combinations of execution time partitions: (0,3), (1,2), (2,1), and (3,0) where (a,b) means the net execution times of two tasks. If we compute the minimum and the maximum bound of net execution time that a task may take within a time window Δt, we can eliminate the infeasible partitions. If the first task cannot take 3 time units in any time window of size 3, (3,0) becomes impossible. With a given net execution time of a task, we find the upper bound of memory access requests.

At first, we define two functions $t_i^{min}(\Delta t)$ and $t_i^{max}(\Delta t)$ that represent the minimum and the maximum execution time amount a task τ_i may take within a time window Δt,

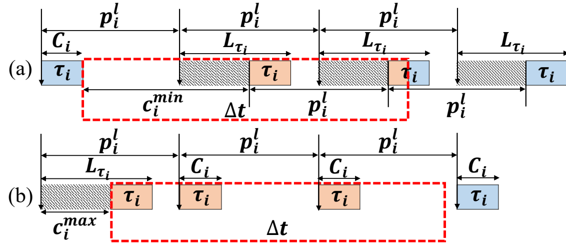


Fig. 4. The minimum execution time scenario (a) and the maximum execution time scenario (b) in a time window Δt

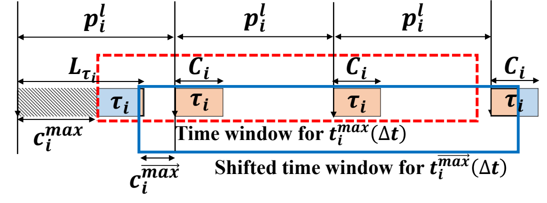


Fig. 5. The shifted time window by $\mathcal{C}_i - 1$ from the time window for $t_i^{max}(\Delta t)$

respectively. Fig. 4 illustrates two scheduling patterns of task τ_i that correspond to $t_i^{min}(\Delta t)$ and $t_i^{max}(\Delta t)$, respectively. In the figure, a dashed rectangle indicates the time window Δt, and the task is invoked with the minimum initiation interval p_i^l. The start time of a task may be delayed by $L_{\tau_i} - \mathcal{C}_i$ in the worst-case by preemption or memory arbitration delay, which is represented as the grey area in the execution profile.

For a task τ_i to take the minimum *net* execution time in the time window, the worst-case interval between two consecutive job instances should be considered. The worst-case interval is observed when an instance finishes its execution as soon as possible with response time of $\mathcal{C}_i$ and the start times of all subsequent instances are maximally delayed by $L_{\tau_i} - \mathcal{C}_i$ as shown in Fig. 4 (a). Then the minimum *net* execution time is found when the time window starts immediately after the finish time of the first instance. In summary, we can derive the function $t_i^{min}(\Delta t)$ as follows:

$$t_i^{min}(\Delta t) = \mathcal{C}_i \cdot \lfloor \max(0, \Delta t - c_i^{min})/p_i^l \rfloor + \min(\mathcal{C}_i, \max(0, \Delta t - c_i^{min}) \bmod p_i^l) \quad (5)$$

where $c_i^{min} = p_i^l + L_{\tau_i} - 2 \cdot \mathcal{C}_i$. The first term and the second term indicate fully included executions and partially included execution, respectively.

On the contrary, we should consider the shortest interval between two τ_i instances in order to compute the maximum execution time in time window Δt: an instance starts as late as possible to finish at its end-to-end latency L_{τ_i} and subsequent instances start immediately at their request time. The execution time amount is maximized in time window Δt when the time window starts at the start time of the first task instance, as illustrated in Fig. 4 (b). The maximum amount of execution time $t_i^{max}(\Delta t)$ is derived as follows:

$$t_i^{max}(\Delta t) = \min\left(\Delta t, \mathcal{C}_i \cdot \lfloor (\Delta t + c_i^{max})/p_i^l \rfloor + \min(\mathcal{C}_i, (\Delta t + c_i^{max}) \bmod p_i^l)\right) \quad (6)$$

where $c_i^{max} = L_{\tau_i} - \mathcal{C}_i$. The first term and the second term indicate fully included executions and partially included execution, respectively.

Now we compute how many instances may exist in a time window Δt. Fig. 5 shows the same task schedule scenario of Fig. 4 (b) and the dashed rectangle indicates the time window to achieve the maximum execution time $t_i^{max}(\Delta t)$ in a time window Δt. In order to cover the task instances as many as possible in the time window, we shift the time window to the right direction. If the shift amount is greater than or equal to $\mathcal{C}_i$, the first task instance becomes outside of the time window, making the number of instances decreases. Hence the shift amount should be less than $\mathcal{C}_i$. On the other hand, we need to shift the time window as much as possible to include the instances at the right side of the time window. In summary, to make the maximum number of task instances that may lie in the time window, the time window should be shifted by $\mathcal{C}_i - 1$. In the figure, the shifted rectangle contains one more instance of the task than the dashed rectangle.

Note that when the number of task instances laid in the time window is maximized, the *net* execution time may be smaller than that for the case when the net execution time in the time window is maximized. Hence we need to compare two cases to find the maximum possible resource demand; (1) the case the number of task instances is maximized and (2) the case the net execution time is maximized. Then we need to compute the maximum *net* execution time for the first case and the maximum number of task instances for the second case. The number of instances for the second case can be computed as $n_i(\Delta t) = \left\lceil \frac{t_i^{max}(\Delta t)}{\mathcal{C}_i} \right\rceil$. We denote the maximum *net* execution time for the first case $\overrightarrow{t_i^{max}}(\Delta t)$, where arrow indicates that the time window is shifted by $\mathcal{C}_i - 1$ to maximize the number of instances. Then $\overrightarrow{t_i^{max}}(\Delta t)$ can be formulated as follows:

$$\overrightarrow{t_i^{max}}(\Delta t) = \mathcal{C}_i \cdot \lfloor \max(0, \Delta t - \overrightarrow{c_i^{max}})/p_i^l \rfloor + \min(\mathcal{C}_i, \max(0, \Delta t - \overrightarrow{c_i^{max}}) \bmod p_i^l) \quad (7)$$

where $\overrightarrow{c_i^{max}} = p_i^l - L_{\tau_i}$. We denote the maximum number of τ_i instances laid in the time window Δt as $\overrightarrow{n_i}(\Delta t)$ and $\overrightarrow{n_i}(\Delta t)$ can be computed from $\overrightarrow{t_i^{max}}(\Delta t)$ to be $\left\lceil \frac{(\overrightarrow{t_i^{max}}(\Delta t)-1)}{\mathcal{C}_i} \right\rceil + 1$.

Finally, we formulate memory access bound function $D_{A_i^C, A^G}(\Delta t)$ which finds the maximum number of accesses from a core A_i^C to global memory A^G within any time window of size Δt. When we distribute the time amount Δt to tasks mapped onto A_i^C, we should consider the constraint that a task τ_k can be assigned the bounded *net* execution time t_k between $t_k^{min}(\Delta t)$ and $t_k^{max}(\Delta t)$. And, for a given Δt, we consider two cases where the number of instances of a task τ_k is $n_k(\Delta t)$ or $\overrightarrow{n_k}(\Delta t)$ as the access bound function of each individual task, $D_{\tau_k, A^G}(t_k, \Delta t)$. In summary, the memory access bound

function is formulated as follows:

$$D_{A_i^C, A^G}(\Delta t) = \max\left\{ \sum_{m_k = A_i^C} D_{\tau_k, A^G}(t_k, \Delta t) \;\middle|\; \begin{array}{c} \sum_{m_k=A_i^C} t_k = \Delta t, \\ \forall_{m_k=A_i^C} t_k \geq t_k^{min}(\Delta t) \end{array} \right\} \quad (8)$$

$$D_{\tau_k, A^G}(t_k, \Delta t) = \max \begin{pmatrix} \eta_{k,A^G}^{e(n_k(\Delta t))}(\min(t_k, t_k^{max}(\Delta t))) \\ \eta_{k,A^G}^{e(\overrightarrow{n_k}(\Delta t))}(\min(t_k, \overrightarrow{t_k^{max}}(\Delta t))) \end{pmatrix} \quad (9)$$

where $\eta_{k,A^G}^n(t)$ is the maximum number of resource accesses that may be issued from n instances of a task τ_k to a memory A^G when the net execution time of τ_i does not exceed t time units. $\eta_{k,A^G}^n(t)$ can be computed by moving the time window of size t on the n task instances that are executed one after another and finding the maximum number of resource accesses among all time windows.

$D_{A_i^C, A^G}(\Delta t)$ can be obtained by the max-plus convolution of individual demand bound functions of (9) in polynomial time since the max-plus convolution has associative property and commutative property. $D_{A_i^C, A^G}(\Delta t)$ is used to bound the arbitration delay during the latency computed by equations (1), (2), and (3). For a preemptive task, it may be blocked by one memory access from a lower priority task. For a cooperative task, we consider the maximum blocking by one lower priority runnable with its worst case memory access delay (8+3+1 per one access). This blocking delay is independently computed and included in the worst-case latency. After computing the lower priority blocking delay, we consider memory accesses that are issued from the target task and higher priority tasks. To bound the interference from a core, we compute the number of memory accesses from the core during the latency of interest.

IV. Proposed Solution Technique for Challenge 3

In this section, we propose a greedy algorithm that determines a label-to-memory mapping to optimize the end-to-end latencies. If a label is mapped to a local memory A_i^L, we can save the crossbar transfer delay (8 cycles) which is larger than the worst-case arbitration delay (4 cycles).

Algorithm 2 presents a pseudo code of the proposed greedy algorithm to determine label-to-memory mapping. Initially labels are mapped to a global memory A^G (line 8). At first, we compute each fitness value of a mapping of $L[i]$ to A_j^L, $F[i][j]$ (line 9). The fitness value is higher if $L[i]$ is more frequently accessed from A_j^C. Then we determine a mapping of each label (lines 11-18). We select the most beneficial mapping according to the fitness values (line 12). Since we assume a limited local memory size, the label $L[l]$ can be mapped to A_m^L in case A_m^L has enough memory size (lines 13-16). The progress is repeated until there is no mapping that optimize the memory access delay (line 11).

Unlike the latency computation in challenge 2, the memory accesses from A_i^C to A_i^L do not involve transfer delay so that only arbitration delay at the memory should be considered. The technique to compute memory arbitration delay bound explained in challenge 2 can be easily extended to compute the memory access bound function of each memory separately.

Algorithm 2 Greedy algorithm to determine label-to-memory mapping

Input: a set of labels L, an array of label sizes S_L and local memory size s
Output: an array of label mapping M
1: $S_M \leftarrow$ one dimensional array of size 4
2: $F \leftarrow$ two dimensional array of size $|L| \times 4$
3: ▷ $S_L[i]$ is a label size of $L[i]$
4: ▷ $M[i]$ is a memory a label $L[i]$ is mapped to
5: ▷ $S_M[i]$ indicates available memory size of A_i^L
6: ▷ $F[i][j]$ is a fitness value of a mapping of $L[i]$ to A_j^L
7: **for** $0 \leq i < |L|, 0 \leq j < 4$ **do**
8: $M[i] \leftarrow A^G$, $S_M[j] \leftarrow s$
9: $F[i][j] \leftarrow \sum_{m_k=A_j^C} \frac{\#\text{accesses of } \tau_k \text{ to } L[i]}{p_k^l}$
10: **end for**
11: **while** $\exists_{i,j} F[i][j] > 0$ **do**
12: find indices l and m that $F[l][m] = \max_{i,j} F[i][j]$
13: **if** $S_M[m] \geq S_L[l]$ **then**
14: $S_M[m] \leftarrow S_M[m] - S_L[l]$
15: $M[l] = A_m^L$
16: **end if**
17: $F[l][m] \leftarrow 0$
18: **end while**

V. Challenge Results

The estimated end-to-end latencies of all tasks and cause-effect chains from the proposed technique are summarized in Table. I. In the table, WCRT and E2E L. mean the worst-case response time and the end-to-end latency, respectively. (C1), (C2), and (C3) columns show the estimated results for challenge 1, challenge 2, and challenge 3, respectively. We assume unlimited local memory size in the experiment since no constraints are given.

Even without memory access delay, 6 out of 21 tasks in the challenge model are unschedulable according to our analysis results since core utilizations are too high: utilizations are 97%, 133.5%, 106.8%, and 117.9% for each core. There is even a task that has the worst-case execution time larger than its deadline (Task_10ms). End-to-end latencies of cause-effect chains cannot be analyzed due to the runnables in unschedulable tasks. We claim that the worst-case execution time should be decreased to make the system schedulable.

Results show that the portion of the memory access delay in the worst-case response time is not significant. Task_50ms becomes unschedulable when memory access delay is not ignored. Because of the memory access delay, its worst-case response time becomes over 8,000,000 and one more preemption of Task_20ms whose worst-case execution time is 2,093,688 occurs, making the response time larger than the deadline. Almost all read/write accesses go to local memory after label-to-memory mapping is done so that the memory access delay decreases accordingly. Task_20ms is barely schedulable after label-to-memory mapping.

We conducted additional experiment to find maximum execution times of tasks satisfying all task deadlines. For each core, we scale down all worst-case execution times of mapped

TABLE I
END-TO-END LATENCIES OF TASKS AND CAUSE-EFFECT CHAINS SPECIFIED IN THE PROVIDED SYSTEM MODEL (UNIT: CYCLE)

Task		WCRT (C1)	WCRT (C2)	WCRT (C3)
CORE0	ISR_10 (τ_0)	6,068	6,308	6,112
	ISR_5 (τ_1)	57,704	58,256	57,785
	ISR_6 (τ_2)	63,894	64,698	63,996
	ISR_4 (τ_3)	137,054	138,278	137,206
	ISR_8 (τ_4)	261,725	263,843	261,973
	ISR_7 (τ_5)	530,598	534,453	531,061
	ISR_11 (τ_6)	853,378	859,207	854,081
	ISR_9 (τ_7)	unschedulable	unschedulable	unschedulable
CORE1	Task_1ms (τ_{11})	152,870	156,345	153,588
	Angle_Sync (τ_{12})	unschedulable	unschedulable	unschedulable
CORE2	Task_2ms (τ_{13})	80,817	82,425	81,188
	Task_5ms (τ_{14})	267,180	270,252	267,900
	Task_20ms (τ_{16})	3,709,404	3,760,278	3,719,254
	Task_50ms (τ_{17})	7,973,611	unschedulable	7,992,287
	Task_100ms (τ_{18})	unschedulable	unschedulable	unschedulable
	Task_200ms (τ_{19})	unschedulable	unschedulable	unschedulable
	Task_1000ms (τ_{20})	unschedulable	unschedulable	unschedulable
CORE3	ISR_1 (τ_8)	7,011	7,383	7,066
	ISR_2 (τ_9)	10,560	11,160	10,635
	ISR_3 (τ_{10})	15,347	16,247	15,448
	Task_10ms (τ_{15})	unschedulable	unschedulable	unschedulable

Cause-effect chain	E2E L. (C1)	E2E L. (C2)	E2E L. (C3)
EffectChain_1	unschedulable	unschedulable	unschedulable
EffectChain_2	unschedulable	unschedulable	unschedulable
EffectChain_3	17,817,190	unschedulable	17,835,552

TABLE II
SCALED WORST-CASE EXECUTION TIMES FOR SCHEDULABLE SYSTEM AND END-TO-END LATENCIES (UNIT: CYCLE)

Task		WCET	WCRT	Deadline
CORE0 (96%)	ISR_10 (τ_0)	5,825	5,867	140,000
	ISR_5 (τ_1)	49,570	55,472	180,000
	ISR_6 (τ_2)	5,942	61,434	220,000
	ISR_4 (τ_3)	70,233	131,706	300,000
	ISR_8 (τ_4)	58,345	251,485	340,000
	ISR_7 (τ_5)	62,375	509,791	980,000
	ISR_11 (τ_6)	58,729	814,042	1,000,000
	ISR_9 (τ_7)	71,133	896,985	1,200,000
CORE1 (71%)	Task_1ms (τ_{11})	108,537	109,164	200,000
	Angle_Sync (τ_{12})	540,360	1,197,321	1,332,000
CORE2 (93%)	Task_2ms (τ_{13})	75,159	75,511	400,000
	Task_5ms (τ_{14})	173,317	249,170	1,000,000
	Task_20ms (τ_{16})	1,947,129	3,383,696	4,000,000
	Task_50ms (τ_{17})	573,714	7,358,075	10,000,000
	Task_100ms (τ_{18})	1,751,743	19,908,947	20,000,000
	Task_200ms (τ_{19})	25,758	19,933,318	40,000,000
	Task_1000ms (τ_{20})	25,511	19,958,468	200,000,000
CORE3 (83%)	ISR_1 (τ_8)	5,819	5,869	1,900,000
	ISR_2 (τ_9)	2,945	8,834	1,900,000
	ISR_3 (τ_{10})	3,973	12,832	1,900,000
	Task_10ms (τ_{15})	1,944,313	1,986,787	2,000,000

Cause-effect chain	E2E Latency
EffectChain_1	2,269,514
EffectChain_2	2,628,493
EffectChain_3	13,888,054

tasks by the same percentage and find the maximum percentages that make all estimated end-to-end latencies of tasks below deadlines. Table II summarizes the scaled worst-case execution times and the end-to-end latencies for challenge 3. Note that the percentage decrease for each core is proportional to the utilization of the core.

VI. CONCLUSION

We present a solution technique to FMTV 2016 verification challenges, combining the response time analysis and schedule time bound analysis. The main contribution is that we consider schedule time bounds of runnables to tightly compute end-to-end latencies of cause-effect chains. Memory access bound functions are described to find the maximum possible arbitration delay with arrival curve analysis. A simple greedy algorithm is proposed to determine label-to-memory mapping. It took about one month to understand the challenge model and to solve the problem, applying the technique we have developed beforehand.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(NRF-2013R1A2A2A01067907) and MSIP(Ministry of Science, ICT&Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2015-H8501-15-1005) supervised by the IITP(Institute for Information&communications Technology Promotion). The ICT at Seoul National University provides research facilities for this study.

REFERENCES

[1] *AMALTHEA: An Open Platform for Embedded Multicore Systems*. [Online]. Available: http://www.amalthea-project.org/

[2] *2016 Formal Methods for Timing Verification (FMTV) challenge, co-located with the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. [Online]. Available: https://waters2016.inria.fr/challenge/

[3] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmark for free," in *sixth International Workshop on Analysis Tools and Methodologies for Embedded Real-time Systems (WATERS)*, 2015.

[4] *AUTOSAR Specification of RTE Software*. [Online]. Available: https://www.autosar.org/fileadmin/files/releases/2-0/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf

[5] *AUTOSAR Specification of Timing Extensions*. [Online]. Available: https://www.autosar.org/fileadmin/files/releases/4-1/methodology-templates/templates/standard/AUTOSAR_TPS_TimingExtensions.pdf

[6] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, Mar 2005.

[7] M. Negrean, S. Schliecker, and R. Ernst, "Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, April 2009, pp. 524–529.

[8] S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 22:1–22:27, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1880050.1880058

FMTV 2016: Where is the Actual Challenge?

Alessio Balsini, Alessandra Melani, Pasquale Buonocunto, Marco Di Natale
Scuola Superiore Sant'Anna, Pisa, Italy
E-mail: {alessio.balsini, alessandra.melani, pasquale.buonocunto, marco.dinatale}@sssup.it

***Abstract*—The FMTV challenge has been formulated and proposed to research groups as a case study and benchmark to compare different analysis methods for real-time multicore fuel injection applications. The nature of the problem is clear enough and the challenge can be likely met by a set of conventional analysis techniques (at least at the current level of description). However, the formulation of the problem and its practical solution are more than likely to reveal a number of additional issues that go from the model of the application, to analysis techniques that consider with much better precision the details of the HW platform, to the need for synthesis and optimization methods.**

I. Introduction

The FMTV 2016 challenge consists of a timing analysis problem in which the AUTOSAR model of a set of cooperating tasks in a fuel injection application is deployed onto a 4-core platform. The objective of the challenge is to apply different analysis methods (worst-case, simulation-based and possibly stochastic) to models of the system with an increasing level of accuracy with respect to the memory placement of communication variables. At the simplest level, memory access times are simply neglected; next, different access times are assumed under the hypothesis of global or local memory allocation; and, finally, the problem of optimizing the placement of the memory items is presented.

True to the spirit of the description, we tackled the objectives of the challenge in a sequence, and because of timing constraints, at the time of this submission, only the results for some of the early activities were available. However, we believe that in the case of this challenge, the experience gathered along the path is at least as valuable as the final solution, and we found several issues that are worth discussing, beyond the presentation of the tool architecture that was used to derive the solution and the hard data that we computed as a result of the analysis.

From the architecture standpoint, we attempted two solutions to the problem: to simulate the time behavior using a scheduling simulator that was previously available at our laboratory, and to analyze the task-set for its worst-case behavior, using a set of formulas derived from the problem description and obtained by adaptation of classical results.

We provide the results of these two analysis methods (with an additional discussion on how to tackle the memory access time problem), but we also believe several issues are worth discussing. Among those:

The **definition of response times when the system contains chains of tasks or runnables communicating asynchronously**. The challenge refers to a set of definitions (reactive and age) for which an application-level justification is not clear enough and for which (despite being formally presented in [1]) a solution in analytical closed form or as an algorithm has never been presented and validated in a peer-reviewed paper.

Next, while the challenge has the merit of restoring to the foreground the consideration of hardware features and issues, its **description of the HW architecture details** is still incomplete and simplistic. For example, the FIFO arbiter controlling accesses to shared memory is likely to be integrated within the crossbar or possibly placed after it, but this information can only be guessed and would affect the access times to memory.

Finally, and most important, the problem probably placed too much emphasis on the analysis part and seems to neglect **the runnables placement problem**, which is most likely the most relevant design issue for a system like this.

II. System Model and Notation

The challenge model is in large part compliant with the AUTOSAR metamodel and adopts from it definitions and most of the semantics for activation and communication of functions (runnables in AUTOSAR). An attempt at the formal characterization of the challenge model is the following.

A task τ_i is composed of an ordered sequence of n_i runnables $\rho_{i,1}, \ldots, \rho_{i,n_i}$, each of which has its execution time defined as a statistical distribution $\mathcal{C}_i$, which is defined as a truncated Weibull distribution for most if not all the runnables in the model. For the purpose of worst-case analysis, the worst-case execution time (WCET) $C_{i,j}$ and a best-case execution time $c_{i,j}$ may be computed from the distribution $\mathcal{C}_i$.

The scheduling of each task is also controlled by its scheduling mode (cooperative or preemptive) and its priority π_i, with preemptive tasks having higher priority than cooperative tasks, and cooperative tasks only preempting each other at runnable boundaries.

The model also defines deadlines that apply to tasks and task chains. For tasks, deadlines bound the worst case completion time with respect to the activation and match the common definition of a relative deadline D_i. Also, all tasks are assumed to be periodic or sporadic, with a period or a minimum inter-arrival time T_i. When applicable, relative deadlines are constrained to be smaller than or equal to periods, i.e., $D_i \leq T_i$. In the end, we assume each task is defined by a tuple (C_i, c_i, D_i, T_i), where $C_i = \sum_{j=1}^{n_i} C_{i,j}$, $c_i = \sum_{j=1}^{n_i} c_{i,j}$.

We denote as $R_{i,j}$ the worst-case response time of the jth runnable of task τ_i, while $r_{i,j}$ denotes its best-case response time. $hp^P(i)$ and $hp^C(i)$ denote the set of preemptive and cooperative tasks, respectively, having priority greater than τ_i. We denote as $hp(i) = hp^P(i) \cup hp^C(i)$ the union of the two disjoint sets.

As for end-to-end chains, the assumed model is based on the asynchronous propagation of information by means of shared data variables. These variables (labels in the model) are read and written by the runnables.

Figure 1 illustrates the three effect chains that are analyzed in the context of the challenge. Note that, in the third chain, we replaced Label 2197 with Label 646 to fix a mistake in the model (Label 2197 is not read nor written by the last two runnables in the chain, while Label 646 is the only one that satisfies the read/write relation imposed by the chain).

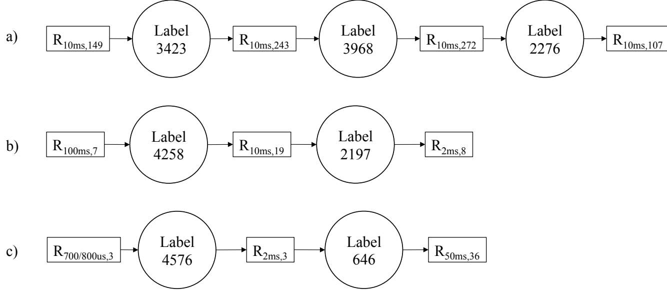


Fig. 1: Effect chains in the model.

The following semantics have been considered for end-to-end latency calculation (from [1]):

- *Last-to-First* (L2F): it considers the delay between the last input that is not overwritten until the first output generated with the same input;
- *First-to-First* (F2F) or *Reactive*: it considers the delay between the first input that may be overwritten until the first output generated with the next different input;
- *Last-to-Last* (L2L) or *Maximum Age*: it considers the delay between the last input that is not overwritten until the last output, considering duplicates.

The problem with this definition is that it is hardly formal, and even in the original reference there seems to be no single point in which a formal definition appears. Hence, we used the following definitions.

Assume a chain of periodic communicating runnables $\Gamma = \{\rho_1, \rho_2, \ldots \rho_n\}$. Also, assume $a_{j,h}$ denotes the h-th activation of runnable ρ_j, $f_{j,h}$ its finishing time, and $I_{j,h}$, $O_{j,h}$ are the sets of input and output values that are respectively read from and written to the labels accessed by the h-th instance of ρ_j.

Then, the L2F latency of the chain Γ is the maximum value $f_{n,r} - a_{1,p}$ (finishing time of the r-th instance of ρ_n minus the activation time of the p-th instance of ρ_1), such that for some p, q, r:

$$\forall i = 1, \ldots, (n-2) \;\; O_{i,p} = I_{i+1,q} \;\text{ and }\; O_{i+1,q} = I_{i+2,r} \\ \text{and } \; I_{i+1,q} \neq I_{i+1,q-1} \;\text{ and }\; I_{i+2,r} \neq I_{i+2,r-1}.$$

Similarly, the F2F latency of the chain Γ is the time interval between the latest $a_{1,p}$ and the earliest $f_{n,r+1}$ such that for some p, q, r:

$$\forall i = 1, \ldots, (n-2) \;\; O_{i,p} = I_{i+1,q} \;\text{ and }\; O_{i+1,q} = I_{i+2,r} \\ \text{and } \; I_{i+1,q} \neq I_{i+1,q+1} \;\text{ and }\; I_{i+2,r} \neq I_{i+2,r+1}.$$

Finally, the L2L latency of the chain Γ is the maximum value $f_{n,r} - a_{1,p}$ (finishing time of the r-th instance of ρ_n minus the activation time of the p-th instance of ρ_1), such that for some p, q, r:

$$\forall i = 1, \ldots, (n-2) \;\; O_{i,p} = I_{i+1,q} \;\text{ and }\; O_{i+1,q} = I_{i+2,r}.$$

Figures 2 and 3 exemplify the definitions in the case of undersampling and oversampling effects, respectively. In particular, referring to the chain $\{\rho_1, \rho_2, \rho_3\}$ in Figure 2, the end-to-end delay by the L2F semantics corresponds to the time interval between the activation $a_{1,1}$ and the finishing time of the runnable activated at time $a_{3,1}$; the end-to-end delay by F2F corresponds to the time interval $[a_{1,1}, f_{3,2}]$. By L2L, it is measured as for the L2F semantics (i.e., $f_{3,1} - a_{1,1}$). In case of oversampling (Figure 3), the end-to-end delay can be measured by the L2F semantics as $f_{3,2} - a_{1,1}$; by F2F it is $f_{3,5} - a_{1,1}$, while the L2L semantics accounts for the same data read by multiple runnable instances (e.g., in the time interval $f_{3,4} - a_{1,1}$).

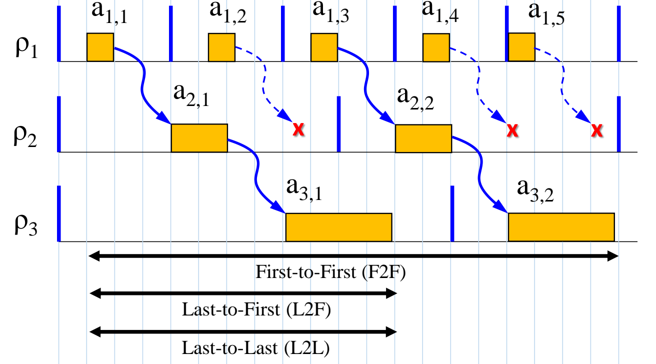


Fig. 2: End-to-end delay in the case of undersampling.

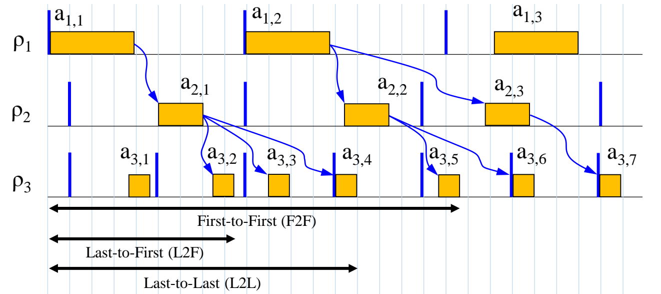


Fig. 3: End-to-end delay in the case of oversampling.

The definition ambiguity leaves open a fundamental issue. **What is the actual meaning and relevance (in application terms) of such definitions**?

III. Worst-case Latency Analysis

This section discusses the analytical approach to compute the worst-case response times for tasks and chains, with and without consideration of the timing for the access to (shared and local) memory.

A. Analysis without memory access times

For any *preemptive* task, the worst-case response time of runnable $\rho_{i,j}$ is given by the fixed point iteration of the following formula (starting with $R^0_{i,j} = \sum_{h=1}^{j} C_{i,h}$):

$$R_{i,j} = \sum_{h=1}^{j} C_{i,h} + \sum_{k \in hp(i)} \left\lceil \frac{R_{i,j}}{T_k} \right\rceil C_k. \quad (1)$$

The above formula quantifies the higher-priority interference suffered by $\rho_{i,j}$ by considering the synchronous periodic arrivals of higher-priority tasks.

For *cooperative* tasks, the worst-case response time needs to consider also the blocking time by lower-priority cooperative runnables and the fact that the last runnable does not suffer any preemption by higher-priority cooperative tasks once it has started executing. In addition, by analogy with the limited preemptive scheduling with fixed preemption points [2], it is not enough to compute the response time of the first job after the critical instant. In particular, the computation must be carried out for all jobs $s \in [1, K_i]$ falling within the so called Level-i Active Period L_i, such that $K_i = \left\lceil \frac{L_i}{T_i} \right\rceil$. Therefore, in case of a cooperative task τ_i, we can compute the worst-case finishing time of the sth job of $\rho_{i,j}$ by the fixed point iteration of the following formula:

$$f_{i,j}^s = \sum_{h=1}^{j} C_{i,h} + B_{i,j} + (s-1)C_i + \sum_{k \in hp^P(i)} \left\lceil \frac{f_{i,j}^s}{T_k} \right\rceil C_k + \sum_{k \in hp^C(i)} \left(\left\lfloor \frac{f_{i,j}^s - C_{i,j}}{T_k} \right\rfloor + 1 \right) C_k,$$

where

$$B_{i,j} = \max_{\substack{q \in lp^C(i) \\ h=1,\ldots,n_q}} C_{q,h}$$

represents the maximum blocking time imposed by lower-priority cooperative tasks.

Then, the worst-case response time of $\rho_{i,j}$ can be computed as:

$$R_{i,j} = \max_{s \in [1, K_i]} f_{i,j}^s - (s-1)T_i. \tag{2}$$

Worst-case start time computation. Another quantity of interest for the end-to-end latency computation is the worst-case start time $S_{i,j}$ of runnable $\rho_{i,j}$. The calculation is the same for both the case of preemptive and cooperative tasks, and is given by:

$$S_{i,j} = \epsilon + \sum_{h=1}^{j-1} C_{i,h} + \sum_{k \in hp(i)} \left\lceil \frac{S_{i,j}}{T_k} \right\rceil C_k, \tag{3}$$

where ϵ is an arbitrarily small constant.

Best-case response time computation.

For preemptive tasks, the best-case response time of runnable $\rho_{i,j}$ is [3]:

$$r_{i,j} = \sum_{h=1}^{j} c_{i,h} + \sum_{k \in hp(i)} \left(\left\lceil \frac{r_{i,j}}{T_k} \right\rceil - 1 \right) c_k. \tag{4}$$

For cooperative tasks, a lower-bound on the best-case response time can be computed by considering a zero blocking-time from lower-priority tasks and the minimum amount of interference from higher-priority tasks [3], [4]:

$$r_{i,j} = \sum_{h=1}^{j} c_{i,h} + \sum_{k \in hp^P(i)} \left(\left\lceil \frac{r_{i,j}}{T_k} \right\rceil - 1 \right) c_k + \sum_{k \in hp^C(i)} \left\lfloor \frac{r_{i,j} - c_{i,j}}{T_k} \right\rfloor c_k. \tag{5}$$

B. *End-to-end Latency Calculation*

The end-to-end latencies have been computed according to the semantics reported in Section II. For each chain, we first compute the end-to-end latency by the Last-to-First (L2F) semantics, and then extend it to obtain the latencies by the F2F and L2L semantics.

Last-to-First semantics. The end-to-end latency of chain $\rho_1, \ldots, \rho_N$ according to the L2F semantics can be computed as:

$$\sum_{i=1}^{N-1} (R_i + \min(T_{i+1} - r_{i+1}, T_i)) + R_N. \tag{6}$$

First-to-First semantics. With respect to the L2F semantics, in the F2F semantics we need to add one cycle delay for the first runnable in the chain, in order to consider the previous input. Therefore, the end-to-end latency of chain $\rho_1, \ldots, \rho_N$ according to the F2F semantics can be computed as:

$$T_1 + \sum_{i=1}^{N-1} (R_i + \min(T_{i+1} - r_{i+1}, T_i)) + R_N. \tag{7}$$

Additionally, the F2F semantics considers previous inputs that are overwritten. In order to compute how many times in the worst case an input is overwritten between consecutive stages of the chain (i.e., between runnables ρ_i and ρ_{i+1}), we need to find the largest possible integer $\overline{n} \geq 1$ that satisfies:

$$T_{i+1} + S_{i+1} - r_{i+1} \geq \overline{n} T_i + r_i - R_i. \tag{8}$$

This relation guarantees that the longest interval between two consecutive reads is greater than the shortest interval between $\overline{n}$ consecutive writes. If the above relation holds (i.e., input overwriting takes place), we compute the end-to-end latency of chain $\rho_1, \ldots, \rho_N$ as:

$$T_1 + \sum_{i=1}^{N-1} (R_i + \overline{n} T_i) + R_N. \tag{9}$$

Last-to-Last semantics. With respect to the L2F semantics, the L2L also considers subsequent outputs that are overwritten. In order to compute how many times in the worst case an output is overwritten between consecutive stages of the chain, we need to find the largest possible integer $\widehat{n} \geq 1$ that satisfies:

$$T_i - r_i + R_i \geq \widehat{n} T_{i+1} - r_{i+1} + S_{i+1}. \tag{10}$$

This relation guarantees that the longest interval between two consecutive writes is greater than the shortest interval to perform $\widehat{n}$ consecutive reads. If the above relation holds (i.e., output overwriting takes place), we compute the end-to-end latency as:

$$\sum_{i=1}^{N-1} (R_i + \widehat{n} T_{i+1} - r_{i+1}) + R_N. \tag{11}$$

Otherwise, the end-to-end latency by the L2L semantics is as the one obtained under the L2F semantics.

C. Analysis with memory access and arbitration times

In the proposed model, the four cores contend for access to a shared global memory (GRAM) with FIFO arbitration. Each read/write access to GRAM costs 9 cycles (there is no caching effect). Therefore, in the worst case each memory access might get blocked by pending accesses from other cores, i.e., each access can be delayed for $9(m-1) = 27$ cycles. Adding up the memory access cost for the current request, we obtain a worst-case memory-access penalty of 36 clock cycles. By exploiting the knowledge of how many labels are read/written by each runnable, we can compute the worst-case memory access latency for its read/write phases.

In the best case, memory accesses do not experience any delays from other cores, leading to a best-case memory-access time of 9 clock cycles. Accordingly, we can compute the best-case memory access latency for the read/write phases.

Such values need to be added to the execution time of each runnable, to which the analysis described in Section III-A can be applied identically.

The worst-case estimate of $9(m-1)$ cycles implies that the 9 cycles access cost is repeatedly applied on each FIFO access, which is most likely a pessimistic estimate given the lack of detailed information on the HW (memory) configuration. Careful consideration of the memory access costs **require a model of the execution HW more detailed than what is typically available in scheduling analysis papers**.

D. End-to-end Latency Calculation

The end-to-end latency calculation can be performed as described in Section III-B, with the following differences.

Last-to-First semantics. Equation (6) is replaced by:

$$\sum_{i=1}^{N-1}(R_i - r_{i+1}^{read} + \min(T_{i+1}, T_i)) + R_N, \tag{12}$$

where r_i^{read} denotes the best-case response time of the read phase of ρ_i.

First-to-First semantics. (8) is replaced by:

$$T_{i+1} + S_{i+1} - r_{i+1}^{read} \geq \overline{n}T_i + r_i - R_i. \tag{13}$$

Last-to-Last semantics. (10) is replaced by:

$$T_i - r_i + R_i \geq \widehat{n}T_{i+1} - r_{i+1}^{read} + R_{i+1}^{read}, \tag{14}$$

where R_i^{read} denotes the worst-case response time of the read phase of ρ_i, which can be computed similarly as in Section III-A.

E. Experimental Evaluation

In order to make the system analyzable, the WCETs of those tasks that were not deemed schedulable by our analysis were scaled down by considering the largest scaling factor $\sigma \in (0, 1]$ that guarantees schedulability. In particular, starting from $\sigma = 1$, WCETs are iteratively scaled down in steps of 0.01 until the system becomes schedulable by the proposed analysis. Table I reports the scaling factor σ for each task, and the scaling factor σ^M obtained when memory access and arbitration are accounted for. The analytical approach described in Section III has been implemented in C++, and the code is fully available online [5].

TABLE I: Scaling factors.

Task	Core	σ	σ^{M}	Task	Core	σ	σ^{M}
ISR10	0	1	1	5ms	2	1	1
ISR5	0	1	1	20ms	2	1	1
ISR6	0	1	1	50ms	2	1	0.52
ISR4	0	1	1	100ms	2	0.28	0.12
ISR8	0	1	1	200ms	2	0.49	0.78
ISR7	0	1	1	1000ms	2	0.18	0.15
ISR11	0	1	1	ISR1	3	1	1
ISR9	0	0.58	0.29	ISR2	3	1	1
1ms	1	1	1	ISR3	3	1	1
Angle Sync	1	0.37	0.26	10ms	3	0.84	0.78
2ms	2	1	1				

1) Effect Chain 1: In the effect chain 1: (i) all runnables belong to the same task (*Task_10ms*, allocated to core 3), hence all runnables are bound to the same rate; (ii) there is backward communication between the third and the fourth runnable, which implies a one cycle delay until the last datum is read. Therefore, the worst-case end-to-end latency of this effect chain by L2F can be computed as:

$$L_1^{L2F} = T_{10ms} + R_{10ms,107} = 13376\ \mu s. \tag{15}$$

Given that all runnables belong to the same task, this result is valid also when considering the L2L semantics. As for the F2F semantics, the analysis needs to consider a one cycle delay for the first runnable, that is:

$$L_1^{F2F} = 2T_{10ms} + R_{10ms,107} = 23376\ \mu s. \tag{16}$$

2) Effect Chain 2: Unlike the previous chain, runnables in this chain belong to different tasks with different rates. In this case, the end-to-end latency calculation should also consider the over-sampling effect between pairs of consecutive runnables. By the L2F semantics, applying Equation (6), we obtain:

$$\begin{aligned} L_2^{L2F} &= R_{100ms,7} + \min(T_{10ms} - r_{10ms,19}, T_{100ms}) \\ &+ R_{10ms,19} + \min(T_{2ms} - r_{2ms,8}, T_{10ms}) \\ &+ R_{2ms,8} = 52222\ \mu s \end{aligned}$$

As for the F2F semantics, due to the over-sampling effect, there are no input overwritings (Condition (8) is never verified), hence the end-to-end latency is simply given by:

$$L_2^{F2F} = L_2^{L2F} + T_{100ms} = 152222\ \mu s.$$

Finally, the end-to-end latency computation for the L2L semantics requires to verify Condition (10) for any pair of consecutive runnables. In this case, we obtain $\widehat{n} = 13$ for the first stage and $\widehat{n} = 5$ for the second stage, which yields:

$$\begin{aligned} L_2^{L2L} &= R_{100ms,7} + 13 \cdot T_{10ms} - r_{10ms,19} + R_{10ms,19} \\ &+ 5 \cdot T_{2ms} - r_{2ms,8} + R_{2ms,8} = 180222\ \mu s. \end{aligned}$$

3) Effect Chain 3: Also in this case, runnables belong to different tasks with different rates. Task periods have increasing values, leading to an under-sampling effect.

By the L2F semantics, applying Equation (6), we obtain:

$$\begin{aligned} L_3^{L2F} &= R_{700/800us,3} + \min(T_{2ms} - r_{2ms,3}, T_{700/800us}) + \\ R_{2ms,3} &+ \min(T_{50ms} - r_{50ms,36}, T_{2ms}) + R_{50ms,36} = 41953\ \mu s \end{aligned}$$

Due to the sporadic nature of the first runnable, we assume $T_{700/800us} = 800\ \mu s$ in order to maximize latency.

The end-to-end latency by the F2F semantics requires to add one cycle delay with respect to L2F and to verify Condition (8) for any pair of consecutive runnables. In this case, we obtain $\overline{n} = 2$ for the first stage[1] and $\overline{n} = 43$ for the second stage, which yields:

$$L_3^{F2F} = T_{700/800us} + 2 \cdot T_{700/800us} + R_{700/800us,3} + 43 \cdot T_{2ms} + R_{2ms,3} + R_{50ms,36} = 127553\ \mu s.$$

Finally, the end-to-end latency for the L2L semantics is equal to the L2F case, because no output is overwritten due to the under-sampling effect.

Similar calculations are performed to compute end-to-end latencies accounting for memory effects, as described in Section III-C.

Table II summarizes the obtained end-to-end latencies calculated according to the different semantics adopted, for each of the two challenges.

TABLE II: End-to-end latency upper bounds (μsec) for the first (I) and second (II) challenge.

Chain	L2F I	L2F II	F2F I	F2F II	L2L I	L2L II
1	13376	13383	23376	23383	13376	13383
2	52222	52796	152222	152796	180222	180796
3	41953	42448	127553	130040	41953	43248

IV. Model Simulator

The analysis by simulation of the challenge model has been performed by a purposely developed extension [6] [7] to the C++ RTSIM [8] scheduling simulator.

A. Data Acquisition

The (engine control) application model that is the subject of the challenge is defined by an XML file that can be parsed to obtain the model data. The model information is then stored in data structures internal to the simulator C++ classes.

Some of the model information requires a preliminary elaboration, such as the execution time that is represented by parameters of a Weibull distributions: the lower bound (b), the upper bound (B), the mean (η), and the probability of having values greater than the upperbound (ρ). Those parameters must be converted to compute the standard Weibull parameters: scale (λ) and shape (k). The transformation has been performed considering that the cumulative distribution function (CDF), given an uniformly distributed random variable x, is null for $x < 0$ and for $x \geq 0$ is defined as $CDF(x) = 1 - e^{-(x/\lambda)^k}$. By considering that for $x = B - b$, it is possible to obtain $CDF(B-b) = 1 - \rho$, and after performing some substitution it is possible to define $\lambda = \frac{\sqrt[k]{-\ln \rho}}{B-b}$. The mean value of a Weibull distribution is calculated as $\eta = \lambda \Gamma\left(1 + \frac{1}{k}\right)$, and, by substituting the first result in the second equation, we obtain $\frac{\sqrt[k]{-\ln \rho}}{B-b} \Gamma\left(1+\frac{1}{k}\right) - \eta = 0$.

The approach followed by the simulator described in this paper to obtain an approximation of the k parameter is to minimize the absolute error of the previously described function

$$\min_{k>0} \left| \frac{\sqrt[k]{-\ln \rho}}{B-b} \Gamma\left(1 + \frac{1}{k}\right) - \eta \right|. \quad (17)$$

The function minimum is obtained by using the GNU Scientific Library [9].

B. Cores, Kernels and Schedulers

In RTSIM the main entity for scheduling simulations is the Kernel. Each Kernel has an associated Core. Once a Kernel is instantiated, the programmer assigns a Scheduler to it. Among the different available schedulers, the one used for the challenge is the fixed priority scheduler. In RTSIM, partitioned multi-core scheduling is obtained by instantiating multiple Kernel objects, one for each core.

C. Tasks

Each task τ_i in RTSIM is defined by its parameters: the activation time of first job ($a_{i,0}$), its relative deadline (D_i), its period or minimum inter-arrival time (T_i), and the sequence of instructions it executes, each defined by an execution time specification (deterministic or random). For any periodic task, the activation time of each job is computed by adding T_i to its last activation time. For sporadic tasks, a random value in the range $[T_i, T_i^{max}]$ is added to the last activation time, where T_i^{max} denotes the maximum inter-arrival time of τ_i. Relative deadlines are set equal to T_i. As for the job instructions, each task executes a sequence of runnables. According to the RTSIM syntax, we defined a new instruction "runnable(runnableName)", and the code of each task is of the form

```
runnable(r1);runnable(r2);...runnable(rN);
```

D. Runnables

Cooperative tasks preempt lower priority cooperative tasks only at runnable borders, while higher priority preemptive task can preempt any lower priority task and runnable. In the case of cooperative tasks, preemption within runnables is prevented by locking and unlocking a core-specific mutex dedicated to cooperative tasks before and after calling a runnable. The resulting job code for a cooperative task is:

```
...lock(muxC);runnable(rX);unlock(muxC)...
```

When a job calls a runnable instruction, the operations performed, in order, are the following: updating end-to-end statistics associated to labels reading events, virtually executing the runnable computations, updating end-to-end statistics associated to labels writing events.

E. Results

All the simulation runs performed for the challenge system produced the following: (i) Complete traces of the task scheduling events; (ii) F2F and L2L end-to-end delays of each chain; (iii) Response times of all runnables involved in each chain. The system simulation was performed collecting sample runs for different initial offsets of the tasks. For periodic tasks, the initial offsets are uniformly selected in the interval $[0, T_i]$, while for sporadic tasks they are chosen in $[0, T_i^{max}]$. The execution of the tasks has been simulated for a total virtual time of one hour. The simulation required 28 minutes and 50

[1]This calculation considers $T_{700/800us} = 800\ \mu s$, since this value maximizes the latency of the given effect chain.

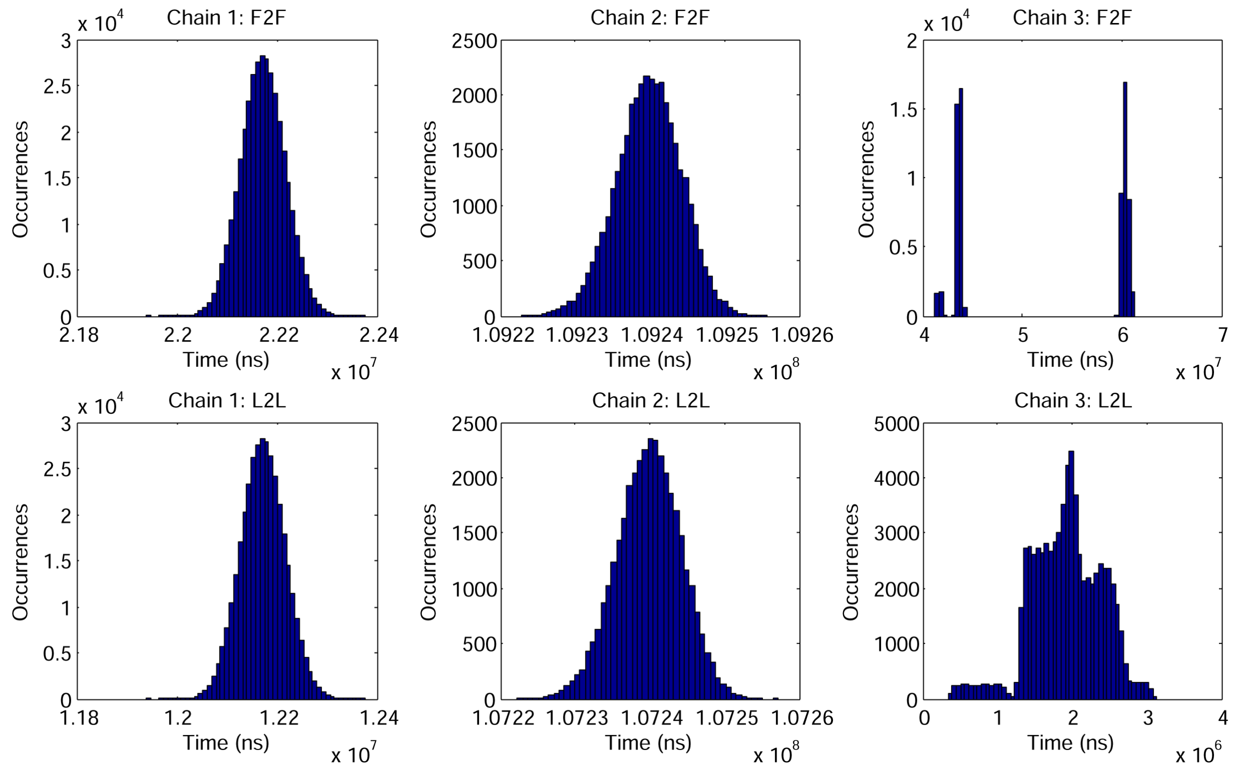


Fig. 4: End-to-end delays obtained by simulation.

seconds on a system with an *Intel i7-2630QM* core running at 2 GHz and 8 GB of DDR3 RAM running at 1333 MHz.

Figure 4 represents the distribution of the F2F and L2L latencies for each chain. For the first chain, the maximum end-to-end delays measured by simulation are 22377 μs for F2F and 12377 μs for L2L. For the second chain, the maximum end-to-end delays measured by simulation are 109.26 ms for F2F and 107.26 ms for L2L. In the end, the simulation returns 61324 μs for F2F and 3139.4 μs for L2L as maximum end-to-end delays for the third chain.

Additionally, Table III establishes a comparison between the worst-case response times of the runnables by the worst-case latency analysis of Section III (WCRT), which takes into account the scaling factors computed in Table I to guarantee schedulability, and the maximum response times observed during our simulations (SIM).

TABLE III: Worst-case response times (μsec) for the first (I) and second (II) challenge.

Runnable	WCRT I	WCRT II	SIM I
$R_{10ms,149}$	5176	5144	3556
$R_{10ms,243}$	7919	7903	5431
$R_{10ms,272}$	8896	8879	6139
$R_{10ms,107}$	3376	3383	2377
$R_{100ms,7}$	39647	39865	6992
$R_{10ms,19}$	770	781	577
$R_{2ms,8}$	142	150	122
$R_{700/800us,3}$	30	33	27
$R_{2ms,3}$	49	53	46
$R_{50ms,36}$	39074	39562	11151

The evaluation of the memory access costs rquires further extensions to the simulation engine that could not be completed in time for this paper.

V. Conclusions

In this paper, we proposed two solutions for the timing verification problem of the FMTV challenge. The first approach builds a mathematical model of the system and calculates worst-case latencies by adaptation of existing response time analysis techniques. Upper bounds on the end-to-end latencies are derived by first ignoring and then including memory access times. Then, a simulator of the given AUTOSAR model has been built on RTSIM to compute end-to-end latencies of the selected effect chains.

References

[1] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics," in *CRTS*, 2008.

[2] G. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. A survey." *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[3] R. Bril, "Existing worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption is too optimistic," *CS-Report 06*, vol. 5, 2006.

[4] R. Bril and W. Verhaegh, "Towards best-case response times of real-time tasks under fixed-priority scheduling with deferred preemption," in *ECRTS, WiP session*, 2005, pp. 17–20.

[5] *A C++ implementation of schedulability analysis and end-to-end latency calculation for WATERS Challenge 2016*, http://retis.sssup.it/%7Eal.melani/downloads/FMTV-analysis.zip, 2016.

[6] "MetaSim2.0 event-based simulator," https://github.com/balsini/metasim2.0, accessed: May 17, 2016.

[7] "RTSIM real-time system simulator extended for waters challenge 2016," https://github.com/balsini/waters/, accessed: Branch 2016.

[8] "RTSIM real-time system simulator," http://rtsim.sssup.it/, accessed: Version 2.0.

[9] "GSL gnu scientific library," https://www.gnu.org/software/gsl/, accessed: Version 1.16.

Computational Analysis of Complex Real-Time Systems - FMTV 2016 Verification Challenge

Ingo Stierand, Philipp Reinkemeier, Sebastian Gerwinn, Thomas Peikenkamp
OFFIS, Oldenburg, Germany
{stierand,reinkemeier,gerwinn,peikenkamp}@offis.de

***Abstract*—Real-time scheduling analysis is an important step in safety relevant embedded system design for many application domains, such as avionics, automotive and automation. Increasing system complexity, not least due to raising automated mobility, requires constant evolution of the analysis approaches, resulting in a vital research domain.**

We like to contribute to the research by presenting a computational analysis approach, where the system model is unfolded as discrete-time state transition system. The analysis engine is tailored particularly for real-time scheduling analysis and exploits respective optimisations. We show the applicability of the approach on an industrial relevant problem, and discuss its advantages and limits.

I. Introduction

The question whether a system can deliver its functions timeliness when deployed on a hardware architecture is an integral part of the safety aspect of embedded system design for many application domains, such as avionics, automotive and automation. Real-time scheduling analysis is a well established discipline, providing a wealth of methods to verify relevant timing aspects of the deployed system. Most approaches are based on also well-established models, so called *task networks*, and differ mainly in details that reflect the focus and capabilities of the underlying mathematical method.

Although the discipline exists for several decades, publicly available benchmarks were rather rare for a long time. People sporadically came up with real-world or carefully designed artificial examples [9], [6]. Such models help the community to compare their methods, to investigate the individual advantages (and disadvantages), and to evolve the approaches.

Recently, a group of researchers came up with the idea of a verification challenge, where particularly timing analysis problems are made public, and invited all interested parties to try their approaches and to discuss the results. We believe this is a very good idea, which is proven to be an effective instrument for progress in other formal verification communities.

We would like to contribute to this effort by providing analysis for a system model that is derived from a large real-world application. The authors of [5] constructed a generator from a anonymised engine control application with thousands of functions, which can be parametrised in order to obtain appropriate benchmarks.

The work has been partially funded by the German Ministry for Education and Research (BMBF) under the funding ID 01IS14029H (AMALTHEA4public) and ID 01IS15031H (ASSUME)

The present system is given as an AMALTHEA4public[1] (A4P) model, and can be downloaded from the WATERS workshop website[2]. On this model, we apply a model-checking based analysis [7]. In contrast to other existing, more general frameworks like timed automata, our approach is based on the idea to construct a highly specialised model-checking engine particularly tailored for real-time scheduling analysis. This has been done before, e.g., with the TIMES tool [1]. Our approach however exploits discrete-time state space construction, using a variant of time darts [4] in order to reduce the footprint of state-space representation.

We briefly discuss the system model, and how we interpret it where needed, in the following section. Section III introduces the analysis approach and details how we tackle the verification challenge. Results are presented in Section IV, followed by a summarising discussion in Section V. Section VI concludes the paper.

II. System Model and its Interpretation

The system of the verification challenge consists of a multi-core processing unit with four identical cores, which are connected to a crossbar switch, and five memory banks. All system components are running at 200 MHz, resulting in a length of 5 ns for a processing cycle.

Every core is directly connected to its local memory bank. Additionally, the cores can access all other local memory banks as well as the global memory bank via the crossbar switch, however, at the cost of additional cycles. The switch, as stated in the challenge call [3], provides full connectivity. We interpret this such that no congestion occurs at the switch due to concurrent memory accesses from different cores. The switch imposes 8 cycles latency on a memory access. The challenge call further states that accesses to the memory banks are serialised using a FIFO strategy. This is also true for the local memories; all accesses from the local core as well as from other cores via the switch go to the same FIFO buffer. Every memory access takes 1 cycle.

The system consists of 21 tasks. Each task contains multiple runnables, which are executed sequentially, as the call graphs in the model indicate. All runnables consist of a similar set of runnable items, which is (1) a sequence of read accesses to various memory cells (labels), (2) execution of an instruction

[1] http://www.amalthea-project.org/
[2] https://waters2016.inria.fr/challenge/

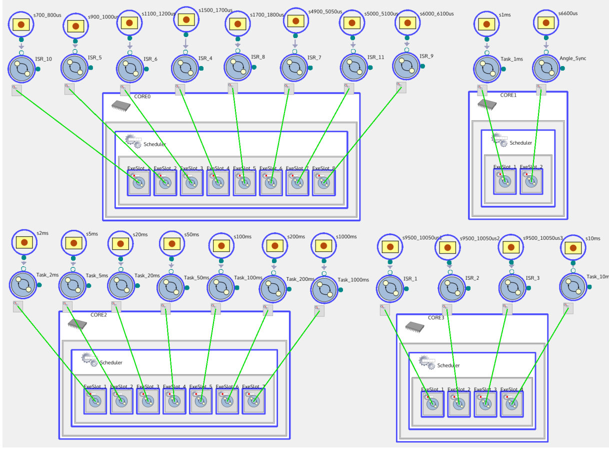


Fig. 1. Analysis Model (all four cores with associated tasks, buses and memories are omitted)

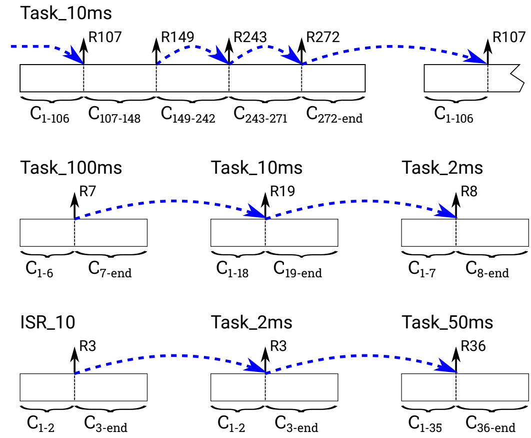


Fig. 2. Three Effect Chains of the Challenge Model

sequence abstracted by a probability distribution, and (3) a sequence of write accesses to memory cells, in this order. We interpret the runnable items as a sequence, i.e., read and write accesses to the labels are performed one after the other.

The tasks are allocated to the four cores as shown in Figure 1. Each core executes an operating system that schedules tasks according to the OSEK standard. To this end, tasks get priorities in ascending order, with 0 being the lowest priority. While these priorities are globally defined, task allocation induces unique core-local priority orders. All except five tasks are preemptively scheduled (cf. Table I). The remaining tasks are cooperative. Preemptive tasks can preempt lower priority tasks – no matter whether preemptive or cooperative – at any point in time. Cooperative tasks can be preempted by higher priority cooperative tasks only at runnable boundaries. Note that, while the A4P meta-model defines so called *schedule points* where cooperative tasks can be preempted, the authors of the challenge explicate it otherwise.

All tasks are activated by individual stimuli. The A4P modelling framework defines various kind of stimuli. Two types are used for the model, namely periodic and sporadic. According to the documentation, periodic stimuli are defined by two parameters. The *offset* defines the time of the first task activation after system initialisation. The *recurrence* parameter defines the activation period relative to the first one. Sporadic stimuli are defined by a probability distribution defining the minimum and maximum inter-arrival time for task activations. All sporadic stimuli in the model are defined by a uniform distribution with lower and upper bound. We assume, as stated by the authors of the challenge, that also for sporadic stimuli the first event occurs at time 0.

The model finally contains three effect chains as shown in Figure 2, which define data flows that can be observed in the system. All events referred to by the chains are start events of runnables. The first chain refers to a sequence of runnables that all belong to task Task_10ms, which is (names are abbreviated) $\{$R149,R243,R272,R107$\}$. A further inspection of the model reveals that theses runnables indeed access common memory labels; every runnable of the chain sequence (except the last) writes to a label that is read by the subsequent runnable. As the runnables of the task are executed in ascending numbering order, the resulting data flow is spread over two subsequent executions of the task, which is indicated in the top part of Figure 2.

The other two are cross-core effect chains. The second chain involves tasks Task_100ms, Task_10ms and Task_2ms, which are allocated to Core 2 and Core 3, respectively. The third chain also crosses two cores, Core 0 (ISR_10) and Core 2 (Task_2ms and Task_50ms).

The challenge states three sub-challenges. All of them ask for tight lower and upper bounds of the end-to-end latencies of the three effect chains. The first effect chain for example is asking for lower and upper bound of the time between the start events of runnables R149 and R107. The first sub-challenge states that all memory accesses shall be ignored. The other two state that memory accesses should be taken into account, which induces additional latencies due to congestions for memory accesses. Concerning the second sub-challenge, we assume that the labels are allocated to the memory banks as defined in the model. The third sub-challenge asks for an allocation of the labels such that the end-to-end latencies become minimal. We do not cope with this optimisation challenge in the present paper.

III. Analysis Approach

In order to keep analysis times and (memory) space manageable, we follow a compositional approach, where we consider the challenge as a set of separate scheduling problems. To this end, we re-model every core and its allocated tasks in terms of our analysis model as exemplified in Figure 3. The top part of the figure shows the relevant artefacts, namely *event sources* (yellow boxes), *tasks* (blue circles) and *processing units* (grey boxes). Event sources emit events according to their assigned event stream behaviour, which is defined by four parameters P^-, P^+, J and O. The time between any two subsequent events is selected non-deterministically as follows: Given time instants $t'_{i+1} \in t'_i + [P^-, P^+]$ the event source emits events at

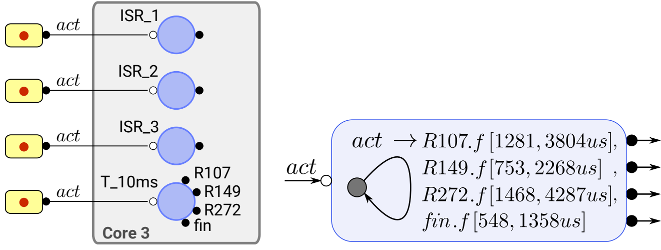


Fig. 3. Analysis Model Example

time points $t_i \in t'_i + [0, J]$. Generally, the first time instant is chosen non-deterministically from the interval $[O, O + P^+]$, i.e., $t'_1 \in [O, O + P^+]$.

Event sources are hence sufficient to model periodic as well as sporadic stimuli of the challenge model. As the stimuli defined in A4P send their first event at a fixed offset (in the model always 0), we have to remove the initial non-determinism of the corresponding event sources in the analysis model. This is obtained by command line parameters of the analysis tool.

Tasks are activated by incoming events (here always act) via their input ports (small white circles), and emit events during execution via their output ports (small black circles). Task execution finishes with the last emitted event. Tasks must emit at least one event. Tasks have internal state transition systems as shown at the right part of Figure 3. Depending on the internal state of a task and the event that activates it, the tasks execution is performed according to the annotation of the corresponding transition. For example, if the task in the figure is activated by an incoming *act* event, it executes the corresponding transition, which is a loop at the sole task state in Figure 3. During execution, it sends event f to output port R107 after $1281 - 3804$ μs “consumed” execution time, an event to output port R149 after $753 - 2268$ μs, and so on. The task finishes execution with the last sent event.

The analysis model allows for two interpretations of the execution times annotated at a transition. For *simultaneous* transitions, the execution times for the individual output events pass simultaneously. The output order of events with overlapping execution time intervals is non-deterministic. For *sequential* transitions, the execution times pass sequentially. At the end of each execution time, the corresponding event is emitted. Only sequential transitions where used for the challenge.

The models used for analysis of the challenge have been manually constructed. For the calculation of the execution times however a simple parser tool has been implemented. While a fully automatic translation would be possible, we avoided the additional effort for the present work. The resulting analysis model is depicted in Figure 1. The minimal and maximal execution times obtained for the tasks, or task segments, from the original model by summing up the execution times of the involved runnables are depicted in Table I with descending (core-local) priority order from top to bottom. In order to enable calculation of bounds for the

TABLE I
TASK PARAMETERS (TIMES W/O MEMORY ACCESSES IN # CYCLES)

Core	Task	preempt.	min	max
0	ISR_10	yes	3.363	6.068
0	ISR_5	yes	25.825	51.636
0	ISR_6	yes	2.980	6.190
0	ISR_4	yes	33.242	73.160
0	ISR_8	yes	26.089	60.777
0	ISR_7	yes	34.678	64.974
0	ISR_11	yes	27.629	61.177
0	ISR_9	yes	35.617	74.097
1	Task_1ms	yes	50.035	152.870
1	Angle_Sync	yes	260.919	761.071
2	Task_2ms	yes	27.748	80.817
2	Task_5ms	yes	73.108	186.363
2	Task_20ms	no	721.008	2.093.688
2	Task_50ms	no	262.830	616.897
2	Task_100ms	no	625.239	1.883.595
2	Task_200ms	no	14.041	27.697
2	Task_1000ms	no	13.610	27.432
3	ISR_1	yes	3.075	7.011
3	ISR_2	yes	2.064	3.549
3	ISR_3	yes	2.424	4.787
3	Task_10ms	yes	797.773	2.342.546

end-to-end latencies, the respective tasks have been modelled using sequential transition executions as shown in Figure 2. For the first effect-chain, task Task_10ms contains a sequential transition with five execution times. The first one subsumes the execution of runnables with numbers 1 to 106 of the tasks call graph. The task contains a corresponding output port R107 at which the start event for runnable 107 can be observed. The second execution segment subsumes the execution of runnables 107 to 148, for which port R149 indicates start of runnable 149, and so on.

During modelling, we made two notable observations. First, execution times for runnables are expressed in terms of Weilbull distributions, which express probabilities for particular execution times. The values in the model are no hard bounds, but define an interval with a certain probability mass, which in our case is $1 - 5 \cdot 10^{-4}$ for all runnables. The definitions imply that there is a non-zero (although potentially very small) probability for each runnable to have very large execution times, which may lead to overload situations where tasks would miss every given finite deadline. Hence, from a safety point of view, the system has to be rejected.

Secondly, we observed that the utilization of three cores (1, 2 and 3) is larger than 100%. For a simple fixed-priority scheduling, this would result in an infeasible task set that cannot be scheduled. The A4P model however defines an OSEK scheduling scheme for all cores, and a limit of one for the maximum number of activations for each task. The model hence implies (considering the OSEK specification) that for each activated task all further activations of this task are ignored until it finishes it execution.

We exploit a model-checking approach for analysing the effect chains, which is implemented in the tool RTANA$_2$ (cf. footnote 3). It is fed with an analysis model and performs a discrete-time state unfolding, resulting either in a closed

state-transition system, or terminates if it detects an infeasible scheduling situation, such as a buffer overflow. After state space construction, the tool performs a path analysis in order to obtain the exact minimal and maximal latencies for the respective effect chain. For further details about the approach the reader is referred to [7].

For the verification challenge, we deal with the state-space explosion problem in three ways. First, we introduce abstractions where needed by increasing the length of discrete time slots, at which scheduling decisions occur. The effect is similar to the so-called tick scheduling [8]. Additionally, we exploit the model characteristics where possible to perform compositional analysis. Foremost, we consider the cores separately. If this does not sufficiently reduce the state space, we incorporate analytic methods to obtain response times for individual tasks. The results are fed back to the computational analysis, indeed introducing additional over-approximations. A detailed discussion of the analysis and their results is given in the following section.

IV. Results

As stated above, we took a number of measures to tackle the problem of state space explosion. Although the analysis exploits some symbolic representation of time, a main factor for the resulting memory footprint is the length of discrete time slots. With respect to the model, a suitable slot length would be $5ns$. Due to the characteristics of the challenge model with its large execution time intervals, this leads to very large state spaces. Hence, we decided to set the slot length to $1\mu s$, which indeed results in an over-approximate analysis. In order to still obtain safe approximations, we adjusted the execution times accordingly: for lower bounds we took the floor, and for upper bound the ceiling. More precisely, we calculated an interval $[l', u']$ of $1\mu s$ slots from execution time interval $[l, u]$ such that $l' = \lfloor \frac{l}{200} \rfloor$ and $u' = \lceil \frac{u}{200} \rceil$.

To further reduce analysis effort, we also constructed individual analysis models for the various sub-problems. For example, two models have been constructed for the second effect chain, where only relevant parts of the original model remain. This includes to sum up the execution times of runnables that are not relevant for the particular analysis task.

The following sections discuss the individual approaches for the sub-challenges. The analysis models and result logs are also publicly available[3].

A. Sub Challenge 1 - First Effect Chain

The first effect chain does not involve further abstraction as it involves only a single task running on Core 3. The model used for analysing respective latency bounds is depicted in Figure 3. The results in Table II for the first effect chain also show the individual task response times obtained with the analysis.

Scenarios for the results are depicted in Figure 4. The lower bound corresponds to the situation where two subsequent activations of task Task_10ms occur, all runnables R1 – R148 and interrupt service routines ISR_1 – ISR_3 of the first activation consume their maximum processing time, and in the second activation R1 – R106 and ISR_1 – ISR_3 consume their minimum processing time.

[3]https://vprojects.offis.de/rtana

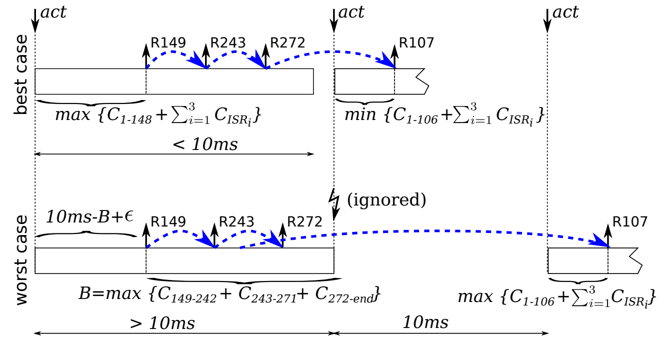


Fig. 4. Scenarios for Sub-Challenge 1 - Effect Chain 1

A scenario for the upper bound is shown at the bottom of Figure 4. Here, all runnables starting from R149 of the first activation consume their maximum execution time. The overall task execution is slightly longer than $10\ ms$, resulting in ignoring the subsequent task activation (OSEK task activation limit). Runnables R1 – R106 and interrupt service routines ISR_1 – ISR_3 of the third activation consume their maximum execution time as well.

B. Sub Challenge 1 - Second Effect Chain

In order to avoid state space explosion when analysing the second effect chain, we exploit (1) a compositional analysis approach in combination with an analytical analysis method implemented by pyCPA [2], and (2) a trick. While the first and last task of the effect chain are executed on Core 2, the intermediate task Task_10ms is executed on Core 3. The analysis is done in three steps. First, we obtain time bounds for execution of task Task_10ms from its activation up to the start event of runnable R19. Second, we create a 'placeholder' task Task'_10ms with execution time bounds according to the results of the first step. The task is not allocated to Core 2, causing the analysis to assume a distinct processing resource solely assigned to the task, which hence executes without any interferences. This way, the model provides a safe over-approximation for imposed data flow latencies on the effect chain. The same approach is applied to obtain time bounds for execution of task Task_100ms from its activation up to the start of runnable R7. To obtain these bounds we setup a pyCPA model with all tasks from Core 2 having a higher priority than Task_100ms. Again, we modelled a placeholder task Task'_100ms based on these results.

Concerning the „trick", cooperative scheduling as in the challenge can be considered as temporal priority inversion. The maximum length of the inversion is no longer than the highest maximum execution time among all runnables of lower-priority tasks. This time is added to the execution time of Task_100ms, which is again a safe over-approximation of the actual behaviour.

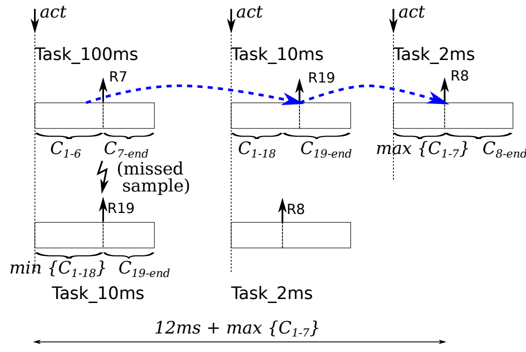


Fig. 5. Worst case scenario for Sub-Challenge 1 - Effect Chain 2

The first two analyses for the second effect chain in Table II depicts the response time bounds for Task_10ms from activation until the start of runnable R19 and for Task_100ms from activation until the start of runnable R7.

A scenario for the upper bound is shown in Figure 5. Runnables R1 – R18 of task Task_10ms consume their minimum processing time. Runnable R19 is just started before runnable R7 of task Task_100ms. So the effect influences the next start of R19 in the next job of Task_10ms. Since $min\{C_{1-18}\} > max\{C_{1-7}\}$, the job of task Task_2ms cannot sample the data of the job of task Task_10ms starting at the same time. Thus, the execution of R8 in the next job of Task_2ms samples that data. In that job of Task_2ms the runnables R1 – R7 consume their maximum processing time.

The calculated lower bound of 0 is due to the compositional approach where task dependencies are lost. Therefore, the start events of involved runnables can occur at the same time instant, and the analysis has to assume that they might occur in the order R7 $\rightarrow$ R19 $\rightarrow$ R8 with no delay inbetween.

C. Sub Challenge 1 - Third Effect Chain

For the third effect chain, we apply a similar approach as for the second one. This time we insert placeholder tasks ISR_10 and Task'_50ms. Again we use pyCPA to obtain the bounds for execution of Task_50ms up to the start of its runnable R36. ISR_10 however is the highest priority task running on Core 0. Hence, it is sufficient to model this task without a resource, but with its core execution times. Concerning the „trick", time is added to the execution time of Task_50ms instead of Task_100ms, which is the maximum execution time among all runnables of Task_100ms, Task_200ms and Task_1000ms.

A scenario for the upper bound is shown in Figure 6. Runnables R1 – R2 of task Task_2ms consume their minimum processing time. Runnable R3 is just started before runnable R3 of ISR_10. So the effect influences the next start of R3 in the next job of Task_2ms. Here again a data sample might be missed and the invocation of runnable R36 in the next job of Task_50ms results in the worst case scenario for the third effect chain. In that job of Task_50ms the runnables R1 – R36 consume their maximum processing time.

The lower bound of the effect chain is 0 for the same reasons as for the lower bound of the second effect chain.

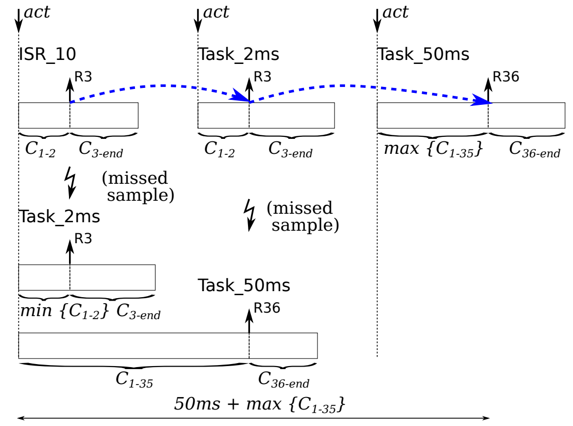


Fig. 6. Scenarios for Sub-Challenge 1 - Effect Chain 3

TABLE II
RESULTS FOR SUB-CHALLENGE 1

	Latency Bound/ BCRT,WCRT	Analysis Time	Space
ISR_1	[15, 36] μs		
ISR_2	[25, 54] μs		
ISR_3	[37, 78] μs		
Task_10ms	[4.024, 11.871] μs		
Effect Chain 1	[5.105, 19.524] μs	49 s	2 GiB
Task'_10ms	[315, 831] μs	41 s	1 GiB
Task'_100ms	[99, 39.890] μs	1 s	22 MiB
Effect Chain 2	[0, 11.826] μs	87 s	400 MiB
ISR_10	[13, 25] μs	–	–
Task'_50ms	[975, 39.029] μs	1 s	22 MiB
Effect Chain 3	[0, 89.015] μs	56 s	400 MiB

D. Sub Challenge 2

The second sub-challenge states that memory accesses of runnables shall be taken into account. A comprehensive analysis would calculate exact bounds on the latencies imposed by concurrent such accesses from tasks running on different cores. As this is currently infeasible by our analysis, we have to calculate safe approximations. This is however simple, as all variables are mapped to the global memory bank. Hence every memory access can be delayed by up to three other memory accesses (from other cores). This results in an overall latency of every memory access between 9 and 12 cycles. Based on these adapted runnable execution times, the analysis then follows the same scheme as for sub-challenge 1. The results are shown in Table III

An interesting result is that the worst-case reaction time of the second effect chain becomes lower when taking memory accesses times into account. This is because the best-case execution time of runnables R1 – R18 of Task_10ms increases, resulting in a smaller time distance to the final reaction of the effect chain.

E. Probabilistic Aspects

It is worth mentioning that the results we reported in this section are only valid with a certain probability. This is due to the fact that the execution times of the different runnables are subject to random fluctuations. Within the model given

TABLE III
RESULTS FOR SUB-CHALLENGE 2

	Latency Bound/ BCRT,WCRT	Analysis Time	Analysis Space
ISR_1	[16, 37] μs		
ISR_2	[27, 56] μs		
ISR_3	[40, 82] μs		
Task_10ms	[4.247, 12.171] μs		
Effect Chain 1	[5.042, 19.782] μs	62 s	2 GiB
Task'_10ms	[332, 854] μs	51 s	1.5 GiB
Task'_100ms	[104, 99.223] μs	1 s	22 MiB
Effect Chain 2	[0, 11.811] μs	848 s	750 MiB
ISR_10	[14, 26] μs	–	–
Task'_50ms	[995, 39.628] μs	1 s	22 MiB
Effect Chain 3	[0, 89.613] μs	54 s	400 MiB

for the challenge, these fluctuations are characterised by a Weibull distribution. Specifically, the individual upper and lower bounds on the execution times, which we used in this section, mark intervals of execution times containing a probability mass of $1 - 5 \cdot 10^{-4}$. From this we can derive a lower bound on the probability that the computed bounds hold. More precisely, the computed bounds hold, if the execution times of all runnables with random execution times fall into their respective intervals. As the individual fluctuations are assumed to be independent, this probability is given by $(1-5\cdot10^{-4})^{1250}$. However, this is a rather pessimistic bound, as the latency bounds could still hold, even if one or more individual execution times lie outside of the intervals used.

V. DISCUSSION

The AMALTHEA4public project aims at defining a comprehensive meta-model for real-time systems with focus on the automotive domain, being compliant with AUTOSAR where possible. Tasks, for example, may contain call graphs, which precisely define execution ordering of the runnables within the tasks, as well as their internal behaviour in terms of runnable items. From this point of view the model was easy to understand. However, there is still room for interpretation.

First, it was an effort to retrieve the exact semantics of stimuli. While the documentation of the A4P meta-model defines precisely the semantics of periodic stimuli, definition of sporadic stimuli is rather sloppy, and required clarification by the challenge authors.

The second obstacle was the interpretation of cooperative tasks. The OSEK standard defines various configurations, resulting in different preemption scenarios for the entire task set. It looks like the A4P meta-model either misses documentation of the chosen interpretation or some bits of information allowing to select the intended one. Furthermore, the A4P meta-model defines the particular type 'schedule point' of runnable entity in order to explicitly define code positions where cooperative tasks can be preempted. While no such entities are defined in the model, the challenge authors state that they should be implicitly assumed to exist.

In summary, it took about a day work, including reviewing documentation, to understand model semantics as precise as required for the analysis. No less time was required to set up the analysis models. The main issue here was to find suitable abstractions such that the analysis would fit into the available memory space. While it would be possible to construct a comprehensive analysis model also including memory accesses for the entire system, it was clearly impossible to get analysis results for such model in reasonable time and space. Particularly memory accesses and preemption with cooperative scheduling involved significant effort in tailoring the models. As this is indeed somehow unsatisfactory, it shows some deficiencies of the current analysis, and points towards potential directions for further improvements, such as improved combination of analytic and computational analysis.

VI. CONCLUSION

We presented a computational analysis approach for the verification of timing characteristics of a non-trivial model that is based on a real-world engine control application. As always with computational approaches, the analysis soon starts to suffer from state-space explosion for „interesting" system sizes. However, as the analysis engine is particularly designed for dealing with real-time scheduling problems, it already shows nice performance compared to generic model-checking approaches such as timed automata or SAT-based engines.

The model presented for the verification has some interesting properties that are hard to encode with classical analytic real-time scheduling approaches. We are convinced that computational approaches can provide valuable results in such cases. We strongly believe that this line of research is still in its infancy and has much potential for further improvements. Real-world problems such as the present are highly useful in order to find sweet spots for such evolution.

REFERENCES

[1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *In Proc. of FORMATS'03, number 2791 in LNCS*, pages 60–72. Springer-Verlag, 2003.

[2] J. Diemer, P. Axer, and R. Ernst. Compositional Performance Analysis in Python with pyCPA. In *3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2012.

[3] A. Hamann, D. Ziegenbein, S. Kramer, and M. Lukasiewycz. FMTV 2016 Verification Challenge. Robert Bosch GmbH Corporate Research, Germany.

[4] K.Y. Jørgensen, K.G. Larsen, and J. Srba. Time-Darts: A Data Structure for Verification of Closed Timed Automata. In *Proc. of the 7th International Conference on Systems Software Verification (SSV)*, volume 102 of *EPTCS*, pages 141–155. Open Publishing Association, 2012.

[5] S. Kramer, D. Ziegenbein, and A. Hamann. Real World Automotive Benchmarks For Free. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS)*, 2015.

[6] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and G. Harbour. Influence of Different System Abstractions on the Performance Analysis of Distributed Real-Time Systems. In *Proc. Conference on Embedded Software (EMSOFT)*, 2007.

[7] I. Stierand, P. Reinkemeier, T. Gezgin, and P. Bhaduri. Real-Time Scheduling Interfaces and Contracts for the Design of Distributed Embedded Systems. In *Proc. International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2013.

[8] K. W. Tindell, A. Burns, and A.J. Wellings. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. *Journal of Real-Time Systems*, 6(2):133–151, 1994.

[9] K.W. Tindell, A. Burns, and A.J. Wellings. Allocating hard real-time tasks: An NP-Hard problem made easy. *Real-Time Systems*, 4:145–165, 1992. 10.1007/BF00365407.

Schedulability and Timing Analysis of Mixed Preemptive-Cooperative Tasks on a Partitioned Multi-Core System

Ignacio Sañudo, Paolo Burgio and Marko Bertogna
Universita di Modena, Italy
{ignacio.sanudoolmedo, paolo.burgio, marko.bertogna}@unimore.it

***Abstract*—This paper proposes a solution for the FMTV verification challenge related to the timing and schedulability analysis of an engine management system to be executed on a shared-memory multi-core platform. The application consists of statically partitioned tasks, each one composed of multiple runnables that are executed according to a read-compute-write policy, where the memory labels required by a runnable are loaded from memory before starting its execution, and they are all stored after the runnable completes its execution. Tasks may be either fully preemptive or only partially at runnable boundaries. The contribution of the paper is threefold. First, we present a tight schedulability analysis for this mixed-preemption setting, neglecting memory accesses (Challenge I). Then, memory access times and arbitration delays are included to the schedulability analysis, addressing Challenge II. Finally, Challenge III is tackled proposing different approaches to map the labels to local/global memories so as to minimize the end-to-end latency of selected event chains.**

I. Introduction

The purpose of this paper is to present a brief overview of a solution to the FMTV verification challenge. The challenges proposed are:

- Challenge I: *calculate tight end-to-end latencies ignoring memory accesses and arbitration*
- Challenge II: *calculate tight end-to-end latencies including memory access and arbitration accesses*
- Challenge III: *optimize end-to-end latencies by mapping the labels among the local and global memories*

The rest of the paper is organized as follow. Section 2 introduces the terminology and notation used in the paper. Section 3 presents the worst-case response time analysis developed to solve Challenge I. Section 4 describes the approach applied to tackle memory access and arbitration accesses (Challenge II). Finally Section 5 presents different solutions for Challenge III.

II. Terminology and notation

In this section, we introduce the terminology and notation used throughout the paper, considering the information abstracted from the Amalthea model. Each task τ_i is specified by a tuple $(C_i, D_i, T_i, P_i, PT_i)$, where C_i is the worst-case execution time (WCET), D_i is the relative deadline, T_i is the period, P_i is the priority, and PT_i is the preemption type. Every period T_i, each task releases a job composed of γ_i subsequent runnables, where $\tau_{i,r}$ represents the rth runnable

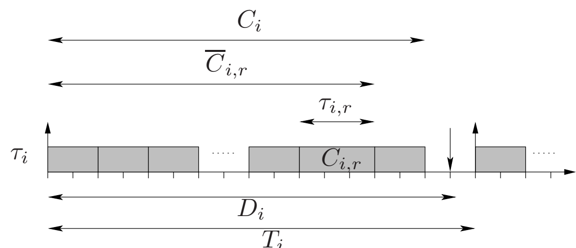


Fig. 1. Notational model for tasks and runnables.

of τ_i, with $1 \leq r \leq \gamma_i$. The execution time of $\tau_{i,r}$ is denoted as $C_{i,r}$. Therefore,

$$C_i = \sum_{r \in [1,\gamma_i]} C_{i,r}. \tag{1}$$

We also denote as $\overline{C}_{i,r}$ the cumulative execution time of runnables $\tau_{i,1}, \ldots, \tau_{i,r}$, i.e.,

$$\overline{C}_{i,r} = \sum_{r \in [1,r]} C_{i,r}. \tag{2}$$

Some of these parameters are exemplified in Figure1 for a generic task τ_i.

Runnables are basic workload units, whose execution follows a read-compute-write policy. The computational part of a runnable cannot start before all its required labels are pre-loaded from memory. Also, no label will be stored to memory before the completion of the runnable. The preemption type PT_i may be either preemptive or cooperative. Preemptive tasks may always preempt lower priority tasks, while cooperative tasks may preempt a lower priority one only at runnable boundaries. Preemptive tasks are assumed to have always a higher priority than any cooperative task.

The execution time of a runnable $\tau_{i,r}$ is computed as $C_{i,r} = n^I_{i,r}/f$, where $n^I_{i,r}$ is an upper-bound on the number of instructions specified by the Weibull estimators for the considered runnable, assuming one instruction-per-cycle (i.e., $IPC = 1$), and f is the core frequency.

The platform is assumed to comprise four identical cores, with tasks statically partitioned to the cores and no migration support.

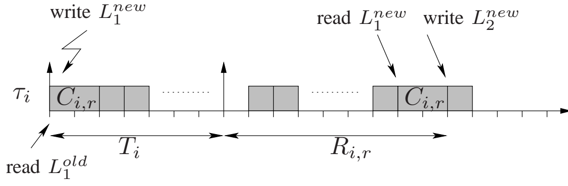


Fig. 2. Worst-case delay propagation in a runnable sub-chain.

III. Memory-oblivious Analysis

In this section, we present a detailed analysis of Challenge I, i.e, a solution to *calculate tight end-to-end latencies ignoring memory accesses and arbitration*. The latencies of interest are those of selected effect chains, where an effect chain is a sequence of producer/consumer runnables working on shared labels. It is worth noting that effect chains do not have a blocking semantic, i.e., tasks and runnables are always active and periodically activated, independently on other runnables and/or external events. What is interesting to analyze is the maximum propagation delay from an initial event to the final runnable involved in the effect chain. An effect chain is triggered by an initial event, which needs to be processed by one or more runnables using a read/execute/store execution model. A first runnable $\tau_{i,x}$ may read a label L_1, compute the necessary instructions, and store a result on a label L_2, which will be later read by another runnable $\tau_{j,y}$ following in the chain, and so on until the last runnable in the chain. The end-to-end propagation delay is the maximum time that may elapse between the initial event and the completion of the last runnable in the chain.

It is easy to observe that an upper bound of such a delay is given by the sum of the propagation delays for each individual runnable in the chain [1]. In particular, consider an effect sub-chain where a runnable $\tau_{i,x}$ writes a label L which is then read by another runnable $\tau_{j,y}$. The worst-case sub-chain propagation delay is found when $\tau_{i,x}$ stores L right after $\tau_{j,y}$ started loading it, as shown in Figure 2. Under this situation, the effect is not propagated until the next instance of $\tau_{j,y}$ may start executing in the subsequent period T_j, and complete its execution after at most $R_{j,y}$ time-units, where $R_{j,y}$ represents the worst-case response time of runnable $\tau_{j,y}$. Therefore, an upper bound on the overall end-to-end propagation delay of an effect chain EC can be computed as

$$\delta(EC) = \sum_{\tau_{i,r} \in EC} (T_i + R_{i,r}), \tag{3}$$

where the sum is extended over all runnables belonging to the effect chain. Note that in case the effect chain includes two consecutive runnables that belong to the same task, it is sufficient to consider only the delay contribution of the later one.

To compute the upper bound of Equation 3, it is necessary to compute the worst-case response time $R_{i,r}$ of each runnable $\tau_{i,r}$ involved in the chain. To this purpose, we will hereafter provide a tight response-time analysis of runnables belonging to either preemptive or cooperative tasks. Since Challenge I allows neglecting memory delays, we can focus uniquely on the execution times of tasks and runnables.

A. Analysis for Preemptive Tasks

According to the considered model, preemptive runnables can only be preempted by higher priority preemptive runnables, and they can always preempt any lower priority task. Therefore, a preemptive task will never experience any blocking delay due to lower priority (preemptive or cooperative) tasks. Hence, the response time for preemptive tasks can be computed adapting the classic response time analysis for arbitrary deadlines presented in [2]. The arbitrary deadline model is used instead of the simpler analysis for constrained deadlines because there are configurations where the response time of a task may be later than the activation of the subsequent job of the same task, i.e., it may be $R_i > T_i$. Under these conditions, the maximum response time of a task is not necessarily given by the first instance released after the synchronous arrival of all higher priority tasks (also called critical instant), but may be due to later jobs.

For each task τ_i, the analysis requires checking multiple jobs until the end of the level-i busy period, i.e., the maximum consecutive amount of time for which a processor may be continuously executing tasks of priority P_i or higher. The longest Level-i active period can be calculated by fixed-point iteration of the following relation, starting with $L_i = C_i$:

$$L_i = \sum_{j:P_j \geq P_i} \left\lceil \frac{L_i}{T_j} \right\rceil C_j. \tag{4}$$

The number of τ_i's instances to check are therefore:

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{5}$$

The finishing time of the k-th instance ($k \in [1, K_i]$) of runnable $\tau_{i,r}$ in the level-i busy period can be iteratively computed as

$$f_{i,r}^k = \sum_{j:P_j > P_i} \left\lceil \frac{f_{i,r}^k}{T_j} \right\rceil C_j + (k-1)C_i + \overline{C}_{i,r}, \tag{6}$$

where the first term in the sum accounts for the higher priority interference, the second term accounts for the $(k-1)$ preceding jobs of τ_i, and the last term considers the contribution of the k-th job limited to $\tau_{i,r}$ and its preceding runnables.

The response time of the k-th instance of $\tau_{i,r}$ can then be easily found subtracting its arrival time:

$$R_{i,r}^k = f_{i,r}^k - (k-1)T_i. \tag{7}$$

Finally, the worst-case response time of runnable $\tau_{i,r}$ can be found by taking the maximum among all K_i jobs in the level-i busy period:

$$R_{i,r} = \max_{k \in [1, K_i]} \{R_{i,r}^k\}. \tag{8}$$

B. Analysis for Cooperative Tasks

The analysis for cooperative tasks is somewhat more complicated, since it needs to take into account (i) the blocking delays due to lower priority cooperative tasks that can be preempted only at runnable boundaries; (ii) the interference due to higher priority cooperative tasks that can preempt the considered task only at runnable boundaries; (iii) the interference of preemptive tasks that may always preempt even within a runnable. To tackle this problem, we will modify and merge the analysis for limited-preemption systems with Fixed Preemption Points (FPP) and for Preemption Threshold Scheduling (PTS), both summarized in [3]. The outcome will be a necessary and sufficient response-time analysis for the considered mixed preemptive-cooperative task model.

Under this model, a preemption threshold is assigned to cooperative tasks. This priority is higher than that of any cooperative task, but lower than that of any preemptive tasks. When a cooperative task τ_i is executing one of its runnables, its nominal priority P_i is raised to the threshold θ_i, so that cooperative tasks cannot preempt it. The nominal priority is restored when the runnable is completed, allowing cooperative preemptions from higher priority tasks.

As with preemptive tasks, also for cooperative tasks it is necessary to consider multiple jobs within a busy window. However, the busy window must also include the blocking due to lower priority tasks. The longest Level-i active period can be calculated adding a blocking factor to the recurring relation of Equation (4):

$$L_i = B_i + \sum_{j:P_j \geq P_i} \left\lceil \frac{L_i}{T_j} \right\rceil C_j. \quad (9)$$

Since a task can only be blocked once by lower priority instances, B_i corresponds to the largest execution time among lower priority runnables[1]:

$$B_i = \max_{j,r:P_j < P_i} \{C_{j,r}\}. \quad (10)$$

Equation (5) can then be used to compute the number of instances to check in the busy window.

The starting time $s_{i,r}^k$ of the k-th instance of runnable $\tau_{i,r}$ can be computed taking into consideration the blocking time B_i, the interference produced by higher priority tasks before $\tau_{i,r}$ can start, the preceding (k-1) instances of τ_i, and the execution time of the preceding runnables of $\tau_{i,r}$:

$$s_{i,r}^k = B_i + \sum_{j:P_j > P_i} \left(\left\lfloor \frac{s_{i,r}^k}{T_j} \right\rfloor + 1 \right) C_j + (k-1)C_i + \overline{C}_{i,r-1}. \quad (11)$$

The finishing time $f_{i,r}^k$ is calculated by adding to the starting time $s_{i,r}^k$, the execution time of the considered runnable $C_{i,k}$, along with the interference of the tasks that can preempt $\tau_{i,r}$, i.e., the preemptive tasks which have a nominal priority higher than the preemption threshold of any cooperative task. To compute this last interfering term, we compute the higher priority instances that may arrive from the critical instant until the finishing time, and subtract those that arrived before the starting time.

$$f_{i,r}^k = s_{i,r}^k + C_{i,r} + \sum_{j:P_j > \theta_i} \left(\left\lceil \frac{f_{i,r}^k}{T_j} \right\rceil - \left(\left\lfloor \frac{s_{i,r}^k}{T_j} \right\rfloor + 1 \right) \right) C_j. \quad (12)$$

Equation (7) and (8) can then be identically used to compute the worst-case response time $R_{i,r}$ of the considered runnable.

Since the deadlines are missed and the utilization is over 1 in almost all cores, we have reduced the worst case execution time of some runnables in order to make the system schedulable, Table I shows the results of the first challenge.

TABLE I
END-TO-END LATENCIES IGNORING MEMORY ACCESSES (μ'S)

Core	Task	WCRT	Deadline	U
CORE0	ISR_10	30.34	700.0	0.04
	ISR_5	288.52	9000.0	0.33
	ISR_6	319.47	1100.0	0.35
	ISR_4	685.27	1500.0	0.60
	ISR_8	1308.62	1700.0	0.78
	ISR_7	2652.99	4900.0	0.84
	ISR_11	4266.89	5000.0	0.90
	ISR_9	4483.08	6000.0	0.93
CORE1	Task_1ms	764.35	1000.0	0.76
	Angle_Sync	5994.08	6660.0	0.97
CORE2	Task_2ms	262.65	2000.0	0.13
	Task_5ms	1194.47	5000.0	0.31
	Task_20ms	16870.06	20000.0	0.84
	Task_50ms	36776.80	50000.0	0.90
	Task_100ms	99719.82	100000.0	0.99
	Task_200ms	99845.02	200000.0	0.99
	Task_1000ms	99973.85	1000000.0	0.99
CORE3	ISR_1	35.05	9500.0	0.003
	ISR_2	52.8	9500.0	0.005
	ISR_3	76.73	9500.0	0.008
	Task_10ms	9992.16	10000.0	0.99
Effect Chain		**End to End Latency**		
Effect Chain_1		13378.124		
Effect Chain_2		149691.134		
Effect Chain_3		72196.007		

IV. MEMORY-AWARE ANALYSIS

In this section, we address Challenge II, including memory and arbitration accesses in the computation of the end-to-end latencies. We follow an identical approach as the one described in the previous section, inflating the runnable execution times $C_{i,r}$ with the maximum possible interference produced by memory-related delay.

We assume all labels be loaded/stored to global memory, leaving the improvements related to the use of local memories to Challenge III discussed in the next section. The delay for

[1] Since the lower priority task must have already arrived before the critical instant, the actual blocking term is actually an infinitesimal amount smaller. We neglect infinitesimal amounts to simplify the formula.

a global memory access is of 8 cycles for crossbar traversing and 1 cycle for the memory access. Since conflicting memory accesses are assumed to be arbitrated in a First-In-First-Out fashion, the memory access time has to be multiplied by the number of cores m that may concurrently access the global memory, i.e., four cores in our setting: $m = 4$. The overall memory access delay can then be found by multiplying the single access delay by the number of reads n^R and writes n^W performed by the considered runnable. Therefore, the resulting WCET $C_{i,r}$ for a runnable can be computed as:

$$C_{i,r} = (n^I/f) + (8 + (1*m)*n^R) + (8 + (1*m)*nW) \quad (13)$$

The multiplying factor m accounts for the maximum possible interference by all cores in the system, that is, we assume that cores continuously generate interfering traffic. This is a pessimistic assumption that may be improved by accounting for data access patterns of target applications, which are known in the Amalthea model. In particular, a possible solution can be found along the lines of the work presented by Nelis et al. in [4], where a method is introduced to model the memory access patterns of a task considering the contention on a shared bus (and not a crossbar, as in the considered model). Other approaches that could be used to tackle this problem are presented in [5] and [6]. However, the computational cost of these solutions is exponential in the number of tasks and the granularity of memory patterns, making it difficult to apply for the considered setting.

V. Memory Mapping Strategies

As requested in Challenge III, this section discusses how to optimize end-to-end latencies by means of a suitable mapping of the labels among the local and global memories. Before tackling this challenge, it is first necessary to question the notion of “optimality” for this setting. As we will show in this section, a given label-to-memory mapping can reduce end-to-end latencies for certain effect chains at the cost of increasing those of other chains, making it difficult to take globally optimal decisions.

In a first step, we performed a preliminary analysis of the memory accesses performed by all runnables in the given Amalthea use-case. We categorized the data items (labels) in three sets:

1) *PRIVATE* labels, which are exclusively accessed by one runnable;
2) *SHARED* labels, which are accessed by multiple runnables (e.g., in a producer-consumer fashion);
3) *UNUSED* labels, which we ignore.

Table II shows the number of labels in the proposed model, and their total memory occupation in KBytes, while Figure 3 shows how many (PRIVATE and SHARED) labels are accessed by (runnables assigned to) each core, and their size in bytes (right).

A first consideration is that there is potentially sufficient space to store all labels in any of the memories of the system, eiher in LRAMs (size 128 KB, according to the specifications) or GRAM (256 KB). For this reason, and for the sake of simplicity, we do not consider memory constraints in our analysis. Enhancing our model and approach including limited memory is left as a future work. Moreover, we assume that all labels can be accessed with a single memory read, neglecting the fact that there are labels which are larger than the bus width (i.e., occupy 64 or 128 bits against a 32-bit bus), hence more consecutive memory accesses may be required for a label transfer. However, the proposed methodology can be easily extended to deal with this issue.

	#	Size (KB)
PRIVATE	8293	22.1
SHARED	1690	9.50
UNUSED	17	-

TABLE II
LABELS

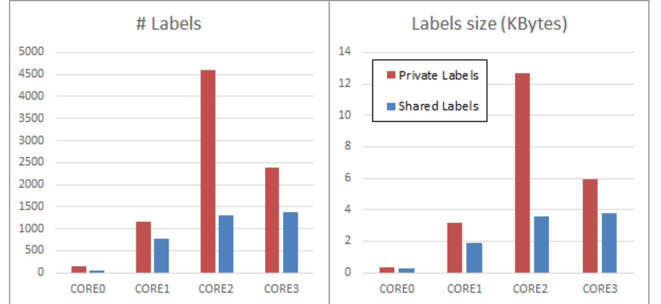


Fig. 3. Distribution of labels on runnables/cores

For the PRIVATE labels, an optimal choice seems to map them to the local memory of the core that exclusively accesses them, because the latency of local accesses to LRAM is always significantly smaller than that to GRAM (1 cycle vs. 8+1 cycles, respectively). Since there are no constraints on the local memory size with relation to the overall labels footprint, moving local labels to other (local or global) memories would only increase the resulting latencies. Moreover, this cannot possibly degrade the delays on other cores, because we *removed* a potential source of contention. This is a quite known technique when programming distributed Non-Uniform Memory Access (NUMA) systems [7].

We define T_{LRAM} as the time spent in the worst case to access a private label stored in local memory, and T_{GRAM} as the worst-case time to access a label stored in shared memory. Assuming the worst-case conflicts in both memories,

$$T_{LRAM} = (m-1) * 1(FIFOqueue) + 1(memory) = m$$

$$T_{GRAM} = 8(xbar) + 1(memory) + (m-1) * 1(FIFO) = 8 + m,$$

where numbers are in clock cycles, and m is the number of cores in the system. Note that time to access private labels store in the local memory may be lower than m when some of the other cores has no label to access in that local memory. This would reduce the number of instances waiting in the FIFO

queue. In the extreme case where each LRAM contains only private labels, $T_{LRAM} = 1$, since there will never be any conflict in accessing local memories.

Moving to the mapping problem of shared labels, we could use a similar approach to map each label to the LRAM "closer" to the core that mostly accesses it. Unfortunately, this could worsen the latencies of other runnables on the same core when accessing private labels stored in the local LRAM, because now they may conflict with remote accesses from other cores. The proposed heuristic is convenient if the accesses to shared labels are more frequent than those on private labels, so that the increased conflicts in accessing private labels are compensated by the gain in loading a shared label from LRAM instead of GRAM.

If memory access patterns are not taken into account, the increase in the latency for private accesses is the same if we map *one or all the* shared labels to the local memory. As a consequence, if we decide to map a single shared label onto the LRAM of a core, paying the consequent private access penalty, it would then make sense to map to that LRAM also other shared labels that are most frequently accessed by that core, since there would not be any further penalty to private accesses. This seems to suggest an *"if one, then all"* approach, according to which a local memory is either left free from any shared label, or it is filled with the most frequently accessed shared labels by the corresponding core.

From the Amalthea model, we know that several runnables act in a producer-consumer fashion, forming multiple *effect chains*. As we showed, privileging one runnable might have the side effect of degrading performance for some other runnables on the same core, which might belong to a different effect chain. For this reason, *it is difficult to design a methodology for shared label mapping which "optimizes" end-to-end latencies in a "generic" sense in the proposed model*. What can be more easily done is tailoring the label mapping problem to one or few privileged effect chains, reducing the latency of the corresponding runnables by selecting their most suitable mapping strategy.

VI. Conclusions

This paper presented a set of possible solutions for the FMTV 2016 Verification Challenge. The main contribution is a tight schedulability analysis for the considered task model in which cooperative and preemptive tasks are concurrently scheduled on the same partitioned platform. Such an analysis has then been extended to include memory access delays and to propose promising heuristics for mapping labels to local memories. A Java implementation is available for the algorithms described in the paper, collecting the information given by the Amalthea model and producing a response time analysis for the task system as well as valid upper bounds on the worst-case end-to-end latency of the effect chains. The source code and the tool may be downloaded from our website[2].

[2] http://hipert.mat.unimore.it/FMTV16/

We already identified possible future enhancements of our approach, for all of the addressed challenges:

1) For Challenge I, we intend to explore how enlarging the non-preemptive region beyond runnable boundaries may improve the response time of the runnables, and related effect chains, as shown in [8];
2) for Challenge II, we aim at exploring approaches based on memory access pattern, such as [4], [5], [6], to improve the computed memory access delays;
3) for Challenge III, we intend to enhance our Java implementation with automatic placement functions to minimize the end-to-end latencies of selected effect chains.

Finally, and most importantly, we plan to apply co-scheduling techniques recently proposed in [9], [10] to avoid conflicting access by design, significantly reducing the penalties due to memory accesses. We believe that the proposed use-case may be a useful benchmark to test the efficiency of co-scheduling approaches.

Acknowledgment

This work was supported by the HERCULES Project, funded by European Union's Horizon 2020 research and innovation program under grant agreement No. 688860

References

[1] A. Davare, Q. Zhu, M. D. Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *2007 44th ACM/IEEE Design Automation Conference*, June 2007, pp. 278–283.

[2] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Real-Time Systems Symposium, 1990. Proceedings., 11th*, Dec 1990, pp. 201–209.

[3] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, Feb 2013.

[4] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, "Response time analysis of cots-based multicores considering the contention on the shared memory bus," in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, Nov 2011, pp. 1068–1075.

[5] D. Dasari, V. Nelis, and B. Akesson, "A framework for memory contention analysis in multi-core platforms," *Real-Time Systems*, pp. 1–51, 2015.

[6] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 741–746.

[7] A. Marongiu, P. Burgio, and L. Benini, "Supporting openmp on a multi-cluster embedded mpsoc," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 35, no. 8, pp. 668–682, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2011.08.010

[8] M. Bertogna, G. Buttazzo, and G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions," in *Proceedings of 32nd IEEE Real-Time Systems Symposium (RTSS 2011)*, Vienna, Austria, December 2011.

[9] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators," in *Proceedings of the CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST'15)*, October 2015.

[10] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Memory-processor co-scheduling in fixed priority systems," in *Proceedings of the 23rd International Conference on Real-Time Networks and Systems (RTNS15)*, Lille, France, November 2015.