International workshop on

Real-time computing and distributed systems in emerging applications

Reaction 2016

Colocated with IEEE RTSS
Porto, Portugal
November 29th, 2016

Luis LINO FERREIRA and Marisol GARCÍA-VALLS
editors
Preface

This volume contains the papers presented at Reaction 2016: 4th International Workshop on Real Time Computing and Distributed Systems in Emergent Applications held on November 29, 2016 in Porto, Portugal. It is the 4th edition of this event that started in 2012 in San Juan de Puerto Rico. Then, the 2013 edition took place in Vancouver, Canada, and on 2014 we met in Rome, Italy.

Reaction focuses on practical approaches to design and implement reliable, time-sensitive distributed systems, from hard real-time to quality of service. The aim of the workshop is to address some of the new challenges of current and future distributed real-time systems. Reaction provides a forum where contributions from different (though related) research communities can be presented around the concept of distributed real-time systems.

A total of eight papers have been accepted that cover a broad domain range with the goal of attracting different complementary research lines.

We hope that this selection is enjoyed by the attendees and readers.

Reaction 2016 PC chairs.


Luis Lino Ferreira Marisol García-Valls
Program Committee

Luca Abeni University of Trento
Karl-Erik Artzen Lund University
Neil Audsley University of York
Faruk Caglar Meliksah University
Antonio Cassimiro Universidade Lisboa
Fabio Checconi IBM TJ Watson
Tommaso Cucinotta Scuola Superiore Sant’Anna
Julien Delange Carnegie Mellon University
Jerker Delsing Lulea University of Technology
Manuel Díaz Universidad de Málaga
Marisol García-Valls University Carlos III of Madrid
Annirudha Gokhale Vanderbilt University
Javier Gutiérrez Universidad de Cantabria
Luis Lino Ferreira ISEP/IPP
Saad Mubeen Mälardalen Real-Time Research Center, Mälardalen University
Luis Miguel Pinho CISTER Research Centre/ISEP
Frank Singhoff University of Brest
Mikael Sjödin Mälardalen University
Pal Varga Budapest University of Technology
Thomas Vergnaud Thales
# Table of Contents

Analyzing End-to-End Delays in Automotive Systems at Various Levels of Timing Information ................................................................. 1  
*Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam and Thomas Nolte*

Using DDS middleware in distributed partitioned systems ......................... 7 
*Marisol García-Valls, Jorge Domínguez-Poblete and Imad Eddine Touahria*

Towards Architectural Support for Bandwidth Management in Mixed-Critical Embedded Systems .................................................. 13  
*Georgios Kornaros, Ioanins Christoforakis and Maria Astrinaki*

Programming Support for Time-sensitive Adaptation in Cyberphysical Systems ............. 19  
*Mikhail Afanasov, Aleksandr Iavorskii and Luca Mottola*

FLOPSYNC-QACS: Quantization-Aware Clock Synchronization for Wireless Sensor Networks ........................................................................... 25  
*Federico Terraneo, Alessandro Vittorio Papadopoulos, Alberto Leva and Maria Prandini*

Cost minimization of network services with buffer and end-to-end deadline constraints .... 31  
*Victor Millnert, Johan Eker and Enrico Bini*

Go-RealTime: A Lightweight Framework for Multiprocessor Real-Time System in User Space ................................................................................. 37  
*Zhou Fang, Mulong Luo, Fatima M. Anwar, Hao Zhuang and Rajesh Gupta*

DTFM: A Flexible Model for Schedulability Analysis of Real-Time Applications on NoC-based Architectures .................................................. 43  
*Mourad Dridi, Stéphane Rubini, Frank Singhoff and Jean-Philippe Diguet*
Analyzing End-to-End Delays in Automotive Systems at Various Levels of Timing Information

Matthias Becker*, Dakshina Dasari†, Saad Mubeen*, Moris Behnam*, Thomas Nolte*
*MRTC / Mälardalen University, Sweden
† Robert Bosch GmbH, Renningen, Germany

Abstract—Software design for automotive systems is highly complex due to the presence of strict data age constraints for event chains in addition to task specific requirements. These age constraints define the maximum time for the propagation of data through an event chain consisting of independently triggered tasks. Tasks in event chains can have different periods, introducing over- and under-sampling effects, which additionally aggravates their timing analysis. Furthermore, different functionality in these systems, is developed by different suppliers before the final system integration on the ECU. The software itself is developed in a hardware agnostic manner and this uncertainty and limited information at the early design phases may not allow effective analysis of end-to-end delays during that phase. In this paper, we present a method to compute end-to-end delays given the information available in the design phases, thereby enabling timing analysis throughout the development process. The presented methods are evaluated with extensive experiments where the decreasing pessimism with increasing system information is shown.

I. INTRODUCTION

Automotive systems are getting complex with respect to traditional components like the Engine Management System (EMS) as well as modern features like assisted driving. While the increase in the EMS complexity is attributed to newer hybrid engines and stricter emission norms, assisted driving requires the perfect convergence of various technologies to provide safe, efficient and accurate guidance. This has led to software intensive cars containing several million lines of code, spread over up to hundred Electronic Control Units (ECU) [1].

Given this complexity, over the last decades, standards like AUTOSAR [2] have been formulated in order to provide a common platform for the development of automotive software. These standards allow software components provided by different suppliers to be integrated on the same ECU, since they provide for a hardware agnostic software development. Such robust interfaces enable designers to design software at early stages without knowledge of the hardware platform which it will be eventually executed. Thus, during the development it is often not known which other applications share the same execution platform.

Most of these automotive applications typically have strict real-time requirements – it is not only important that a computation result is the correct result, but also that the result is presented at the correct time. In addition to the timing requirements for each task execution (i.e. the tasks deadline), these applications often have constraints for the data propagation through a chain of tasks, so called end-to-end timing constraints, one of which is the age constraint. The age constraint specifies the maximum time from reading a sensor value until the corresponding output value is produced at the end of the chain. This kind of constraint is especially important for control systems, such as the EMS, where it directly influences the quality of control.

Many design decisions have direct influence on the data age. Thus, bounding the data age of a chain early during the design process can potentially avoid costly software redesigns at later development stages. The analysis gets complex as a chain may consist of tasks with different periods leading to over- and under-sampling situations. Most of the available analysis methods for such systems analyze existing schedules [3] and thus they are not applicable during early phases. In [4], a generic framework to calculate the data age of a cause-effect chain is presented, which targets single processor systems and is agnostic of the scheduling algorithm used. In this paper we show how this analysis can be extended to cater the needs of the complete development process of automotive applications. The increased system information during the design process can thus be used to obtain end-to-end latencies with decreasing pessimism at various levels of timing information.

Contributions: In this work, we extend our earlier work on analyzing end-to-end delays among multi-rate effect chains [4] to utilize the information available in different development stages. Specifically, we highlight the generic nature of the framework by showing how extensions with varied levels of information can be used to compute the maximum data age given the:

1) Knowledge of offsets: The analysis for systems without knowledge of the schedule is extended to allow for task release-offset specifications.
2) Knowledge of the scheduling algorithm (like Fixed Priority Scheduling (FPS)): Most ECUs utilize operating systems which schedule tasks based on FPS. This allows to utilize existing analysis for such systems to determine worst-case response times of the individual tasks. It is then shown how the concepts of the analysis can be adapted to account for this information.
3) Knowledge of the exact schedule: Similar to most of the existing end-to-end delay analyses, we show how the exact schedule can be used to determine the exact delays with low computational overheads.
4) Knowledge of the communication semantic: We extend
the analysis to incorporate Logical Execution Time (LET), an important paradigm guiding how and when data is exchanged between tasks of an automotive application [5]. Finally, we compare these different scenarios with extensive evaluations, considering i) the tightness of the computed bounds and ii) the computation time for the analysis.

II. RELATED WORK

The end-to-end timing constraints found in automotive multi-rate systems were first described in [6]. Here, the authors describe the different design phases and link them to EAST-ADL [7] and AUTOSAR [2]. An increased level of system knowledge during the consecutive design phases is outlined.

A method to compute the different end-to-end delays of multi-rate cause-effect chains is presented in [3]. In addition, the authors relate the reaction delay to "button to reaction" functionality and the maximum data age delay to "control" functionality. In this work the focus lies on the maximum data age and hence on control applications.

A model-checking based technique to compute the end-to-end latencies in automotive systems is proposed in [8]. The authors generate a formal model based on the system description which is then analyzed.

The end-to-end timing analysis in an industrial tool suite is discussed in [9]. Two different activation methods are discussed; trigger chains, where a predecessor task triggers the release of a successor task, and data chains, where tasks are individually triggered and hence over- and under-sampling may occur. In this work we focus on chains with the latter activations.

End-to-end delays in heterogeneous multiprocessor systems are analyzed in [10]. Ashjae et al. [11] propose a model for end-to-end resource reservation in distributed embedded systems, and also present the analysis, based on [3], for end-to-end delays under their model.

Additionally, several industrial tools implement the end-to-end delay analysis for multi-rate effect chains [12], [13], [14]. However all of the discussed works require system information which is only available in the implementation level. In [4], a scheduling agnostic end-to-end delay analysis for data age is described, where only information about the tasks of the cause-effect chains is required. In this work, we extend the results presented in [4], and show that by augmenting information available during the different design phases, we can analyze the maximum data age with decreasing degree of pessimism.

III. SYSTEM MODEL

This section introduces the application model, inter task communication mechanisms, and the notion of cause-effect chains as used in this work.

A. Application Model

We model the application as a set of periodic tasks $\Gamma$. Each task $\tau_i \in \Gamma$ can be described by the tuple $\{C_i, T_i, \Psi_i\}$, 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 $\tau_i$ is equal to $T_i$. A task can further have a release offset $\Psi_i$. 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 $\tau_i$ executes a number of jobs during one $HP$, where the $j^{th}$ job is denoted as $\tau_{i,j}$.

B. Communication between Tasks

In this work inter-task communication is realized via shared registers. This is a common form of communication and can be found in many industrial application areas. In such a communication model, a sending task writes an output value to a shared register. Similarly, a receiving task reads the current value of this register. Hence, there is no signaling between the communicating tasks, and a receiving task always consumes the newest value (i.e. last-is-best).

In order to increase determinism, tasks operate on the read-execute-write model. Meaning, a task reads all its input values into local copies before the execution starts. During the execution phase only those local copies are accessed. Finally, at the end of the execution the task writes the output values 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, at the beginning and end of the tasks execution respectively. This is a common communication form found in several industrial standards (i.e. in AUTOSAR this model is defined as the implicit communication [15]. Also the standard IEC 61131-3 for the automation systems defines similar communication mechanisms [16]).

C. Cause-Effect Chains

For many systems it is not only important that the individual tasks execute within their timing constraints but also that the data propagates through a chain of tasks within certain time bounds. One example is the Air Intake System (AIS), which is part of the Engine Management System (EMS) in a modern car. For a smooth operation, the air and fuel mixture inside the engine must be controlled and the AIS is responsible for injecting the correct amount of air. To do so, an initial sensor task periodically samples the position of the pedal, followed by a number of control tasks that process this information, and finally an actuator task actuates the throttle to regulate the amount of air inside the engine. For the control algorithm, it is important that the sensed input data is fresh in order to reach the required control quality. Hence, the time from reading the data until the actuation is subject to delay constraints in addition to the task’s individual timing constraints.

In AUTOSAR such constraints are described by the cause-effect chains [17]. For a task set $\Gamma$, a set of cause-effect chains $\Pi$ can be specified. Where $\Pi$ contains the individual cause-effect chains $\zeta_i$. A chain $\zeta_i$ is represented by a Directed Acyclic Graph (DAG) $\{\mathcal{V}, \mathcal{E}\}$. The nodes of the graph are represented by the set $\mathcal{V}$ and contain the tasks involved in the cause-effect chain. The set $\mathcal{E}$ includes all edges between the nodes. An edge from $\tau_i$ to $\tau_k$ implies that $\tau_i$ has at least one output variable which is consumed by $\tau_k$. A cause-effect chain can have forks and joins, but the first and the last task in the chain must be the same for all possible data paths. To simplify the analysis, chains with fork/join operations are decomposed into individual sequential chains. Hence, all cause-effect chains in $\Pi$ are sequential.
End-to-End Timing Requirements: For each cause-effect chain, an end-to-end timing requirement can be specified. Several end-to-end timing requirements are defined for automotive systems [17], [7]. In this work the data age, as the most important timing requirements for control systems, is examined. A detailed discussion of end-to-end delays is provided in [3].

For the 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, $\tau_1$, $\tau_2$, and $\tau_3$. All tasks are part of a cause-effect chain in this order. Note that $\tau_1$ and $\tau_3$ are activated with a period of $T = 2$, while $\tau_3$ 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 $\tau_1$ is consumed by the first instance of $\tau_2$, the data produced by the second instance of $\tau_1$ is overwritten before $\tau_2$ has the chance to consume it. Similarly, the data produced by the first instance of $\tau_2$ is consumed by the first instance of $\tau_3$. Since no new data is produced before the second instance of $\tau_3$ is scheduled the same data is consumed. In the example, this constitutes the maximum data age, from sampling of the first instance of $\tau_1$ until the last appearance of the data at the output of the second instance of $\tau_3$.

IV. RECAP: CALCULATION OF DATA PROPAGATION PATHS

In this section we recapitulate the calculations of data propagation paths for systems without prior knowledge of the schedule, as this is basis for the work presented in this paper. For a more in depth explanation a reader is referred to [4].

A. Reachability between Jobs

The main concept to decide if data can be propagated between two distinct jobs are the read interval and the data interval. For a job $\tau_{i,j}$, the read interval is defined as the interval starting from the earliest time a job can potentially read its input data ($R_{\min}(\tau_{i,j})$) until the last possible time a job 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 a job can be available ($D_{\min}(\tau_{i,j})$) up to the latest time a successor job of the same task overwrites the data ($D_{\max}(\tau_{i,j})$). Hence, the read interval $R_{i,j}$ is the interval $[R_{\min}(\tau_{i,j}), R_{\max}(\tau_{i,j})]$, and the data interval is $[D_{\min}(\tau_{i,j}), D_{\max}(\tau_{i,j})]$.

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:

\[
\text{Follows}(\tau_{i,j}, \tau_{k,l}) = \begin{cases} 
\text{true,} & \text{if } R_{i,j} \cap D_{i,j} \neq \emptyset \\
\text{false,} & \text{otherwise}
\end{cases}
\]

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 $\tau_i$, as shown in Fig. 2 is followed by a job of a task $\tau_k$. $\tau_k$ is released with the same period as that of $\tau_i$, but the execution time of $\tau_i$ is shorter than the one of $\tau_k$. $\text{Follows}(\tau_{i,1}, \tau_{k,1})$ returns true and indicates that $\tau_{k,1}$ can potentially consume the data of $\tau_{i,1}$. 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 $\tau_{i,1}$ earliest at time $D_{\min}(\tau_{i,1})$. Consequently, this data can earliest be available as output data of $\tau_{k,1}$ at time $D_{\min}(\tau_{i,1}) + C_k$. $D'_{\min}(\tau_{k,l}, \tau_{i,j})$ 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 then 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.

The function starts at the first level of the cause-effect chain, for the initial job all jobs of the second task of the chain are
found where Follows() returns true. To these nodes a logical data path is created. The same principle is applied from each of these nodes to the jobs of the next lower level of the cause-effect chain. Once the last level is reached all possible paths are calculated and the function returns. Interested readers are referred to [4] for a detailed explanation.

C. Constructing Data Propagation Paths and Maximum Data Age

For one data path the maximum end-to-end latency, and thus the data age, can be computed as follows, where \( \tau_{\text{start}} \) is a job of the first task of the cause-effect chain, and \( \tau_{\text{stop}} \) is a job of the last task of a cause-effect chain:

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

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

V. REACHABILITY BETWEEN JOBS IN DIFFERENT SYSTEMS

The basic computation of data age latencies without prior knowledge of the schedule can result in pessimistic results. Many of the computed data propagation paths may not occur since scheduling algorithms impose a recurring order of jobs in each hyperperiod of the task set. In this section, modifications of the read and data interval are presented to reflect the behavior of systems with more elaborate knowledge on the scheduling decisions. One key observation is that the presented method to calculate the different data paths and the maximum data age is independent of the concrete system model as long as the read and data intervals are adjusted accordingly. Table I depicts the required changes to the read and data interval for different levels of system information. The remainder of this section discusses these required modifications in more detail.

### A. Knowledge of Task Release Offsets

In our earlier work [4], no task release offsets were considered in the analysis. In order to account for known offsets, the read interval needs to be adjusted. Given an offset \( \Psi \), a job of a task can now read its input data only after \( \Psi \) time units after the start of its period. The end of the read interval is unchanged at \( C \) time units before the next period starts. Since \( D_{\text{min}} \) and \( D_{\text{max}} \) are described by \( R_{\text{min}} \) and \( R_{\text{max}} \), no direct changes are required in the formulation.

### B. Reachability in known Schedules

Many real-time systems deploy time-triggered schedules in order to guarantee a deterministic system behavior. In such a schedule it is known at design time when the different jobs of the different tasks are executed. Thus, a complete knowledge of the system is available. On the other hand, for dynamically scheduled systems it is often possible to compute the Worst-Case Response Time (WCRT). In that case the exact execution times of a task are not known but the earliest and latest time a task can execute is known.

1) **Schedule is Available:** Let’s assume an offline schedule is available for the system. So for each job \( \tau_{i,j} \) of the task set its exact start time is known as \( \text{start}_{i,j} \), and similarly its finishing time is known as \( \text{end}_{i,j} \), see Fig. 3. With this additional knowledge the read and data interval can be adjusted as shown in Table I.

Since the start of the jobs execution, and hence the time it reads its input data, is known, the read interval collapses to a point. This also leads to smaller data intervals, resulting to no overlap between consecutive jobs.

#### Fig. 3: Read and data intervals of consecutive jobs of \( \tau_i \) if the exact schedule is available.

2) **Worst Case response Time is Available:** For systems where the WCRT of a task \( \tau_i \) is known as \( WCRT_i \), the read and data interval can be adjusted to account for this more accurate system information (see Fig. 4). The modifications of the read interval mainly reflect the possible execution of a job during its execution window (bounded by the WCRT).

#### Fig. 4: Read and data intervals of consecutive jobs of \( \tau_i \) if WCRT of the tasks are available.

---

<table>
<thead>
<tr>
<th></th>
<th>No Knowledge</th>
<th>Exact Schedule Known</th>
<th>WCRT Known</th>
<th>LET Execution Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_{\text{min}}(\tau_{i,j}) )</td>
<td>( \Psi_i + (j - 1) \cdot T_i )</td>
<td>( \text{start}_{i,j} )</td>
<td>( \Psi_i + (j - 1) \cdot C_i )</td>
<td>( (j - 1) \cdot T_i )</td>
</tr>
<tr>
<td>( R_{\text{max}}(\tau_{i,j}) )</td>
<td>( j \cdot T_i - C_i )</td>
<td>( \text{end}_{i,j} )</td>
<td>( R_{\text{min}}(\tau_{i,j}) )</td>
<td>( R_{\text{min}}(\tau_{i,j}) )</td>
</tr>
<tr>
<td>( D_{\text{min}}(\tau_{i,j}) )</td>
<td>( R_{\text{min}}(\tau_{i,j}) + C_i )</td>
<td>( \text{end}_{i,j} )</td>
<td>( R_{\text{min}}(\tau_{i,j}) + C_i )</td>
<td>( j \cdot T_i )</td>
</tr>
<tr>
<td>( D_{\text{max}}(\tau_{i,j}) )</td>
<td>( R_{\text{max}}(\tau_{i,j+1}) + C_i )</td>
<td>( \text{end}_{i,j+1} )</td>
<td>( R_{\text{max}}(\tau_{i,j+1}) + C_i )</td>
<td>( (j + 1) \cdot T_i )</td>
</tr>
</tbody>
</table>
C. Reachability in the LET model

The LET model provides an abstraction to the system designer by temporally decoupling the communication among tasks from the tasks execution. In this model, the input values of a task are always read at the release of the task. The output values become available once the next period starts. In Fig. 5 these points are highlighted by the thick orange line below the arrows marking the job releases. This temporal decoupling of communication and execution has significant advantages for the end-to-end delay calculations.

The periodic access to all input variables at the beginning of the period collapses the read interval to a point. The data interval is also defined, making the output data available for exactly the period after the jobs execution. These modifications are shown in Table I. As can be seen, all descriptions are independent of the actual execution time of the job.

Fig. 5: Read and data intervals of consecutive jobs of $\tau_i$.

D. Discussion

All presented modifications affect solely the read and/or data intervals of the jobs. Hence, the existing calculations for the maximum data age, as presented in [4] and recapitulated in Section IV, can be applied without modification. This fact allows the system designer to perform the calculation of maximum data age during early design phases, where only limited knowledge is present, or during the end of the system design where more complete system knowledge can be obtained.

A tradeoff between required system knowledge and accuracy of the obtained maximum data age exists. For systems with exact knowledge, and for the LET system, it can be observed, that data intervals of different jobs of the same task are never overlapping with each other. This means that it is always certain which data is consumed by a job and thus, all data paths which are computed are observed in the execution of the system.

Lemma 1. If it holds for all tasks in a chain, that the read intervals of a task are reduced to a point (i.e. $R_{\min}(\tau_{i,j}) = R_{\max}(\tau_{i,j})$), then the calculated end-to-end delays are exact. Here “exact” means that all calculated end-to-end delays are observed during the execution of the real system.

Proof. From the definition of the read- and data-interval in Section V we can see that once $R_{\min}(\tau_{i,j}) = R_{\max}(\tau_{i,j})$ the resulting data intervals of consecutive jobs are not overlapping. Given that data intervals are not overlapping and read intervals are reduced to points, the function $\text{Follows}(\tau_{i,j}, \tau_{h,k})$ only returns true for the jobs which actually consume the respective data during the execution of the real system.

VI. Evaluation

This section presents the evaluation of the proposed approaches to analyze the end-to-end delay based on various levels of system knowledge. First the computed worst-case data-age based on various information levels is compared. Further the required computation time to perform the analysis at the presented information states is evaluated.

A. Experimental Setup

The analyzed cause-effect chains for the experiments are generated according to the automotive benchmarks described in [18]. The task periods are uniformly selected out of the set $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ ms. The individual task utilization is computed by UUnifast [19]. As stated in [18], an individual cause-effect chain is comprised of either 2 or 3 different periods, where tasks of the same period appear in sequence in the chain. Note that not all period-pairs are valid predecessors in a cause-effect chain [18], which is accounted for during the random generation of the cause-effect chain. For each of the presented data points 1000 random cause-effect chains are examined.

Fixed priority scheduling is used as scheduling algorithm in order to compute the response times and the information for the exact schedule of the tasks. Priorities are assigned based on the Rate Monotonic [20] policy, where priorities between tasks of the same period are assigned in arbitrary order. For the evaluation of the systems with required response times or known schedule, the response times are calculated based on the well-known analysis for task scheduling presented in [21], and the schedule is generated by simulation of the tasks execution.

B. Pessimism during the Individual Design Phases

The first experiment examines systems under all four presented information states, no information, response times, known schedule, and LET model. The same cause-effect chain is analyzed with increased available system information. The system contains 30 tasks while the cause-effect chain is comprised of 4 to 10 tasks in the case of two activation patterns (i.e. periods), and 6 to 15 tasks in the case of three activation patterns and the system utilization is set to 80%. The results are presented in Fig. 6. The calculated end-to-end latencies are normalized in respect to the chains hyperperiod and shown on the y-axis. The decreased pessimism in the analysis with increased system knowledge is visible for all experiments. The computed worst-case data age of the same scenario includes lesser pessimism from systems with no prior information to systems with known response times up to systems where the schedule is available. Additionally, we present the maximum data age under the LET model, which behaves close to the computed results based on response times in our setting. The difference of the execution semantic becomes visible when comparing with the results for known schedules. Both results are exact results under the respective execution semantic but the observed maximum data age for the LET model is two times as large as the value for the known schedule.
C. Analysis of the Computation Time

One of the main improvements behind the presented approaches is to only modify the input set while the analysis framework is unchanged. In this experiment we evaluate the required computation time for the analysis under the different levels of system knowledge. The system contains 30 tasks while the cause-effect chain under analysis has a length of 4 to 10 tasks with two involved activation pattern. All experiments were performed on a system containing an Intel i7 CPU (4 cores at 2.8GHz), and 16GB of RAM. The results are shown in Fig. 7. The two scenarios with exact knowledge (i.e. the known schedule and the LET model) have very low analysis times with almost no increase with increasing length of the chain under analysis. On the other hand, the scenarios with less system information experience an exponential increase in analysis time. This can be explained by the increased uncertainty due to overlapping data-intervals, which leads to multiple possible successors which all need to be checked by the algorithm.

Fig. 7: Average analysis time for cause-effect chains with 2 involved periods.

VII. CONCLUSION AND OUTLOOK

In this work we have shown how to utilize the different levels of system information available during the design of automotive systems in order to compute the maximum data age of a cause-effect chain. This is done by extending the analysis method presented in [4] by adjusting the read- and data-intervals, which are used as input values of the analysis, to reflect the increase in system knowledge. A clear trade-off can be observed between the required information for the analysis and the pessimism in the obtained results. Future work focuses on the analysis of maximum data age over a cause-effect chain which is distributed over multiple nodes, connected by a network.

ACKNOWLEDGMENT

The work presented in this paper is supported by the Swedish Knowledge Foundation (KKS) through the projects PREMISE, DPAC, and PreView.

REFERENCES

Using DDS middleware in distributed partitioned systems

Jorge Domínguez-Poblete, Marisol García-Valls, Imad Eddine Touahria
Universidad Carlos III de Madrid
Av. de la universidad 30
28911 Leganés, Madrid, Spain
{jdominguez,mvalls}@it.uc3m.es

Abstract—Communication middleware technologies are slowly being integrated into critical domains that are also progressively transitioning to partitioned systems. Especially, avionics systems have transitioned from federated architectures to IMA (Integrated Modular Avionics) standard that targets partitioned systems to comply with the requirements of cost, safety, and weight. In the future developments, it is fully considered the integration of middleware to support data communication and application interoperability. As specified in FACE (Future Airborne Capability Environment), middleware will be integrated into mixed criticality systems to ease the development of portable components that can interoperate effectively. Still nowadays, in real-time environments, communication middleware is perceived as a source of unpredictability; and still there are very few contributions that present real applications of the integration of communication middleware into partitioned systems to support distribution.

This paper describes the usage of a publish-subscribe middleware (precisely, DDS –Data Distribution Service for real-time systems) into a fully distributed partitioned system. We explain the design of a reliable communication setting enabled by the middleware, and we exemplify it using a distributed monitoring application for an emulated partitioned system with the goal of obtaining the middleware communication overhead. Implementation results show stable communication times that can be integrated in the resource assignment to partitions.

I. INTRODUCTION

Communication middleware and virtualization technologies are two main contributions to the development and maintainability of software systems as well as to machine consolidation [2]. These were initially used in mainstream applications, but are progressively entering into the critical environments and complex systems, where their role is increasingly important. In fact, in the avionics domain, the combination of IMA [9] and FACE [4] require the usage of both virtualization technologies to develop partitioned systems and middleware to ease interoperability and portability of components. This satisfies key requirement regarding cost, space, weight, power consumption, and temperature.

On the one hand, middleware brings in the capacity to abstract the low-level details of the networking protocols and the associated specifics of the physical platforms (e.g. endianness, frame structure, and packaging, among others). Consequently, the productivity of systems development is augmented by easing the programmability, maintainability, and debugging.

On the other hand, the penetration of virtualization technology has opened the door to the integration of heterogeneous functions over the same physical platform. This effect of virtualization technology has also arrived to the real-time systems area. The design of mixed criticality systems (MCS) [7] is an important trend that supports the execution of various applications and functions of different criticality levels [6] in the same physical machine. The term criticality refers to the levels of assurance over the system execution in what concerns failures. For example, in avionics systems, software design follows DO-178B that is a de facto standard for software safety; software is guided by DAL (Design Assurance Levels), and failure conditions are categorized against their consequences: from catastrophic (DAL A) to no effect (DAL E). Then, an MCS is one that has, at least, two functions of different criticalities on the same physical machines.

Over the past 30 years, middleware technology has been applied in critical domains but in those subsystems of lower criticality levels. This is the case of, e.g., CORBA applied to combat systems [25] or, recently, DDS [21] applied to control of interoperability of unmanned aircraft and air traffic management 1, mainly for ground segment control. Still middleware is mostly used directly on bare machine deployments; yet it is not used in partitioned software systems.

This paper provides an initial design of a fully distributed partitioned deployment that integrates DDS middleware. We exemplify this concept on a data monitoring application that has been developed to provide hands on the actual technology, to analyze the temporal behavior of the overall distributed partitioned setting. The system is fully distributed across different physical machines to perform data sampling and transmission that is later received, processed and displayed at a remote node. The nodes of the monitoring system emulate a mixed criticality system. The setting replicates that of a partitioned system in a FACE compliant architecture. We concentrate on the design of the software stack and the analysis of the middleware performance over an Ethernet network that emulates an AFDX (Avionics Full Duplex) [5] compliant communication.

The paper is structured as follows. Section II presents the work background and a selected related contributions, including concepts and technologies relative to partitioned systems

1http://www.atlantida-cenit.org
and distribution middleware technologies for critical domains. Section III analyzes the most important characteristics and properties of DDS for partitioned systems within FACE. Section IV presents the design of the distributed partitioned system, illustrated for a remote monitoring application. Section V provides the implementation of the system and presents the results obtained for the communication. Section VI draws some conclusions and describes the future work.

II. BACKGROUND

IMA has been a very successful approach to transition from the former federated architectures to a more efficient design and final deployment into avionics systems. In this context, different standards are proposed to facilitate componentization, portability and interoperability at different levels of a system. In this way, ARINC 653 standard decouples the real-time operating system platform from the application software. For this purpose, it defines an APEX (APplication EXecutive) where each application software is called a partition having its own memory space. Each partition has dedicated time slots allocated through the APEX API, so that each partition can have multi-tasking and its own scheduling policy. Overall, the execution is embodied in a hierarchical scheduling policy where the top level is a cyclic schedule. Current work is to enhance ARINC 653 for multi-core processor architectures. The underlying network is AFDX that uses commercial technology with redundancy to support safe transmission. The integration of networked and distributed systems follows ARINC 429 that is the data bus defining the characteristics of the data exchanges among the connected subsystems. It defines the physical and electrical interfaces and a data protocol for a local avionics network.

In other real-time systems, network scheduling typically relies on either: network scheduling off-line network transmission plan given the a-priori knowledge of the generated traffic (e.g. [26]); architectures such as TTA (Time Triggered Architecture) [17]; or distributed component models highly related to the hardware on-chip design such as Genesis [19]. Nevertheless, in the last decade the trend in the development of complex systems is to move to more productive ways of designing the communication and interaction. The avionics industry has developed FACE (Future Airborne Capability Environment) standard to facilitate the development and easy integration of portable components. The communication middleware is given a key role as an interoperability enabler. Different technological choices can be used in FACE such as CORBA, Web services, or DDS, among others. The most popular technology at the moment is (probably) OMG’s DDS standard [21] that has been applied in a number of domains such as remote systems control [24]. It provides an asynchronous interoperability via a publish-subscribe (P/S) paradigm that is data-centric, and the communication can be fine tuned through quality of service (QoS) policies.

There are some related works such as a thorough performance study of DDS for desktop virtualization technologies [22] but was not dealing with partitioned systems; or the execution of DDS over a real-time hypervisor [20] although the actual network stack processing was not measured; or [13], [14] for network level P/S evaluation, and [15] for bare machine deployments. Overall, there is not sufficient analysis on the actual execution characteristics of specific middleware technologies in general partitioned environments. Moreover, there are no practical design models of partitioned systems that can comprehensively put forward the required software levels integration and there actual performance results. This paper contributes in this direction with a practical design of a distributed partitioned environment based on DDS, providing a study of the behavior of a partitioned system that communicates using this technology.

III. MIDDLEWARE WITHIN A PARTITIONED SYSTEM

A. Middleware in FACE Standard

FACE standard defines the software computing environment and interfaces designed to support the development of portable components across the general-purpose, safety, and security profiles required by the avionics domain. Its goal is to define the interaction points between the different technologies that ease systems integration. Actually, FACE uses industry standards for distributed communications, programming languages, graphics, operating systems, among other areas. Version 2.0 further promoted application interoperability and portability, with enhanced requirements for exchanging data among FACE components; also, v2.0 emphasizes on defining common language requirements for the standard. Precisely, the operating system segment of FACE defines different levels: the portable components segment (the applications), the transport services segment (where middleware technologies are employed for interoperability), the platform specific services segment (for the common services of a given domain), and the I/O services segment (related to the low-level adapters and drivers to access peripherals including the actual network).

At the transport services segment, many technological choices can be employed (e.g. CORBA, Web services, DDS, etc.). At the moment, the most widely accepted is DDS.

B. Data Distribution Service for real-time systems

The Data Distribution Service (DDS) is an OMG standard that provides a publish-subscribe (P/S), i.e., a decoupled interaction model among remote components. DDS relies on the concept of a global data space where entities exchange messages based on their type and content. Such entities are remote nodes or remote processes, although it is also possible to communicate from within the same local machine. Entities can take two roles; they can be publishers or subscribers of a given data type. Types are based on the concept of topics that are constructions supporting the actual data exchange. Topics are identified by a unique name, a data type and a set of QoS policies; also, they can use keys that enable the existence of different instances of a topic so that the receiving entities can differentiate the data source.

Applications organize the communicating entities into domains. Essentially, a domain defines an application range
where communication among related entities (an application) can be established. A domain becomes alive when a participant is created. A participant is an entity that owns a set of resources such as memory and transport. If an application has different transport needs, then two participants can be created. A participant may contain the following child entities: publishers, subscribers, data writers, data readers, and topics. Publishers and subscribers are an abstraction that manages the communication part, whereas the data writers and data readers are the abstractions that actually inject the data.

One of the most successful elements in DDS is the set of quality of service parameters that it defines, namely QoS policies. In fact, not all of them are related to the temporal behavior of the communication. Most QoS policies provide other guarantees over the data transmission. A short summary of policies that influence the communication time (marked as t) and others affecting the actual overhead of the system (marked as o) are provided in Table I. The entities to which they apply are also indicated.

<table>
<thead>
<tr>
<th>QoS policy</th>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline (t)</td>
<td>DR, DW</td>
<td>Max expected elapsed time between arriving data samples (unkeyed data) or instances (keyed data) or Max committed time to publish samples or instances</td>
</tr>
<tr>
<td>Resource Limits (o)</td>
<td>DP</td>
<td>Limit to the allocated memory (message queues for history, etc.). It limits the queue size for History when the Reliability protocol is used</td>
</tr>
<tr>
<td>History (o)</td>
<td>DR, DW</td>
<td>Stores sent or received data in cache. It affects Reliability and Durability (receive samples sent prior to joining) QoS policies</td>
</tr>
<tr>
<td>Latency Budget (t)</td>
<td>T, DR, DR</td>
<td>Indication on how to handle data that requires low latency. Provides a maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications</td>
</tr>
<tr>
<td>Time based Filter (t)</td>
<td>DR</td>
<td>Limits the number of data samples sent for each instance per a given time period</td>
</tr>
<tr>
<td>Transport priority (t)</td>
<td>DW</td>
<td>Establishes a given priority for the data sent by a writer. Underspecified: It is dependent on the actual transport characteristics and also supported by only some OSs</td>
</tr>
<tr>
<td>Reliability (o)</td>
<td>DR, DW</td>
<td>Global policy that specifies whether or not data will be delivered reliably. It can be configured on a per DataWriter/DataReader connection</td>
</tr>
</tbody>
</table>

**TABLE I: Subset of QoS policies affecting overhead**

**C. Communication in partitioned systems**

The mainstream design of a critical partitioned system consists of isolating the communications of a system into one single partition that is selected as the communications partition (see figure 1). Transmissions take place purely during the time of the cycle assigned to the partition. When any partition needs to transmit or receive data, it sends it to the communication partition via a shared memory space. When the kernel schedules the partition at its assigned time slot, the transmission takes place.

This design approach has not included middleware, but has used direct implementation of network protocols for ARINC 429. So, standards such as DDS are progressively considered into the actual implementations of critical partitioned environments in the context of FACE.

**IV. SYSTEM DESIGN**

A. Software design and middleware communication

Figure 2 shows the major software blocks for the distributed monitoring application. The system has two differentiated subsystems: System A that is a meteo proximity server and System B that is the Global Monitor.

Two sensors (humidity and temperature) gather data samples that are passed to System A that is able to perform an initial basic processing of the data to take basic decisions on the sensors operation and detect faults. System A passes these data to System B that is able to perform more complex analysis over the data, and also it is able to make decisions on the configuration of the sensors and the overall system operation. The proximity sensor can do an initial processing of the samples and later, it sends the data to the global monitor.

Systems A and B are two different physical machines that are partitioned emulating an ARINC 653 deployment. They
are connected via an Ethernet link as compliant with AFDX. All partitions can integrate the communication middleware, so that any of them can send or receive data to and from other either local or remote partitions as shown in figure 3. In this way, communications may take place during the time slots that the kernel assigns to each of the partitions.

![Diagram of System A and System B](image)

**Fig. 3: Proposal architecture- Distributed monitoring application design in a partitioned system**

System A has two partitions: $V_{A1}$ that monitors the sensor reads and performs basic analysis, and $V_{A2}$ that performs data displays to the operators. System B has two partitions: $V_{B1}$ that receives the sensor reads and performs complex analysis to guide the system operation, and $V_{B2}$ that performs data displays to the remote operators.

Partition $V_{A1}$ receives the sampled data, and it sends them to $V_{B1}$ through the appropriate topics: *temperature* and *humidity*, to send temperature and humidity values, respectively. Also, a *configuration* topic is used by system B to change the operation parameters of A such as the sampling rate.

The general implementation of the reliability mechanism for RTPS is the usage of acknowledgment messages (ACK). They are similar to a heartbeat mechanism and ACK/NACK. Using a sequence number and a storage space (cache), it is possible to know which messages have arrived to the subscriber per message stores the assigned sequence number, the history of the sent data values, and the history of whether the sample has been delivered to the reader.

### B. Temporal bounds of the communication within the partitions schedule

The hierarchical scheduling of partitions is related to the temporal bounds of the middleware communication across the partitions. In order to incorporate the transmission times to the schedule, the temporal bounds on the communication among the partitioned nodes are analyzed.

In IMA systems, temporal partitioning is guaranteed though partition windows. The operating system kernel applies a deterministic scheduling algorithm based on a static configuration file that indicates the time windows that are assigned to each partition. This is defined in different environments such as VxWorks [8] like other ARINC 653 compliant operating systems or the MultiPartes approach [1].

We undertake a local schedulability analysis for each system and consider the middleware cost within the partitions as follows. Let $\mathcal{V}$ be the set of $n$ partitions of a system such that $\mathcal{V} = \{V_k\}, \forall k = 1..n$. The execution requirements of each partition are expressed as follows. A partition $V_k$ requires to use the processor for $C$ time units every period of $T$ time units: $V_k = (C_k, T_k)$.

The execution life of a partitioned system follows a hierarchical approach of two main levels. At the top level,
there is a sequence of equal duration time slots namely major frames. Each major frame \( F^M \) is divided into a number \( (q) \) of time slots of equal duration called minor frames, \( F^m \): 
\[
F^M = \sum_{j=1}^{q} F^m.
\]
Each minor frame is divided into a number \( r \) of time slots of different durations \( (C_k) \). Each of these time slots is assigned to a specific partition \( V_k \): 
\[
F^m = \sum_{k=1}^{C_k} C_k.
\]
During its assigned time slot, a partition has uninterrupted access to common resources.

A part from the numbers provided by middleware vendors, a middleware stability analysis is performed off-line. The analysis considers the message size required by the specific application (1024KB in our case), the required load conditions, and the specific hardware. In the case of the used implementation, DDS RTI Connext, this yields efficient values and a stable behaviour in different situations. The maximum value of the middleware communication cost \( c_{mw} \) is associated to very exceptional situations \(< 0.1\), although possible. Consequently, the maximum value \( c_{mw} \) is considered in the schedule as part of the corresponding partition \( C_k \) that make use of it. For a partition, 
\[
c_{mw} = \frac{c}{\delta} C_k
\]
where \( \delta \) is the ratio between the overall partition time and the middleware cost.

V. Results

Experimental results are presented with the goal of assessing the cost of using DDS in a partitioned context for supporting remote communication across local and remote partitions. We intend to validate the suitability of the proposed system design; we analyze the system in terms of the communication time from the side of the invoking node. Even for an unreliable setting, we have measured a partition-level reliable communication implementation, i.e., including the client response reception. The goal is to show that the chosen middleware provides communication bounds for this distributed partitioned system. Also, results show the stability of the middleware in the partitioned system, that is compared against the control group (a bare machine deployment in an unreliable DDS configuration). Results of the execution of the implemented system show its feasibility, given the performance of the middleware in different scenarios over a sufficiently large number of trials to obtain the maximum values of the communication enabled by the middleware. The presented data is compiled from 1000 iterations to provide meaningful information. The communication results show the overhead of the whole processing stack of a partition [22].

The monitoring system is implemented with DDS RTI Connext 5.2.0 over two networked machines connected by a 100Mbps Ethernet link. The hardware of the physical nodes is a double core Intel E3400 at 2.6Ghz and with 2GB RAM.

Initially, a control group experiment was performed to analyze the performance of the middleware on a favorable situation, i.e., a distributed setting with a best effort configuration set at the data writer and data reader entities of the distributed partitions. This scenario was analysed on, both, empty load conditions and with progressive load up to 100%.

The communication cost was increased by 7x in the partitioned scenario. Nevertheless, it was shown that the average case times ranged from 3x to 7x lower than the worst case.

Figure 4 shows the results of a bare machine deployment of the emulated scenario for the distributed monitoring application. This scenario fully replicates the system and it also provides a DDS reliable communication configuration. The scenario of figure 4(a) (left side graph) is executed with no additional load, whereas 4(b) (at the right side) presents a progressive loaded system. Resulting times follow the expected pattern, and a similar phenomena as in the previous scenario is observed. Average behavior for (a) are between 3.7x and 5.33x smaller than the maximum times. Again, the communication proves to be very stable; dispersion is around 50µs for all the cases except for the largest message size that is 98.2µs. For the scenario (b) that shows progressive load, the average times are between 11x and 4x smaller that the worst case. Nevertheless, the system shows to be stable, and the dispersion is between 47.2µs and 123.20µs.

In the last scenario, the full fledged deployment of the system is shown. Figure 5 shows the reliable communication setting on a distributed partitioned system deployment with no additional load. Figure 6 shows the same scenario with progressive load.
the order of 300µs for the empty scenario and between 212 and 1400µs for the progressive load. Moreover, the worst case is, for the empty scenario, between 8x and 18x larger than the average case. For the progressive load tests, the average case is between 8x and 39x smaller than the worst case. Also, it can be seen that the minimum and average cases are very close. Results show that the fully distributed partitioned environment yields a stable execution and the communication overhead of the middleware follows the expected pattern.

VI. CONCLUSION

The paper describes the design of a distributed partitioned system that supports communication of remote partitions through DDS middleware. Topics are defined to support the data centric model and it is exemplified for a distributed monitoring application. The communication overhead caused by the middleware and the partitioned setting is analyzed for a sufficient number of trials. The novelty of the paper is the exhaustive trials on the specific DDS technology and the measurements that provide the overhead of the whole middleware processing stack. Results show that the communication is stable even in presence of very high loads. The average case times are significantly smaller that the worst case (from 8x to 39x) and dispersion is 1.4ms for the worst possible scenario. We show that the overhead can be obtained for its integration in the partition resource assignment.

ACKNOWLEDGMENT

This work has been partly supported by the Spanish Ministry of Economy and Competitiveness through projects REM4VSS (TIN 2011-28339) and M2C2 (TIN2014-56158-C4-3-P).

REFERENCES


Towards Architectural Support for Bandwidth Management in Mixed-Critical Embedded Systems

Ioannis Christoforakis, Maria Astrinaki and George Kornaros
Computer Engineering Department
Technological Educational Institute of Crete
71500, Heraklion, Crete, Greece
Email: kornaros@ie.teicrete.gr

Abstract—Mixed-critical platforms require an on-chip interconnect and a memory controller capable of providing sufficient timing independence for critical applications. Existing real-time memory controllers, however, either do not support mixed criticality or still fail to ensure negligible interference between applications. On the other hand, Networks-on-Chip manage the traffic injection rate mainly by employing complex techniques; either back-pressure based flow-control mechanisms or rate-control of traffic load (i.e. traffic shaping). This work proposes such a Traffic Shaper Module that supports both monitoring and traffic control at the on-chip network interface or the memory controller. The advantage of this Traffic Shaper Module is that at system level it provides guaranteed memory bandwidth to the critical applications by limiting traffic of non-critical tasks. The system is developed in the Xilinx ZYNQ7000 System-on-Chip while the measurements were captured on a Zedboard development board. By enabling the Traffic Shaper in our architecture we achieved fine-grain bandwidth control with negligible overhead, while providing bandwidth of only 0.5-5 percent less than the theoretical specified bandwidth.

I. INTRODUCTION

Nowadays, the modern Systems-on-Chips (SoCs) present a complex growth. Power, area and cost constraints [13], due to resource sharing, occur since the amount of applications mapped to a single system have been increasing. Applications characterized by real-time constraints exist alongside with applications with no constraints thus creating mixed time-criticality systems. Large numbers of applications run simultaneously in different combinations or use-cases and are dynamic, therefore they can be dynamically stopped or started. In order to guarantee satisfaction in application requirements, the system has the need to be verified for each of the use-cases. The verification complexity grows exponentially with the number of applications if they can be combined arbitrarily.

In the case of the active use-case change, in order to adapt to the new set of applications and requirements, the hardware components may have to be reconfigured. Even during reconfiguration, the running applications must demonstrate correct behavior. For this reason, the transitions between use-cases result in additional analysis complexity. Finding a common analysis model has been proved difficult when mixed time-critical applications and different models of computation are combined. Even if such a model exists, a single change in an application requires all use-cases, in which the application is active, to be re-verified in general.

A recent trend in real-time systems is to integrate tasks and components of different criticality levels on the same hardware platform. The objective of such mixed-criticality systems [1], [2] is to save space, weight, or energy by reducing the number of computation platforms, and at the same time to provide safety guarantees for the critical components of the system. Timing predictability is an important property when designing such systems, especially for the safety-critical components. Worst-case execution time (WCET) analysis [3], [4] becomes significantly easier if the hardware is more predictable. Many researchers have explored the possibility of designing predictable processors [5], [6], [7] and predictable memory hierarchies [9].

To provide guarantees for time-critical applications it is mandatory to precisely control the usage of system resources. This work describes a methodology for bandwidth management on Network-on-Chip (NoC) interfaces and system memory traffic, by using custom circuitry with minimal overhead. In particular, we demonstrate how to achieve this with the use of a specialized hardware Traffic Shaper Module (TSM); through attaching a TSM instance at any initiator Network Interface (NI) we control the number of accesses based on configured bandwidth and regulate the flow of traffic. We present a prototype system based on the ZYNQ7000 Processing System-on-Chip [12]. More specifically, we make the following contributions:

- introduce a low overhead Traffic Shaper Module at register-transfer level (RTL) that allows precise fine-grain traffic control in bytes/clock cycle; the block may provide the possibility for both monitoring, as well as for guaranteeing maximum bandwidth at an initiator on-chip network interface
- propose a technique that can be used to enforce bitrates different than what a physical interface is capable of
- prove that, by enabling the Traffic Shaper Module in our architecture, we achieve a performance overhead of just 0.5-5%, in contrast to the theoretical bandwidth

The rest of this paper is organized as follows. In Section II this work is positioned with respect to related works. Section III presents the proposed architecture and the developed traffic control methodologies, while experiments are shown in Section IV, and finally Section V concludes the paper.
II. RELATED WORK

There is a range of previous works for creating systems that support quality of service with mixed-criticality techniques both in register-transfer level and in the operating system and kernel level. For instance, several techniques for predictable real-time DRAM controllers have been proposed in previous works, although not all are suitable for mixed-criticality systems, since non-critical task performance cannot be sacrificed too much for critical task predictability. In a real-time system, bounded latency and interference must be considered, in addition to overall throughput and fairness. Kim et al [14] exhibited a DRAM controller that is able to separate the critical from the non-critical memory access groups (MAGs). They define algorithms for computing safe and tight upper bounds of worst-case latencies, resulting in predictable memory accesses for critical MAGs. In [15] developers designed a reconfigurable SDRAM controller. This controller offers both predictable and composable service, making it a suitable SDRAM resource for use in virtual platforms. Composability is enabled by the use of composable memory patterns combined with TDM arbitration, and they show that the worst-case performance degradation is negligible. The work in [16] deals with the problem of mixed time criticality workloads in the context of an SDRAM controller. The main idea is to allow the command scheduler to exploit locality that presents itself within the time window that naturally exists between opening and closing a row. The controller can exploit a fraction of the locality available in the request stream, without increasing the worst-case schedule length. Compared to a close-page policy (keeping a DRAM bank in idle state), the average execution time of the benchmark applications is reduced by 7.9% using the conservative open-page, while still satisfying the constraints of the Field Resource Tracker (FRT) application (guarantee enough worst-case performance to satisfy requirements).

Another technique is the one presented in [17]. This paper presents FlexPRET, a fine-grained multi-threaded processor designed to exhibit architectural techniques useful for mixed criticality systems (e.g. scratchpad memories). The designers claim that their system provides hardware-based isolation to hard real-time threads while allowing soft real-time threads to efficiently utilize processor resources. There are also timing instructions, extending the RISC-V ISA, that enable cycles to be reallocated to other threads when not needed to satisfy a temporal constraint. They even provide a concrete soft-core FPGA implementation, which is evaluated for resource usage.

A further option that can also be combined with our proposed mechanisms from our perspective is the implementation of algorithms for Mixed Criticality Optimization as in [18], where the optimization of mapping and partitioning is performed at the same time, and not separately.

III. TRAFFIC CONTROL METHODOLOGY

Initially, we introduce the design of a modern heterogeneous SoC to support a mixture of critical and non-critical computing entities with the requirement to provide application isolation, in terms of usage of system resources, e.g., memory bandwidth.

A. System Architecture

We target embedded devices that can host applications with different requirements in terms of bandwidth either of system memory or of other on-chip components. We assume a Network-on-Chip viewed as an interconnect intellectual property module with conventional read and write semantics; the network interface offers read/write and block transfers. Our architecture is based on ZYNQ System-on-Chip of Xilinx. We utilize both the Processing System that consists of the dual ARM Cortex-A9 CPU [12], as well as the Programmable Logic (PL) area, wherein we integrate the Traffic Shaper Module (TSM). More precisely, our architecture involves the following components and attributes.

- It utilizes the Processing System (PS) Direct Memory Access (DMA) controller that is used for burst transactions.
- The CPU routes accesses to the memory through the GP port to the PS DDR3 memory. A central interconnect is located within the PS that comprises multiple switches in order to connect to the system resources by using the AXI point-to-point channels for communicating addresses, data, and response transactions between master and slave ports [12]. This ARM AMBA 3.0 interconnect implements a full array of the interconnect communication capabilities and overlays for QoS, debug, and test monitoring. The interconnect manages multiple outstanding transactions and is architected for low-latency paths for the ARM CPUs and for the PL master controllers. The accesses from the PS DMA cross only one GP Port and HP port (HP0), as shown in Fig. 1.
- The device integrates a Microblaze soft-processor [10] that is used to monitor resources that are used by the main processor (A9 CPU) and to collect information from the AXI performance monitor. Even though monitoring and controlling of system resources, i.e. NoC and memory bandwidth must be implemented with customized hardware to offer fine-grain control and response times, a software-oriented approach (using MicroBlaze) offers a coarser-grain solution but more flexible.
- The device uses a custom Address Remap Block, to remap addresses that arrive through the PS and redirects them to the DDR.
- Custom circuitry is integrated at RTL level, which provides the ability of monitoring, control and supplying guaranteed bandwidth for critical applications.

B. Traffic Shaper Module (TSM)

Shaping is a QoS (Quality-of-Service) technique that we can use to enforce pre-specified bitrates, different than what the physical interface is capable of. TSM is a hardware block that implements control of incoming data traffic which actually consists of read and write transactions, following the AXI4 protocol [22]. More precisely, the developed TSM controls the number of accesses based on configured/programmed
bandwidth and regulates the flow of traffic, in order to apply restrictions on the consumed bandwidth. Figure 1 shows the prototype system that integrates the TSM. The proposed architecture contains the registers that are listed in Table I. The designer’s options are, a) to attach it on a Master Interface and control it via the main CPU, b) to connect the TSM control port and establish the overall management of the TSM to another independent core, which thus offloads the burden to dynamically monitor and evaluate QoS per connection or per process; the main CPU (ARM Cortex-A9) communicates with that core only to provide configuration parameters. In this work we chose the second method.

TSM utilizes a separate AXI-lite interface for configuration. The TSM main components are shown in Figure 2. The TSM configuration is done via the MicroBlaze. The programming of registers determines the maximum data transfer bytes to transfer in a time window. The AxLen signals, AxSize and AxValid are responsible for indicating a new transaction and whether this is a single word or a burst. The DMA inside the PS supports AXI burst lengths of 1 to 16 transfers, for all burst types [3]. However, the AXI4 protocol extends the burst length support for the INCR burst type to support 1 to 256 transfers. The burst length for AXI3 is defined as, \( \text{Burst Length} = \text{AxLEN}[3:0] + 1 \), while the burst length for AXI4 is defined as, \( \text{Burst Length} = \text{AxLEN}[7:0] + 1 \), to accommodate the extended burst length of the INCR burst type in AXI4.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxbw</td>
<td>Stores the maximum transfer bytes limit, maximum bytes to transfer in ( \text{maxcc} ) time interval</td>
</tr>
<tr>
<td>acc</td>
<td>Accumulating counter adding the new transaction bytes, should be less than ( \text{maxbw} ) limit</td>
</tr>
<tr>
<td>totalacc</td>
<td>Sums all the new accs and keep the values after the reset</td>
</tr>
<tr>
<td>burst</td>
<td>Stores each new AXI transaction, burst or single word</td>
</tr>
<tr>
<td>cccnt</td>
<td>Clock cycle counter; in each clock cycle ( \text{cccnt} ) increases by 1 measuring total clock cycles</td>
</tr>
</tbody>
</table>

**IV. Evaluation**

**A. Methodology**

Several different ways and methods exist, so as to monitor the activity of a processor or a module. As we presented previously this can be done by designing a controller that supports mixed-criticality service. Software techniques also exist, which support control at task level or at OS level [19].

Our methodology supports both control and monitoring on the Network-on-Chip (NoC) Interface, without requiring the re-design or tampering with the NoC routers or the memory controller [20]. The Traffic Shaper Module provides guaranteed bandwidth to the critical applications by limiting consumed bandwidth of the non-critical tasks.

We used a tightly system-coupled configuration to run our applications. Tightly system-coupled software means routines (modules or full applications) that work on only one type of system since these are dependent on each other and on the system configuration. For example, the driver of an embedded cyber-physical system device requires extensive programming changes to work in another environment, but is usually optimized to offer predictability, minimal latency and even fault-tolerance. We ran two stand-alone (baremetal) applications in A9 CPU and MicroBlaze. The MicroBlaze is responsible to monitor and control the configuration of the TSM registers. The A9 CPU can be configured to be in control of this block (start, restart, counters initialization and monitoring), but the main goal is to dedicate the processor only to the application running on it. In our case the accesses cross only one GP port (GP0) towards the slave PS interconnect (as shown in Fig. 1). If the DMA engine is utilized to perform bulk data transfers, then the DMA sequence of operations is as follows:

- DMA Configuration
- Setup DMA Interrupt
- DMA Initialization
• DMA Start Transfer
• DMA Check handler done
• DMA Stop and reset

If, we want to exploit the maximum theoretical bandwidth that manufacturers specify [12], then the NoC specificities and the memory controller properties should be carefully exploited. For instance, in the ZYNQ architecture any transaction must use specific ports and route along the interconnection scheme to avoid conflicts. The maximum supported bus clock is 533 MHz in the DDR3 module for all speed grades, reaching a theoretical maximum bus bandwidth of 1333 Mb/s. All DMA transactions use AXI interfaces to move data between the on-chip memory, the DDR memory and the slave peripherals in the PL.

Initially, in evaluating the accuracy of bandwidth monitoring we used a software only approach. We used a baremetal application executing on the MicroBlaze which, with the aid of a hardware AXI performance monitor, captures the activity of the AXI interconnect. The A9 CPU drives the PS DMA to send traffic over this interconnect. As Figures 3 and 4 show, when fine-grain resolution is adopted the software monitoring delivers poor results, while when the hardware TSM is activated the mean error is negligible.

There are many options and routes to transfer data in an advanced SoC such as ZYNQ. Two typical DMA transaction examples include: 1) memory to memory (On-chip memory to DDR memory) 2) memory to/from a PL peripheral (DDR memory to PL peripheral). The incoming traffic arrives only from one GP Port crossing the AXI HP0 port and the S2 OCM port [12].

We developed two applications to move data using the first configuration. In one scenario, one alleged critical application runs on the A9 CPU in a baremetal fashion and the MicroBlaze enables the TSM and generates traffic. As Figure 5 shows, in the first scenario the MicroBlaze writes to the memory by using the DMA engine, thus reaching 23.5MB/sec. We also notice that the traffic from the A9 CPU towards the DDR is not affected, and this is because the TSM provides the maximum throughput to the critical application. In the second scenario in Figure 6 we clearly observe that the process on the A9 CPU that makes the Reads and Writes is influenced. The traffic generator running in the MicroBlaze is activated at specific times and significantly affects the bandwidth received by the A9 CPU.

As long as the Traffic Shaper Module is in a ‘disable’ mode, the DMA can reach up to a limit depending on AxLen (Burst size and Length). Moreover, as we have observed the GP Port performs poorly in throughput, as it also appears in Figure 5. When the TSM is enabled the shaper can be configured to regulate the traffic until the consumed bandwidth is less than or equal to the maximum bandwidth that can be reached when the shaper is disabled.

Fig. 3. Comparison of accuracy when scaling the time window using varying data unit size

If, we want to exploit the maximum theoretical bandwidth that manufacturers specify [12], then the NoC specificities and the memory controller properties should be carefully exploited. For instance, in the ZYNQ architecture any transaction must use specific ports and route along the interconnection scheme to avoid conflicts. The maximum supported bus clock is 533 MHz in the DDR3 module for all speed grades, reaching a theoretical maximum bus bandwidth of 1333 Mb/s. All DMA transactions use AXI interfaces to move data between the on-chip memory, the DDR memory and the slave peripherals in the PL.

Initially, in evaluating the accuracy of bandwidth monitoring we used a software only approach. We used a baremetal application executing on the MicroBlaze which, with the aid of a hardware AXI performance monitor, captures the activity of the AXI interconnect. The A9 CPU drives the PS DMA to send traffic over this interconnect. As Figures 3 and 4 show, when fine-grain resolution is adopted the software monitoring delivers poor results, while when the hardware TSM is activated the mean error is negligible.

There are many options and routes to transfer data in an advanced SoC such as ZYNQ. Two typical DMA transaction examples include: 1) memory to memory (On-chip memory to DDR memory) 2) memory to/from a PL peripheral (DDR memory to PL peripheral). The incoming traffic arrives only from one GP Port crossing the AXI HP0 port and the S2 OCM port [12].

We developed two applications to move data using the first configuration. In one scenario, one alleged critical application runs on the A9 CPU in a baremetal fashion and the MicroBlaze enables the TSM and generates traffic. As Figure 5 shows, in the first scenario the MicroBlaze writes to the memory by using the DMA engine, thus reaching 23.5MB/sec. We also notice that the traffic from the A9 CPU towards the DDR is not affected, and this is because the TSM provides the maximum throughput to the critical application. In the second scenario in Figure 6 we clearly observe that the process on the A9 CPU that makes the Reads and Writes is influenced. The traffic generator running in the MicroBlaze is activated at specific times and significantly affects the bandwidth received by the A9 CPU.

As long as the Traffic Shaper Module is in a ‘disable’ mode, the DMA can reach up to a limit depending on AxLen (Burst size and Length). Moreover, as we have observed the GP Port performs poorly in throughput, as it also appears in Figure 5. When the TSM is enabled the shaper can be configured to regulate the traffic until the consumed bandwidth is less than or equal to the maximum bandwidth that can be reached when the shaper is disabled.

Fig. 4. For four different time intervals (1500, 1000, 500, 250 cycles) we measure the error rate that raises between the theoretical and the measured traffic bandwidth without activation of the TSM. For each interval we count three different data size transfers (20, 24, 36 bytes). For example, as we observe in the graph, for 1500 cycles the average difference arises to 17.48% of bandwidth that was measured compared to the theoretical values that were calculated. By enabling the TSM the error rate in the worst case reaches 6.07%.

As long as the Traffic Shaper Module is in a ‘disable’ mode, the DMA can reach up to a limit depending on AxLen (Burst size and Length). Moreover, as we have observed the GP Port performs poorly in throughput, as it also appears in Figure 5. When the TSM is enabled the shaper can be configured to regulate the traffic until the consumed bandwidth is less than or equal to the maximum bandwidth that can be reached when the shaper is disabled.

Fig. 5. Nearly perfectly unaffected bandwidth served in control of the TSM

B. Use Case for Healthcare Application Example

Healthcare devices that integrate Bluetooth version 4.0 [21] may be found in three different locations on the patient’s body, which transmit to a gateway device similar to the ZYNQ-based one that we presented previously. In this particular scenario we consider such a device to run a critical application that can process streaming data that reach 25Mbit/sec or 3 MB/sec.
Fig. 6. System impact when the TSM is deactivated; random DMAs cause irregular and uncontrolled memory bandwidth consumption

each [21]. Consequently, we could reach up to N devices*3/sec theoretical aggregate throughput. The gateway device utilizes only a single Bluetooth interface to communicate with each external healthcare device.

In our use-case scenario that is depicted in Figure 7, the incoming data traffic can reach up to the maximum of 3MB/sec and the gateway forwards this traffic with at least equal speed. Consequently the maximum aggregate bandwidth is 6MB/sec, i.e 3MB/sec incoming traffic and 3 MB/sec outgoing traffic.

In addition, best-effort applications can be executing in parallel that also demand additional memory throughput. As we have analyzed in the previous section, when the Cortex A9 CPU communicates through the AXI_GP0/1 ports to the system memory we can achieve 10.797 MB/sec maximum throughput. The Bluetooth communicates via the ZYNQ gateway in order to feed the information to the user via a local Ethernet or WIFI connection.

When enabling the Traffic Shaper Module, we can throttle the throughput of the best effort applications so as to provide guaranteed BW for the critical function of conveying the devices data, which may contain important messages for the patients that must receive guaranteed delivery to the user. The TSM can be configured so that it can support the desired service level even to the maximum throughput (9 MB/sec).

C. Simulation of use case scenario using a random traffic generator

The above scenario was implemented by creating a Random Traffic Generator (RTG), as shown in Figure 8, which can generate up to 3MB/sec read/write traffic in total. The RTG runs on the Microblaze whereas a non-critical application with its own demands in throughput runs in parallel on the A9 CPU with none impact to its performance.

In addition, notice that even in the case where the Microblaze has the need to consume the maximum bandwidth (3MB/Sec), the TSM ensures that the application’s demands in terms of throughput, which runs in parallel on the A9 CPU, is not affected by the needed bandwidth that is consumed by the Microblaze.

V. CONCLUSIONS

A multicore embedded system is demonstrated in this paper, where the bandwidth is controlled by establishing a hardware Traffic Shaper Module. In such systems, in order to ensure system predictability and applications’ properties such as criticality, we have to control the usage of systems resources. We achieve this by using a Traffic Shaper Module; its goal is to accurately control the number of accesses to provide the desired configured bandwidth. We presented a methodology for bandwidth management suitable for NoC interface and memory controller with the usage of custom circuitry and a minimal software monitoring application. The system was prototyped using the Xilinx ZYNQ7000 System-on-Chip and the measurements extracted from a zedboard development board. By enabling the Traffic Shaper in our architecture we achieved very fine-grain control with negligible overhead, while providing bandwidth of only 0.5-5 percent less than the theoretical bandwidth specified.
ACKNOWLEDGMENT

This research has been co-financed through the European FP7 DREAMS project under the Grant Agreement No. 610640.

REFERENCES


Programming Support for Time-sensitive Adaptation in Cyberphysical Systems

Mikhail Afanasov
Politecnico di Milano, Italy
mikhail.afanasov@polimi.it

Aleksandr Iavorskii
John Wiley & Sons Inc., Russia
yavalvas@gmail.com

Luca Mottola
Politecnico di Milano, Italy
luca.mottola@polimi.it

Abstract—Cyberphysical systems (CPS) integrate embedded sensors, actuators, and computing elements for controlling physical processes. Due to the intimate interactions with the surrounding environment, CPS software must continuously adapt to changing conditions. Enacting adaptation decisions is often subject to strict time requirements to ensure control stability, while CPS software must operate within the tight resource constraints that characterize CPS platforms. Developers are typically left without dedicated programming support to cope with these aspects. This results in either to neglect functional or timing issues that may potentially arise or to invest significant efforts to implement hand-crafted solutions. We provide programming constructs that allow developers to simplify the specification of adaptive processing and to rely on well-defined time semantics. Our evaluation shows that using these constructs simplifies implementations while reducing developers’ effort, at the price of a modest memory and processing overhead.

I. INTRODUCTION

Cyberphysical systems (CPS) enable the tight integration of embedded sensors, actuators, and computing elements into feedback loops for controlling physical processes. Example applications include factory automation, automotive systems, and robotics [25]. CPS operate at the fringe between the cyber domain and the real world [11]. Both the execution of the control logic and the platforms it runs on are thus inherently affected by environmental dynamics [25]. This requires CPS software to continuously adapt to these dynamics. To enact the needed adaptations, developers may employ various approaches, including dynamically changing the control logic.

To complicate matters, control loops are most often timesensitive [24]; the control logic must be executed at a given frequency to ensure the stability of processes. Adaptation is an integral part of the control loop, and thus subject to the same time constraints. Thus, the timing aspects of taking and enforcing adaptation decisions become crucial. Such complex time-sensitive adaptive processing must withstand the strict resource constraints of CPS platforms; the most advanced CPS devices feature 32-bit micro-controller units (MCUs) with tens of Kbytes of RAM, while being battery-operated.

As we illustrate in Sec. II, developers are often left without dedicated support to implement adaptive time-sensitive CPS software. This leads to easily overlooking the potential issues related to the timing aspect of run-time adaptation, ultimately affecting the stability of the controlled processes. Fritsch et al. [7] show, for example, that timing aspects are often neglected in developing adaptive automotive software. Similar observations also apply to robot controllers [2], [18]. Whenever developers do recognize these issues, they tend to implement complex hand-crafted solutions, mostly due to the lack of time semantics in mainstream programming abstractions.

To address these issues, we design and implement a custom realization of context-oriented programming [10], [22] that is: i) conceived for resource-constrained embedded devices, and ii) embeds well-specified notions of adaptation modality and time. As described in Sec. III, these notions allow developers to distinguish different ways to schedule an adaptation decision and to place an upper-bound on the time taken to apply such decisions. The former is useful, for example, to avoid functional conflicts when switching from one control logic to another. The latter provides a specified time semantics when adaptation decisions need to abide to real-time deadlines. We render these notions in a dedicated extension of the C++ language we call COP-C++, supported by a corresponding tool-chain we develop.

As reported in Sec. IV, we assess our work by quantitatively comparing the complexity of representative implementations of CPS software using traditional programming constructs against those we design. Our results indicate that the developers’ effort is reduced using COP-C++. The cost to gain this benefit is a modest increase in resource consumption, especially in processing time and memory occupation. For example, the worst-case processing overhead we measure through real-world experiments on modern 32-bit MCUs amounts to only 20µs, negligible given the time scales of the considered control loops. We end the paper by discussing related efforts in Sec. V and with brief remarks in Sec. VI.

II. PROBLEM

Consider the need to localize a gas leak in an indoor environment. Tiny aerial drones are envisioned to perform this task efficiently and with minimal cost [4]. Their behavior, as dictated by a given control logic, depends on surrounding conditions, application state, and sensed data [4].

Fig. 1 depicts a possible design for such an application. Initially, every drone moves to a predefined location using
static void step() {
  // should be called at 100Hz or more
  switch (current_controller) {
    case NAVIGATE: navigate_step(); break;
    case HOVERING: hovering_step(); break;
    case LEAK_LOC: leak_loc_step(); break;
    case LANDING: landing_step(); break;
  }
  current_controller = controller;
}

static bool set_controller(uint8_t controller) {
  bool success = false;
  switch (controller) {
    case NAVIGATE: success = navigate_init(); break;
    case LEAK_LOC: success = leak_loc_init(); break;
    case LANDING: success = landing_init(); break;
    default: success = false; break;
  }
  if (success) {// update controller
    exit_mode(current_controller, controller);
    current_controller = controller;
  } else {// log error
    Log_Write_Error(controller);
  }
  return success;
}

Fig. 2: Example implementation of adaptive controller.

A Navigation controller. Upon arriving, a drone switches to a Hovering controller to sample the gas concentration. Whenever it detects a value above a safety threshold, a drone switches to a LeakLocalization controller that broadcasts alert beacons using a low-range radio. Nearby drones that receive the beacon also switch to the LeakLocalization controller, and come closer to the drone that initially detected the leak to obtain finer-grained measurements. In case of emergencies, such as a hardware failure, the Landing controller is activated.

Fig. 2 depicts an example implementation of the required adaptive behavior. The structure of the code reflects real implementations in the considered systems; for example, in the Ardupilot [2] autopilot software for drones. Control is triggered in function step() in line 3, which is called at 100Hz. Depending on the global variable current_controller, different concrete controllers are executed. Controller adaptation is implemented in function set_controller() in line 8. Depending on what controller is to be activated, an individual controller is first initialized, the clean-up routine of the previous controller is executed on line 8, and the global variable indicating the active controller is updated in line 9.

Despite being simplified, the code in Fig. 2 already shows several issues, some not even entirely evident:

I) The processing is strictly coupled with the different controllers. For example, adding a new controller, or removing an existing one would require changing the code in several places. On the other hand, resource constraints prevent using high-level languages that ameliorate these issues. As a result, implementations are typically tangled and thus difficult to debug and to maintain.

II) In Ardupilot, control runs at 100 Hz; every 10 ms a controller must perform the necessary actuation. However, current implementations enforce no time limit on adaptation. This may become an issue when executing set_controller: should a controller’s initialization or clean-up take too long, the controller will not be executed in time, which may affect the system’s stability.

III) When switching controller, the previous and new controllers may conflict with each other. For example, the LeakLocalization controller includes a periodic task to transmit alert beacons that may still operate when the Landing mode is possibly initialized. This would mean the drone keeps beaconing also when it lands, which may affect the system’s correctness. Asynchronous operations, such as interrupt handlers firing while switching controller, may create similar issues.

As we argued earlier, these problems are often overlooked by CPS developers. Issue I1 impacts the quality of implementations, whereas issue I2 and I3 potentially affect dependability. Addressing these issues, however, is also not trivial without proper programming support. For example, issue I3 often emerges as developers intentionally overlap initialization and clean-up operations to increase parallelism when performing I/O operations. Solving issue I1 by simply switching the ordering of clean-up and initialization decreases parallelism, possibly prolonging the time required for switching controllers and thus exacerbating issue I2. To remedy this, developers implement hand-crafted solutions to regulate the time for switching controllers, further impacting the quality of implementations, making issue I1 even worse.

III. COP-C++

Context-oriented programming (COP) [10] is a paradigm to simplify the implementation of adaptive software. It fosters a strict separation of concerns, achieved through two key notions: i) the different situations where the software needs to operate are mapped to different contexts, and ii) the context-dependent behaviors are encapsulated in layerd functions, that is, functions whose behavior changes—transparently to the caller—depending on the active context.

COP already proved effective in creating adaptive software in mainstream applications, such as user interfaces [13] and text editors [12]. To that end, COP extensions of popular high-level languages, such as Java and Erlang, emerged [12], [22]. Embedding COP extensions within an existing language, however, often relies on features such as reflection and dynamic binary loading, which are difficult to implement on resource-constrained platforms, such as those employed in CPS.

A. Context-constrained C++

To address issue I1 in Sec. II, we embed the key COP abstractions within C++, as it arguably represents a large fraction of the existing codebase in CPS. The resulting language, called COP-C++, retains the original C++ semantics with the addition of custom semantics and keywords. We focus on the local adaptation, while any distributed functionality is orthogonal to our efforts. Indeed, our approach can be used with any middleware for adaptation in distributed systems; for example, i.Land [8].

For simplicity, we illustrate the language through examples here. The full grammar is publicly available together with the corresponding tool-chain.1

Context groups and individual contexts. Similar to previous work [1], we group together contexts sharing common characteristics, such as the functionality provided or the environment dimension that drives the transition between contexts. In the example of Sec. II, individual contexts would map to the individual controllers in Fig. 1. These contexts would be grouped together as they all provide control functionality.

1https://bitbucket.org/neslabpolimi/cop_cpp_translator
context group FlightControlGroup {
  context Hovering; context Navigate; context LeakLoc;
  context Landing;
  public:
  1. layered void step() = 0;
}

Fig. 3: Example definition of context group.

class LeakLoc : private FlightControlGroup {
  public:
  1. LeakLoc(): _ticker(new Ticker()) {};
  2. virtual ~LeakLoc();
  private:
  3. layered void step() { // ... controller functionality}
  4. bool initialize() {_ticker->attach(&broadcast, 0.3);} 
  5. void cleanup() {_ticker->detach();}
  6. void broadcast() { // broadcast routine}
  7. _ticker = new Ticker();
}

Fig. 4: Example implementation of individual context.

Fig. 5: Lazy and fast activation of a new context.

Adaptation. Using COP-C++, enacting an adaptation decision that prompts to easily switch between the controllers. For example, the command activate FlightControlGroup::Hovering fast within 5ms performs the change of the currently-executing controller including initialization and clean-up. The specific scheduling of these operations depends on the qualifiers lazy and fast[within] in the same lines, whose semantics we explain next. The call to function FlightControlGroup::step() is then transparently dispatched to the currently active context.

Compared to plain C/C++, as shown in Fig.2, the code appears much simplified. No global variables are necessary to keep track of the current controller. No cumbersome switch statements are required, either. Only a single call to the step function appears in the code, which is automatically dispatched to the active context. The interleaving of controllers’ initialization and clean-up while switching, as well as the corresponding time semantics, are completely encapsulated in the aforementioned qualifiers, as explained next.

B. Qualifiers

As described in Sec. II, functionality meant to operate in different situations may conflict with each other during the switch—as per issue I2. In addition, potential issues may emerge as current implementations enforce no time limit on the execution of adaptation decisions, as per I3. To help programmers cope with these issues, the qualifiers lazy and fast[within] indicate different modes to enact adaptation decisions and possibly specify time constraints.

Modes. Fig.5 illustrates the difference in performing a context switch using lazy or fast. Using lazy, the clean-up of the previous context needs to complete before initializing the new context. As a result, no functional conflicts may ever arise. However, during the switch, the system rests in an uncertainty state where no context is active. A call to a layered function within this time results in no operation. In addition, as no parallel execution occurs, the latency grows as the sum of the time to clean-up from the previous context and the time to initialize the new one.

In contrast, as shown in Fig.5, using fast the system first initializes the new context; then performs the clean-up of the previous one. If some operations inside initialize are non-blocking and may be asynchronously executed, such as those involving I/O operations, fast allows the system to increase parallelism. As a result, the time to apply an adaptation decision reduces, yet programmers must take additional care to avoid functional conflicts between contexts during the switch. There is indeed a time where both contexts might be possibly simultaneously executing.

Deadlines and rapid context switches. To control the time invested in switching between contexts, programmers may define an optional activation deadline. Say, for example, that in the scenario of Sec. II the context switch needs to complete within T ms to let the new controller perform the actuation within the next (10 − T) ms. The optional qualifier within allows programmers to specify the upper-bound T on the time to switch contexts, as shown in Sec. III-A. Should the upper bound be violated, the initialization is interrupted and the programmer is notified through a callback, which can be used to implement application-specific countermeasures.

All the activate commands are placed in a queue that is asynchronously checked by the system. The latter pulls out the
first **activate** command and executes it. In an emergency situation, such as a collision threat, programmers may want to switch the context immediately. To this end, an instruction **activate FlightControlGroup**: **Landing** immediately cancels all pending **activate** commands and immediately switches to the **Landing** context.

The qualifiers we discuss here also naturally apply across multiple context groups, defined as explained in Sec. III-A, in applications with parallel adaptive controllers.

**C. Translator**

We implement the translator as an extension to the CDT plug-in for Eclipse. It allows programmers to translate from COP-C++ to pure C++ and to rely on standard toolchains to generate executable binaries. First, the translator ensures the consistency of the context-oriented design. For example, a context may not implement any layered functions or not belong to any context group. In this case the translation stops and the developer is informed. Second, the generated source code remains human readable; programmers can modify it to perform further fine-grained optimizations and tuning.

**IV. Evaluation**

We assess the effectiveness of COP-C++ along several dimensions. Sec.IV-A quantitatively demonstrates the benefits of COP-C++ in complexity of the implementations and required programming effort. Such benefits come at a price of processing and memory overhead, which we report in Sec.IV-B. We compare the performance of different combinations of qualifiers for context switch in Sec.IV-C, whereas Sec.IV-D shows the programming effort required to manually cope with functional conflicts when using **fast** switching.

Our evaluation targets modern 32-bit ARM Cortex M MCUs that are often employed in CPS. We employ STM Nucleo prototyping boards equipped with Cortex M3 MCUs running at 32 MHz and 80 KBytes of RAM. The architecture of these is similar, if not the same, to devices employed in real-world CPS applications, while the board also offers convenient testing facilities that enable the kind of fine-grained measurements we discuss next.

As input to our evaluation, we implement the gas leak localization application described in Sec.II using COP-C++. We use two functionally-equivalent implementations as baselines. The first one uses pure C++ by following the structure of an original Ardupilot-like implementation, discussed in Sec.II. We call this implementation **PUREC++**. The second baseline is similar to **PUREC++**, with the addition of manually-implemented functionality to control the time for switching controllers, that is, the same functionality the **within** qualifier provides declaratively. Such a baseline is instrumental to examine the difference between the manual implementation and the automatic generation of this functionality. We call this implementation **TIMEC++**. We implement all three versions of the application using the mBed [15] libraries provided by STM. We use the standard ARM **gcc** tool-chain for compiling.

**A. Complexity and Effort**

We compare the complexity and efforts for the three implementations using Halstead metrics [9]. These are intended to investigate the properties of source code independently of the programming language. Halstead et al. use four basic metrics: the total number of operands (OP), the total number of operators (OD), the number of unique operands (UOP), and the number of unique operators (UOD). An operator is a language-specific keyword, whereas variables and constants are operands. For example, in Fig.4, **layered** is an operator, whereas _ticker is an operand. Based on the four basic metrics, other metrics are derived as shown in Fig.6.

**Results.** Fig.7 reports the values of the Halstead metrics for the three aforementioned implementations. The Difficulty in a COP-C++ implementation is slightly higher than for the baselines. This is due to the additional keywords we add to define context groups and individual contexts, as well as the use of qualifiers.

On the other hand, COP-C++ reduces the Volume of the program; therefore, maintenance and debugging should be facilitated as programmers need to absorb less information to understand a program. Programmers also spend less Effort to realize the program in the first place. The benefits of COP-C++ in these regards become even more apparent when considering **TIMEC++**. In this case, both the Volume and Effort further increase, amplifying the benefits of COP-C++.

**B. Processing and Memory Overhead**

The benefits above incur a run-time cost in processing time and memory occupation. To assess these, we separately compare a COP-C++ implementation that only uses the **fast** qualifier against **PUREC++**, and a COP-C++ implementation that also employs the **within** qualifier against **TIMEC++**. The original Ardupilot implementation only uses a kind of controller switch similar to the semantics of the **fast** qualifier, so we do not study here the run-time overhead for the **lazy** one. We investigate this in Sec.IV-C.

According to Fig.1, different environmental events may trigger the adaptation. We emulate these events on the Nucleo board as external interrupts through GPIO pins. We use a Tektronix TBS 1072B-EDU oscilloscope attached to the board.
to measure the time to perform the controller switch. Memory usage statistics are obtained from the mBed [15] on-line IDE.

Results. Fig. 8 shows the processing time to switch controller depending on the external event. Adaptation in COP-C++ takes slightly more time—approximately $11\mu$Sec—compared to PUREC++. The absolute values vary due to different initialization routines; for example, periodic beaconing is only initialized when LeakLocalization is activated by alert beacon or high gas concentration. With the within qualifier, the processing overhead compared to TIMEC++ reaches $20\mu$Sec. Such a penalty, however, is still almost unnoticeable, as the typical control loop runs at hundreds of Hz.

COP-C++ shows a mere 200B RAM overhead, which is negligible compared to both baselines that consume 2.1kB of RAM. RAM consumption is often an issue when developing CPS software; therefore, minimizing the impact on this figure is key. On the other hand, the program memory usage using COP-C++ appears 14.6kB higher than in the baselines that use 20kB. This is mainly due to the simplified implementation of the control loops we employ for this study, where only basic functionality are included and platform-specific libraries are replaced with empty stubs. Functionally-complete implementations are much larger; for example, the full ArduPilot [2] requires 792 KB of program memory. A major fraction of these are not processed by our translator; therefore, we expect the relative overhead in program memory to amortize.

C. Qualifiers

As the example application in Sec. II only uses the fast qualifier, we quantitatively study here the trade-offs between the adaptation qualifiers in COP-C++, described in Sec. III-B. The memory overhead is the same independent of the specific combination of qualifiers that appear in the code, and corresponds to the values shown in Sec. IV-B. Therefore, here we focus on the latency to perform the context switch depending on the combination of qualifiers. The experimental setup is the same as in Sec. IV-B.

Results. Our investigations show that the lazy qualifier requires $33, 6\mu$Sec to complete the context switch. This latency is the price programmers pay to ensure that no functional conflicts arise during the switch. To reduce this time, programmers can use the fast activation type that requires only $21, 4\mu$Sec without optional qualifier within. This choice, however, requires programmers to handle potential functional conflicts by hand, increasing the programming effort. We investigate this aspect in Sec. IV-D.

Using the optional within or immediately qualifier only marginally increases the latency in switching context. As for the former, the latency grows by $10\mu$Sec because of the additional processing required to initialize a dedicated timer that fires if the switch takes too much time. In the latter case, the additional processing time amounts to only $\approx 1\mu$Sec, required to purge the queue of pending context switches. In both cases, the absolute values are very limited, and they should not impact the timings of control loops that typically run with periods that are orders of magnitude larger.

D. Development Trade-offs

Using the fast qualifier may result in functional conflicts because the initialization and clean-up routines of different contexts overlap in time. Programmers need to handle these conflicts by hand. To assess the additional programming effort required, we implement a new version of the application in Sec. II with the addition of simple wrappers around any of the classes where asynchronous events may fire during a fast context switch. These include timers and classes that signal hardware interrupts. The wrappers intercept any such asynchronous event and forward it further only after checking that the active context corresponds to the one the event is addressed to. Cleaner, yet more complex solutions are also possible. Considering simple wrappers provides a lower-bound in terms of the additional programming effort.

We assess the added programming effort by re-calculating the Halstead metrics of Sec. IV-A on the new implementation. Further, the wrappers obviously add latency to the context switch. We measure this with the same setup of Sec. IV-B.

Result. As shown in Fig. 9, the wrappers add a mere $\approx 2\mu$Sec in latency during a context switch. Thus, their performance impact is negligible. However, the complexity of the implementation increases considerably, as reported in Fig. 10. Programmers need to invest significant efforts not only in implementing the wrappers, but also to nail down all the classes that could possibly lead to functional conflicts, and provide a wrapper for each of these. Fig. 10 reports a considerable increase in all the complexity metrics when wrappers are used. For example, the Volume of the source code increases by 80%, the source code is 53% more Difficult, and requires almost 3 times the Effort to be written.

V. RELATED WORK

Time-sensitive software adaptation in CPS is a multi-facted problem. Albeit comprehensive programming support largely
lacks, works exist that tackle individual aspects.

**Parameter and component configurations.** Adjusting the software operating parameters is one way to adapt. For example, Garcia-Valls et al. [8] focus on the reconfiguration of the nodes in distributed soft real-time system that meets stated performance goals. In the area of adaptive controllers, Mokhtari et al. [17] and Frew et al. [6] tune the operation of unmanned aerial vehicles (UAVs) based on sensor inputs. These approaches focus on adapting a specific fraction of the system’s functionality, for example, motors’ parameters or nodes’ configuration, and cannot be applied where the whole control logic must be changed on a single node. Our work does not focus on the mechanism to adapt a specific functionality, but provides generic programming support for implementing adaptive CPS software under time constraints.

In component-based systems, software reconfiguration often occurs by plugging components in/out or by changing component wirings [21]. Dedicated component models exist that allow developers to verify—using formal techniques such as model-checking—the correctness of new component configurations [20]. Some of these works focus on specific application domains such as automotive [26] and autonomous underwater vehicles [16]. Unlike our work, these approaches offer no programming support to deal with enacting time-sensitive adaptation decisions.

**Programming support for adaptation.** Software adaptation for traditional platforms is extensively studied. Some of the works explicitly focus on programming support. For example, COP [10] itself was implemented in a number of high-level languages [3, 12, 22, 23]. The techniques normally used to embed COP abstractions into a host language tend to be impractical in CPS because of resource constraints. Similar observations apply to Meta- and Aspect-oriented programming (AOP) [22]. The corresponding abstractions often require self-modification of the deployed binaries [14], which is hard to implement on resource-constrained platforms. Our work renders COP concepts amenable for implementation on typical CPS devices, while adding semantics useful when enacting time-constrained adaptation decisions.

The need to provision programming support for time-sensitive adaptation was also recognized in the area or real-time operating systems (RTOSes). Dedicated programming abstractions based on reflection were added to existing RTOSes [19] or specific services were made available that perform the needed reconfiguration in a safe manner [5]. These attempts utilize language- or operating system-specific features that are often not available in typical CPS platforms, because of resource-constraints. Target platforms of these approaches are either the traditional computing machines [8], [21] or FPGAs [5], which greatly surpass CPS platforms in terms of available resources and energy consumption. Our solution, instead, is designed for resource-constrained devices, does not require any language-specific features such as reflection, and remains decoupled from the the underlying operating system.

**VI. Conclusion**

We presented COP-C++, an extension to C++ we expressly conceived to simplify the implementation of time-sensitive CPS software. To that end, we borrowed concepts from COP and realized them in a way that is feasible on resource-constrained devices, while adding semantics to govern the time aspects during the adaptation process. We implemented a dedicated translator from COP-C++ to pure C++. Our quantitative evaluation showed that COP-C++ simplifies implementations of paradigmatic CPS functionality while reducing programmers’ effort, at the price of a modest run-time processing and memory overhead. For example, processing overhead in our experiments is limited to tens of µSec, while RAM overhead is negligible. Program memory overhead, on the other hand, should be amortized with the increasing size of implementations.

**References**

FLOPSYNC-QACS: Quantization-Aware Clock Synchronization for Wireless Sensor Networks

Federico Terraneo*, Alessandro Vittorio Papadopoulos†, Alberto Leva*, Maria Prandini*

*Politecnico di Milano, Milano, Italy
†Mälardalen University, Västerås, Sweden

Abstract—The development of distributed real-time systems often relies on clock synchronization. However, achieving precise synchronization in the field of Wireless Sensor Networks (WSNs) is hampered by competing design challenges, such as energy consumption and cost constraints, e.g., in Internet of Things applications. For these reasons many WSN hardware platforms rely on a low frequency clock crystal to provide the local timebase. Although this solution is inexpensive and allows for a remarkably low energy consumption, it limits the resolution at which time can be measured. The FLOPSYNC synchronization scheme provides low-energy synchronization that takes into account the quartz crystal imperfections. The main limitation of the approach are the effects of quantization.

In this paper we propose a clock synchronization scheme that explicitly takes into account quantization effects caused by low frequency clock crystal, thus addressing the clock synchronization issue in cost-sensitive WSN node platforms. The solution adopts switched control for minimizing the effect of quantization, with minimal overhead. We provide experimental evidence that the approach manages to reach a synchronization error of at most 1 clock tick in a real WSN.

I. INTRODUCTION

During the past decade, the Internet of Things (IoT) has gained significant attention both in academia and industry. The basic idea of IoT is to connect objects to the internet and make them communicate with each other. According to a recent study by Gartner [4], the current count of IoT devices is around 6.4 billion devices (not including smartphones, tablets, and computers), and it is expected to grow up to 21 billion by 2020. In such a scenario, wireless communication technologies will play a major role, and in particular, Wireless Sensor Networks (WSNs) will proliferate in many application domains. Indeed, the small, inexpensive and low power WSN nodes can be an enabling technology for IoT [8, 13].

Among the various challenges, time synchronization is one of the most important for the correct operation of WSNs [23, 27]. In particular, it allows for successful communication between the different nodes, but also for location and proximity estimation [14], energy efficiency [11, 21], and mobility [22]. In general, WSN nodes coordination is crucial, especially when real-time operations must be executed. Since many IoT devices are battery powered, and time synchronization is continuously executed, energy efficient architectures and protocols are necessary [1].

Even more, since the cost of the nodes is relevant and a high-precision time synchronization may not be needed in some IoT applications, the use of low resolution clock oscillators is often a viable choice, see e.g. [17, 18]. Apparently, the clock resolution affects the minimum achievable synchronization error, and protocols able to push the performance to the limits are needed. In this paper, we propose a synchronization mechanism based on a switched control policy aimed to minimize the effect of quantization on the synchronization error, with a minimal overhead. The following work is cast in the framework of multi-hop master-slave clock synchronization, i.e., when the network is composed by a master, and of a number of slaves which synchronize their clock to the masters’ clock.

The rest of the paper is organized as follows. Section II presents a technological view on the clock synchronization problem. Section III describes the proposed switched control scheme, while Section IV presents simulation and experimental results. Section V concludes the paper.

II. THE SYNCHRONIZATION PROBLEM

A master-slave clock synchronization scheme works by disseminating the timing information using packets transmitted by the radio transceiver of the WSN nodes. In simple topologies, such as star networks, synchronization packets can be transmitted by the master node. In multi-hop networks, flooding schemes [3, 15] allow for the dissemination of synchronization packets also to nodes that are not directly in the radio range of the master node. Packets may contain a timestamp with the time of the master node, or this information can be implicit if packets are transmitted periodically over a contention-delay-free MAC [24]. A clock skew compensation [10, 19, 28] scheme is often used to minimize the divergence of the nodes’ local clock in between synchronizations. Drift compensation is also possible, accounting for common error causes such as temperature variations [20, 24]. Finally, propagation delay estimation and compensation [12, 25] can be employed for ultra high precision synchronization.

This work was partially supported by the European Commission under the project UniCoVeCPS with grant number 643921, and by the Swedish Foundation for Strategic Research under the project “Future factories in the cloud (FiC)” with grant number GMT14-0052.
Despite the many different features a master-slave clock synchronization scheme can be composed of, all of them have in common the need for timestamping incoming packets using the local clock. Timekeeping in WSN nodes is performed by dedicating a hardware timer/counter unit of the node microcontroller that is read to know the time, and allows to set interrupts for generating events. One way to perform incoming packet timestamping is reading the hardware counter in the packet reception interrupt handler [15]. Alternatively, one may use a hardware input capture module [3, 24], a common feature of modern timer peripherals that takes a snapshot of the counter upon receiving an event. In all cases, the finite frequency of the local hardware counter introduces a quantization effect in packet timestamping, of increasing magnitude as the counter frequency is decreased.

Energy consumption constraints limit the frequency at which a WSN node hardware counter can be operated. It is well known that the power consumption is proportional to the operating frequency, load capacitance and the square of the operating voltage. The dependency of power, and thus energy consumption on the clock frequency is a relevant concern in the hardware counter, that has to remain active also in sleep states to not lose the notion of time. For this reason, it is common in WSN nodes to have the microcontroller CPU clocked at several megahertz, while the timer used for timekeeping at only 32768Hz [18]. This choice is based on the availability of inexpensive and ultra low power clock crystals for this frequency.

A breakthrough solution is the Virtual High-resolution Time (VHT) algorithm [21]. It resynchronizes a high frequency timer, which is turned off in deep sleep, while a low frequency timer is always active. Although this solution has been used to achieve high synchronization precision and ultra-low energy consumption [24], its implementation requires hardware support that is uncommon in most WSN nodes. The main requirements for implementing VHT are two: a high frequency timer clocked with a stable (e.g. clock crystal) oscillator, and hardware support for timestamping an edge of the low frequency clock with the high frequency timer. In [21] this problem was solved through hardware modifications to an existing node and also proposed a hardware implementation in VHDL. Although specialized nodes appeared having this support [26], widely deployed nodes [6], such as the TelosB [18], are not capable enough to implement VHT. This is not only a legacy issue, as cost sensitive IoT applications may not have such strict timing requirements to justify the additional cost of VHT support. Addressing the clock synchronization problem using low frequency clock timers is therefore a relevant research topic that has the potential to enable low cost yet real-time capable WSN node platforms.

To further evidence how clock synchronization with a high and low frequency clock are two completely different scenarios, we made a test synchronizing two nodes using the FLOPSYNC-2 [24] scheme on the WandStem [26] WSN node platform. This platform has support for the VHT, so it was possible to compare the synchronization quality in both cases, on the same hardware. The tests were performed using a 10s synchronization period, and the reported error samples are taken at the end of each synchronization period, immediately before the next synchronization. The top graph in Figure 1 shows the results with a high frequency timer, while the bottom graph shows the low frequency timer case. Although the high frequency timer has a resolution of 20.8ns (indicated with the gray area), the standard deviation of the clock synchronization error in the reported test is 164ns (dashed lines in the figures), a significantly higher value. The reason for this is that at high frequencies, jitter caused by the oscillator phase noise, and the packet transmission jitter at the radio transceiver are greater than the quantization-induced error [24]. On the contrary, when the low frequency timer is used, the standard deviation of the error is 24.6µs (indicated with the dashed lines), which is lower than the 30.5µs timer resolution (indicated with the gray area). More significantly, the error shows a regular pattern composed of only three values: 0 and ±1 timer tick, highlighting that the quantization induced error is greater in magnitude than the noise sources.

The clock synchronization algorithm proposed in this paper includes a switching control scheme that can minimize the effect of quantization on the clock synchronization error. The proposed solution is applicable in all cases where quantization is the major source of synchronization error.

III. FLOPSYNC-QACS

After detailing the sources of quantization in clock synchronization, we here formalize the problem and introduce FLOPSYNC-QACS, our proposed switched control scheme that minimizes the effect of quantization.
A. Sources of quantization

When we analyze the quantization problem from an hardware perspective, it is clear that the hardware counter is incremented on the active edge of the clock signal, and asynchronous events – such as the packet reception one – can occur at any time between two active edges. In this case the reported timestamp will naturally be the value of the counter incremented by the previous edge. Thus, the hardware timestamping operation works like the mathematical floor operation. A second quantization occurs when the output of the clock correction algorithm, which is computed using floating point or fixed point numbers, has to be converted back to the tick resolution. In this case, since the conversion is done by a software routine, it is possible to choose the quantization function, for example using the mathematical rounding operator.

From a mathematical perspective, a quantizer maps a real-valued function into a piecewise constant function taking values in a discrete set. Here quantizers are either rounding operators (for the corrective actions) or floor operators (for values in a discrete set. Here quantizers are either rounding operators (for the corrective actions) or floor operators (for the measured synchronization error). Given a real number \( z \), we denote with \( \text{sign}(z) \) the sign function, with \( |z| \) the floor operator, and with \( \rho(z) \) the rounding operator, with \( \rho(0.5) = 1 \), and \( \rho(-0.5) = -1 \). We also define the rounding error of a real number \( z \) as \( \Delta_z := z - \rho(z) \). Notice that the rounding error of a real number \( z \) is always bounded as \( |\Delta_z| \leq \frac{1}{2} \). Finally, note that given two real numbers \( a \in \mathbb{R} \) and \( b \in \mathbb{R} \), we have that \( \rho(a + b) = \rho(a) + \rho(b) \).

B. Problem formalization

The problem of time synchronisation in a distributed system is a well known and studied problem in computer science [2, 5, 9, 16], and has recently gained a lot of attention in control theory as well [7, 11]. We here consider the setting of a WSN where the master node sends a synchronization signal with a fixed period \( T \) to all the slave nodes in the network. Then, synchronization is achieved by controllers of each slave node that communicates with the master and receives the synchronization packets every \( T \) seconds. If a flooding scheme like glossy [3] is used for the transmission, it can be assumed that the medium access contention does not introduce uncertainty in the transmission time.

The synchronization error at time \( kT \), \( k \in \mathbb{N} \), can be defined as:

\[
e(k) := t(k) - \hat{t}(k),
\]

where \( t(k) \) denotes the master node clock at the \( k \)-th synchronization, and \( \hat{t}(k) \) is the slave estimate of the master node’s clock. Since the error accumulates over time, during each time interval \( [kT, (k + 1)T] \) the synchronization error dynamics can be described as:

\[
e(k + 1) = e(k) + d(k),
\]

where \( d(k) \) is a disturbance that accounts for different factors influencing the synchronization error. Notice that the time scale of the considered dynamics is defined by the synchronization period \( T \). In general, the disturbance over a synchronization period can be characterized as:

\[
d(k) = - \int_{kT}^{(k+1)T} \frac{\delta_f(t)}{f_o} \, dt,
\]

where \( f_o \) is the nominal frequency of the slave clock oscillator, and \( \delta_f(t) \) the (continuous-time) variation of that frequency caused by manufacturing tolerances, aging, thermal stress, and short-term jitter. The minus sign in (2) is because a positive \( \delta_f \) makes the local clock advance, while for convenience (1) contains \( d(k) \) with the plus sign.

Notice that all the uncertainty is confined in the disturbance term \( d(k) \), and based on model (1), a controller can be designed to reject \( d(k) \). Whence \( d(k) \) is characterized, a simple control policy can be implemented with a very little computational overhead, like, e.g., [10], in contrast to other classical alternatives. The various sources of disturbances can be counteracted by considering the different time scale of their contributions:

- Tolerances due to imperfections in the manufacturing process of the quartz crystals result in a constant frequency error \( \delta_f \).
- Aging acts on a time scale of days while reasonable values for the synchronization period \( T \) are seconds or minutes, hence it can be thought of as a constant disturbance contribution, and eliminated at steady state by integral control.
- The temperature dependence of clock crystal oscillators is one of the most common source of variable disturbance [21]. In a wide variety of operating conditions a WSN undergoes either abrupt but sporadic thermal stress episodes, or environmental variations that are slow compared to the thermal dynamics of typical nodes. The proposed controller can be extended to compensate for abrupt thermal variation episodes [24].
- In between such events, however, this disturbance contribution can be considered constant as well.
- Short-term jitter acts on the time scale of electronic noise, hence being too fast to compensate, and therefore providing the ultimate bound for the achievable synchronization quality. However, here we are addressing the case where quantization resolution is a greater source of error than jitter, as shown in Section II.

The above characterization of the disturbance contributions allows us to focus on optimizing the controller for the case when \( d \) is constant, although the proposed controller will obviously still be able to cope with variable disturbances.
C. The FLOPSYNC control scheme

In [10], FLOPSYNC was proposed as a viable solution for clock synchronization in WSNs. FLOPSYNC introduces as control variable a corrective action $u$ on the synchronization error, which is then quantized. Hence, (1) becomes:

$$e(k + 1) = e(k) + \rho(u(k)) + d(k).$$

(3)

In the FLOPSYNC scheme, $u$ is determined from the quantized measurements of the synchronization error $[e]$, by a discrete-time Proportional Integral (PI) controller of the form:

$$u(k + 1) = u(k) + [e(k)] - \alpha |e(k + 1)|$$

(4)

where $\alpha$ is the only design parameter. Figure 2 shows the resulting control scheme, where $P$ is the dynamic process (3), and $\hat{R}$ is the controller (4). Plugging (4) into (3), we get that:

$$e(k + 1) = e(k) + d(k) + \rho(u(k - 1) + [e(k - 1)] - \alpha |e(k)|)$$

In [10] the FLOPSYNC control scheme was designed by ignoring the quantizers. More specifically, when no quantization were in place, and inverting (3), leads to:

$$e(k + 1) = (1 - \alpha)e(k) + e(k - 1) + u(k - 1) + d(k)$$

which corresponds to an asymptotically stable linear system if $1 < \alpha < 3$. For a constant disturbance, i.e., $d(k) = d(k-1) = \bar{d}$, the scheme guarantees the convergence of the synchronization error to zero, with a rate of convergence that depends on the parameter $\alpha$.

When quantizers are in place, on the other hand, the synchronization error still converges towards zero, but, intuitively, it is not possible to distinguish errors below the clock resolution. Moreover, the disturbance $d$ is integrated over time according to (3). The integrated residual disturbance is not detectable on the quantized output $[e]$ as long as it is smaller than the clock resolution due. This causes the controller to react whenever the quantization of the integrated residual disturbance switches to 1 or $-1$, steering the controlled system to a limit cycle of amplitude 2. An example of this effect is shown in Figure 3, with $\alpha = 1.5$, $d(k) = \bar{d} = \sqrt{2}$, and the control system initialized as $e(0) = 2, u(0) = 0$.

This paper proposes a switched control scheme that reduces the effect of quantization, steering the system to a limit cycle of an amplitude that is half of the one obtained with the FLOPSYNC control scheme in [10]. The proposed solution has the additional advantage of sticking to simple controllers that lead to an easily implemented system in an embedded device, with very low overhead.

D. The proposed switched control scheme

In this section, we propose a switched variant of the classical FLOPSYNC controller which improves its performance in the presence of quantization effects, and it is called FLOPSYNC-QACS.

The controller is composed by a linear part:

$$\hat{u}(k + 1) = [e(k)] - \alpha |e(k + 1)|$$

(5)

and of a switched part where the control action $\hat{u}$ is set as the input to the following modified discrete-time integrator:

$$\begin{cases}
  u(k + 1) = u(k) + \hat{u}(k + 1), & \text{if } |e(k + 1)| \neq 0 \\
  u(k + 1) = \rho(u(k)) + \hat{u}(k + 1), & \text{if } |e(k + 1)| = 0
\end{cases}$$

that finally computes the actual corrective action $u$, based on the quantized synchronization error measurement $[e]$.

Figure 4 shows the resulting switched control scheme with $\hat{R}$ being (5), $P$ being the synchronization error dynamics (3), and $z^{-1}$ being a unit delay operator in discrete time.

The switched control system dynamics is characterized by $e$ as per (3), and by $u$, that can be computed as:

- if $[e(k + 1)] = [e(k) + \rho(u(k)) + d(k)] = 0$, then:
  $$u(k + 1) = \rho(u(k)) + [e(k)]$$

(6)
The other FLOPSYNC switched variant of the scheme. The other two, one runs the bare FLOPSYNC scheme, and the control scheme is able to reject them. The performed experimental test provide experimental evidence that this is practically constant, according to the considerations in Section III-B. Real disturbances are actually varying, but it is safely assumed that such variability occurs at a time scale much longer than the synchronization period, and therefore we can consider it as an application for the Miosix\textsuperscript{1} microcontroller operating system. For our implementation, we decided to use $\alpha = 11/8$, since it preserves stability of the closed-loop linear dynamics, and it can be easily implemented in an embedded device with fixed-point arithmetic.

A. An experimental test

We assessed the performance of the proposed solution in a real-world setting, where the actual disturbance acting on the system is not known, but it can be considered practically constant, according to the considerations in Section III-B. Real disturbances are actually varying, but it is safely assumed that such variability occurs at a time scale much longer than the synchronization period, and therefore the control scheme is able to reject them. The performed experimental test provide experimental evidence that this is the case.

In the test, three nodes are used. One plays the role of the master, broadcasting synchronization packets. Out of the other two, one runs the bare FLOPSYNC scheme, and the other FLOPSYNC switched variant of the scheme. The nodes are placed in an office environment, therefore exposed to radio interference from local wireless networks, and to temperature variations like those encountered in a typical indoor setting with standard climatization.

The synchronization period $T$ is 10 seconds. The nodes’ hardware timers have a measurement and actuation resolution – the tick – of 30.5$\mu$s, which is the source of quantization, and it is normalized to 1. The control parameter $\alpha$ was set to 11/8. In order to show the long-term behavior of the system in the face of slowly varying disturbances, the experiment was set to last 10 hours.

Figure 6 shows both the quantized synchronization error in ticks for FLOPSYNC (left column) and FLOPSYNC-QACS (right column). The horizontal axes report the experiment time in hours. The $[-1, 1]$ synchronization error range is highlighted in both top plots with a gray area.

Notice that in the case of FLOPSYNC, the gray area is practically covered by the quantized synchronization error trajectory. This is because the quantized synchronization error oscillates between $\{-1, 0, 1\}$ with an excursion of amplitude 2.

In the case of FLOPSYNC-QACS, the quantized synchronization error first switches between $\{-1, 0\}$, then, after a brief transient, it switches between $\{0, 1\}$. For practically the whole experiment, the quantized synchronization error has an excursion of amplitude 1. More in details, the error lies in the $\{-1, 0\}$ or $\{0, 1\}$ range for 99.3% of the time.

We compare the two results by computing the Root Mean Square (RMS) performance index of the quantized synchronization error. The RMS computed in the case of FLOPSYNC is 0.878, while the RMS computed for FLOPSYNC-QACS is 0.499, i.e., about half than with bare FLOPSYNC. We can conclude that the proposed control scheme results in a lower RMS error magnitude also in a practical setting, where the disturbance is not rigorously constant.

Analyzing a bit more in detail the obtained result of

\begin{equation}
\text{if } |e(k+1)| = |e(k) + \rho(u(k)) + d(k)| \neq 0, \text{ then:} \\
u(k+1) = u(k) + |e(k)| + \alpha |e(k) + \rho(u(k)) + d(k)|
\end{equation}

\[u(k+1) = u(k) + |e(k)| - \alpha |e(k) + \rho(u(k)) + d(k)|
\]

Figure 6: Experimental results comparing FLOPSYNC and FLOPSYNC-QACS.

Figure 5: One of the WandStem WSN nodes that was used to test the FLOPSYNC-QACS scheme.
FLOPSYNC-QACS, after about 8 hours and 30 minutes, the quantized synchronization error starts switching between the values \{-1, 0, 1\}, and then settles to a new regime, switching only between 0 and 1. This type of behavior is probably explained considering that the disturbance is not exactly constant, but it might have a slowly varying behavior.

In order to better investigate what caused the transition, we performed a simulation study trying to replicate the same behavior with a slowly changing disturbance. Figure 7 shows the quantized synchronization error, and the disturbance acting on the system with its quantization. Remember that the corrective action is rounded, and it needs to compensate the quantized disturbance, i.e., in principle a perfect compensation is $u(k) = -\rho(d(k))$. Therefore, the transition between the two regimes happen exactly at the time when the disturbance crosses the 0.5 threshold of the rounding. More specifically, we selected a disturbance that starts as a constant $d = 0.6$, i.e., $\Delta_d = -0.4 < 0$. Then from time $k = 100$ the disturbance slowly decreases linearly up to the value $d = 0.4$, i.e., $\Delta_d = 0.4 > 0$. Finally, the disturbance keeps constant and equal to $d_0$. The simulation produces exactly the same behavior that can be observed in the experimental data of Figure 6. We can thus conclude that the behavior that appeared in the experimental results may have been caused by a disturbance similar to the one presented in the bottom graph of Figure 7.

V. CONCLUSIONS AND FUTURE WORK

A control-based time synchronization mechanism for WSNs, called FLOPSYNC-QACS, was proposed for reducing the degradation effect due to quantization of both corrective actions and synchronization error. FLOPSYNC-QACS was implemented in a real WSN, and experimental results back up the proposed solution.

As a future work, we plan to provide formal guarantees on the obtainable performance of the proposed approach, like, for example, convergence time of the algorithm. We also plan to include a more thorough discussion and experimental analysis of the power consumption obtained with the proposed methodology, compared to state-of-the-art approaches.

REFERENCES

Cost minimization of network services with buffer and end-to-end deadline constraints

Victor Millnert*, Johan Eker†, Enrico Bini‡
*Lund University, Sweden
†Ericsson Research, Sweden
‡Scuola Superiore Sant’Anna, Pisa, Italy

Abstract—Cloud computing technology provides the means to share physical resources among multiple users and data center tenants by exposing them as virtual resources. There is a strong industrial drive to use similar technology and concepts to provide timing sensitive services. One such is virtual networking services, so called services chains, which consist of several interconnected virtual network functions. This allows for the capacity to be scaled up and down by adding or removing virtual resources. In this work, we develop a model of a service chain and pose the dynamic allocation of resources as an optimization problem. We design and present a set of strategies to allot virtual network nodes in an optimal fashion subject to latency and buffer constraints.

1. Introduction

Over the last years, cloud computing has swiftly transformed the IT infrastructure landscape, leading to large cost-savings for deployment of a wide range of IT applications. Some main characteristics of cloud computing are resource pooling, elasticity, and metering. Physical resources such as compute nodes, storage nodes, and network fabrics are shared among tenants. Virtual resource elasticity brings the ability to dynamically change the amount of allocated resources, for example as a function of workload or cost. Resource usage is metered and in most pricing models the tenant only pays for the allocated capacity.

While cloud technology initially was mostly used for IT applications, e.g. web servers, databases, etc., it is rapidly finding its way into new domains. One such domain is processing of network packages. Today network services are packaged as physical appliances that are connected together using physical network. Network services consist of interconnected network functions (NF). Examples of network functions are firewalls, deep packet inspections, transcoding, etc. A recent initiative from the standardisation body ETSI (European Telecommunications Standards Institute) addresses the standardisation of virtual network services under the name Network Functions Virtualisation (NFV) [1].

The expected benefits from this are, among others, better hardware utilisation and more flexibility, which translate into reduced capital and operating expenses (CAPEX and OPEX). A number of interesting use cases are found in [2], and in this paper we are investigating the one referred to as Virtual Network Functions Forwarding Graphs, see Figure 1.

We investigate the allocation of virtual resources to a given packet flow, i.e. what is the most cost efficient way to allocate VNFs with a given capacity that still provide a network service within a given latency bound? The distilled problem is illustrated as the packet flows in Figure 1. The forwarding graph is implemented as a chain of virtual network nodes, also known as a service chains. To ensure that the capacity of a service chain matches the time-varying load, the number of instances $m_i$ of each individual network function $VNF_i$ may be scaled up or down.

The contribution of the paper is

- a mathematical model of the virtual resources supporting the packet flows in Figure 1,
- the set-up of an optimization problem for controlling the number of machines needed by each function in the service chain,
- solution of the optimization-problem leading to a control-scheme of the number of machines needed to guarantee that the end-to-end deadline is met for incoming packets under a constant input flow.

Related work

There are a number of well known and established resource management frameworks for data centers, but few of them explicitly address the issue of latency. Sparrow [3] presents an approach for scheduling a large number of parallel jobs with short deadlines. The problem domain is different compared to our work in that we focus on sequential rather than parallel jobs. Chronos [4] focuses on reducing latency on the communication stack. RT-OpenStack [5] adds real-time performance to OpenStack by usage of a real-time hypervisor and a timing-aware VM-to-host mapping.

The enforcement of an end-to-end (E2E) deadline of a sequence of jobs to be executed through a sequence of computing elements was addressed by several works, possibly
under different terminologies. In the holistic analysis [6], [7], [8] the schedulability analysis is performed locally. At global level the local response times are transformed into jitter or offset constraints for the subsequent tasks.

A second approach to guarantee an E2E deadline is to split a constraint into several local deadline constraints. While this approach avoids the iteration of the analysis, it requires an effective splitting method. Di Natale and Stankovic [9] proposed to split the E2E deadline proportionally to the local computation time or to divide equally the slack time. Later, Jiang [10] used time slices to decouple the schedulability analysis of each node, reducing the complexity of the analysis. Such an approach improves the robustness of the schedule, and allows to analyse each pipeline in isolation. Serreli et al. [11], [12] proposed to assign local deadlines to minimize a linear upper bound of the resulting local demand bound functions. More recently, Hong et al [13] formulated the local deadline assignment problem as a Mixed-Integer Linear Program (MILP) with the goal of maximizing the slack time. After local deadlines are assigned, the processor demand criterion was used to analyze distributed real-time pipelines [14], [12].

In all the mentioned works, jobs have non-negligible execution times. Hence, their delay is caused by the preemption experienced at each function. In our context, which is scheduling of virtual network services, jobs are executed non-preemptively and in FIFO order. Hence, the impact of the local computation onto the E2E delay of a request is intensively investigated in the networking community in the broad area queuing systems [15]. In this area, Henriksson et al. [16] proposed a feedforward/feedback controller to adjust the processing speed to match a given delay target.

Most of the works in queuing theory assumes a stochastic (usually markovian) model of job arrivals and service times. A solid contribution to the theory of deterministic queuing systems is due to Baccelli et al. [17], Cruz [18], and Parekh & Gallager [19]. These results built the foundation for the network calculus [20], later applied to real-time systems in the real-time calculus [21]. The advantage of network/real-time calculus is that, together with an analysis of the E2E delays, the sizes of the queues are also modelled. As in the cloud computing scenario the impact of the queue is very relevant since that is part of the resource usage which we aim to minimize, hence we follow this type of modeling.

2. Problem formulation

Section 1 we consider a service-chain consisting of functions $F_1, \ldots, F_n$, as illustrated in Figure 2. Packets are flowing through the service-chain and they must be processed by each function in the chain within some end-to-end deadline, denoted by $D_{\text{max}}$. A fluid model is used to approximate the packet flow and at time $t$ there are $r_{i-1}(t) \in \mathbb{R}^+$ packets per second (pps) entering the $i$th function and the cumulative arrived requests for this function is

$$R_{i-1}(t) = \int_0^t r_{i-1}(\tau) \, d\tau. \quad (1)$$

In a recent benchmarking study it was shown that a typical virtual machine can process around 0.1–2.8 million packets per second, [22]. Hence, in this work the number of packets flowing through the functions is assumed to be in the order of millions of packets per second, supporting the use of a fluid model.

![Figure 2: Illustration of the service-chain.](image)

2.1. Service model

As illustrated in Figure 3, the incoming requests to function $F_i$ are stored in the queue and then processed once it reaches the head of the queue. At time $t$ there are $m_i(t) \in \mathbb{Z}^+$ machines ready to serve the requests, each with a nominal speed of $s_i \in \mathbb{R}^+$ (note that this nominal speed might differ between different functions in the service chain, i.e. it does not in general hold that $s_i = s_j$ for $i \neq j$). The maximum speed that function $F_i$ can process requests at is thus $m_i(t)s_i$. The rate by which $F_i$ is actually processing requests at time $t$ is denoted $s_i(t) \in \mathbb{R}^+$. The cumulative served requests is defined as

$$S_i(t) = \int_0^t s_i(\tau) \, d\tau. \quad (2)$$

At time $t$ the number of requests stored in the queue is defined as the queue length $q_i(t) \in \mathbb{R}^+$:

$$q_i(t) = \int_0^t (r_{i-1}(\tau) - s_i(\tau)) \, d\tau = R_{i-1}(t) - S_i(t). \quad (3)$$

Each function has a fixed maximum-queue capacity $q_i^{\text{max}} \in \mathbb{R}^+$, representing the largest number of requests that can be stored at the function $F_i$.

The queueing delay, depends on the status of the queue as well as on the service rate. We denote by $D_{i,j}(t)$ the time taken by a request from when it enters function $F_i$ to when it exits $F_j$, with $j \geq i$, where $t$ is the time when the request exits function $F_j$:

$$D_{i,j}(t) = \inf \{ \tau \geq 0 : R_{i-1}(t-\tau) \leq S_j(t) \}. \quad (4)$$

The maximum queueing delay then is $D_{i,j} = \max_{t \geq 0} D_{i,j}(t)$. The requirement that a requests meets it end-to-end deadline is $D_{1,n} \leq D_{\text{max}}$.

To control the queueing delay, it is necessary to control the service rate of the function. Therefore, we assume that it is possible to change the maximum service-rate of a function by changing the number of machines that are on, i.e.

![Figure 3: Illustration of the structure and different entities of the service chain.](image)
changing \(m_i(t)\). However, turning on a machine takes \(\Delta_i^\text{on}\) time units, and turning off a machine takes \(\Delta_i^\text{off}\) time units. Together they account for a *time delay*, \(\Delta_i = \Delta_i^\text{on} + \Delta_i^\text{off}\), associated with turning on/off a machine.

In 2012 Google profiled where the latency in a data center occurred, [4]. They showed that less than 1% (\(\approx 1\mu s\)) of the latency occurred was due to the propagation in the network fabric. The other 99% (\(\approx 85\mu s\)) occurred somewhere in the kernel, the switches, the memory, or the application. Since it is difficult to say exactly which of this 99% is due to processing, or queueing, we make the abstraction of considering queueing delay and processing delay together, simply as queueing delay. Furthermore, we assume that no request is lost in the communication links, and that there is no propagation delay. Hence the concatenation of the functions \(F_1\) through \(F_n\) implies that the input of function \(F_1\) is exactly the output of function \(F_{i-1}\), for \(i = 2, \ldots, n\), as illustrated in Figure 2.

### 2.2. Cost model

To be able to provide guarantees about the behaviour of the service chain, it is necessary to make hard reservations of the resources needed by each function in the chain. This means that when a certain resource is reserved, it is guaranteed to be available for utilization. Reserving this resource results in a cost, and due to the hard reservation, the cost does not depend on the actual utilisation, but only on the resource reserved.

The computation cost per time-unit per machine is denoted \(J_i^c\), and can be seen as the cost for the CPU-cycles needed by one machine in \(F_i\). This cost will also occur during the time-delay \(\Delta_i\). Without being too conservative, this time-delay can be assumed to occur only when a machine is started. The average computing cost per time-unit for the whole function \(F_i\) is then

\[
J_i^c (m_i(t)) = \lim_{t \to +1} \frac{1}{T} \int_0^T m_i(s) + \Delta_i \cdot \partial_- m_i(s) \cdot ds \tag{4}
\]

where \((x)_+ = \max(x, 0)\), and \(\partial_- m_i(t)\) is the left-limit of \(m_i(t)\):

\[
\partial_- m_i(t) = \lim_{a \to t^-} \frac{m_i(t) - m_i(a)}{t - a},
\]

that is, a sequence of Dirac’s deltas at all points where the number of machines changes. This means that the value of the left-limit of \(m_i(t)\) is only adding to the computation-cost whenever it is positive, i.e., when a machine is switched on.

The queue cost per time-unit per space for a request is denoted \(J_i^q\). This can be seen as the cost that comes from the fact that physical storage needs to be reserved such that a queue can be hosted on it, normally this would correspond to the RAM of the network-card. Reserving the capacity of \(q_i^\text{max}\) would thus result in a cost per time-unit of

\[
J_i^q (q_i^\text{max}) = J_i^q q_i^\text{max}. \tag{5}
\]

### 2.3. Problem definition

The aim of this paper is to control the number \(m_i(t)\) of machines running in function \(F_i\), such that the total average cost is minimized, while the E2E constraint \(D_i\) is not violated and the maximum queue sizes \(q_i^\text{max}\) are not exceeded. This can be posed as the following problem:

\[
\begin{align*}
\text{minimize} & \quad J = \sum_{i=1}^n J_i^c (m_i(t)) + J_i^q (q_i^\text{max}) \\
\text{subject to} & \quad \bar{D}_{i,n} \leq D_i^\text{max} \\
& \quad q_i(t) \leq q_i^\text{max}, \quad \forall t \geq 0, \quad i = 1, 2, \ldots, n
\end{align*} \tag{6}
\]

with \(J_i^c\) and \(J_i^q\) as in (4) and (5), respectively. In this paper the optimization problem (6) will be solved for a service-chain fed with a constant incoming rate \(r\).

A valid lower bound \(J_i^\text{lb}\) to the cost achieved by any feasible solution of (6) is found by assuming that all functions are capable of providing exactly a service rate \(r\) equal to the input rate. This is possible by running a fractional number of machines \(r/s_i\) at function \(F_i\). In such an ideal case, buffers can be of zero size \((\forall i, q_i^\text{max} = 0)\), and there is no queuing delay \((\bar{D}_{i,n} = 0)\) since service and the arrival rates are the same at all functions. Hence, the lower bound to the cost is

\[
J_i^\text{lb} = \sum_{i=1}^n J_i^c \frac{r}{s_i} \tag{7}
\]

Such a lower bound will be used to compare the quality of the solution found later on.

In Section 3 we make a general consideration about the on/off scheme of each machine in presence of a constant input rate \(r\). Later in Section 4, the optimal design problem of (6) is solved.

### 3. Machine switching scheme

In presence of an incoming flow of requests at a constant rate \(r_0(t) = r\), a number

\[
\bar{m}_i = \left\lfloor \frac{r}{s_i} \right\rfloor \tag{8}
\]

of machines running in function \(F_i\) must always stay on. To match the incoming rate \(r\), in addition to the \(\bar{m}_i\) machines always on, another machine must be on for some time in order to process a request rate of \(s_i\rho_i\) where \(\rho_i\) is the normalized residual request rate:

\[
\rho_i = r/s_i - \bar{m}_i, \tag{9}
\]

where \(\rho_i \in [0, 1)\).

In our scheme, the extra machine is switched on at a desired on-time \(t_i^\text{on}\):

- off \(\rightarrow\) on: function \(F_i\) switches on the additional machine when the time \(t\) exceeds \(t_i^\text{on}\).

Since the additional machine does not need to always be on, it could be switched off after some time. The off-switching is also based on a time-condition, the desired stop-time \(t_i^\text{off}\), i.e., the time-instance that the machine should be switched off, and is given by:

\[
t_i^\text{off} = t_i^\text{on} + T_i^\text{on},
\]

where \(T_i^\text{on}\) is the duration that the machine should be on for, and something that needs to be found. The off-switching is then triggered in the following way:

- on \(\rightarrow\) off: function \(F_i\) switches off the additional machine when the time \(t\) exceeds \(t_i^\text{off}\).

Note that this control-scheme, in addition with the constant input, result in the extra machine being switched on/off.
periodically, with a period $T_i$. We thus assume that the extra machine can process requests for a time $T_i^\text{on}$ every period $T_i$. The time during each period where the machine is not processing any requests is denoted $T_i^\text{off} = T_i - T_i^\text{on}$. Notice, however, that the actual time the extra machine is consuming power is $T_i^\text{on} + \Delta_i$ due to the time-delay for starting a new machine.

In the presence of a constant input, it is straightforward to find the necessary on-time during each period—in order for the additional machine to provide the residual processing capacity of $r - \bar{m}_i \bar{s}_i$, its on-time $T_i^\text{on}$ must be such that

$$T_i^\text{on} \bar{s}_i = T_i(r - \bar{m}_i \bar{s}_i),$$

which implies

$$T_i^\text{on} = T_i \rho_i, \quad T_i^\text{off} = T_i - T_i^\text{on} = T_i(1 - \rho_i). \quad (10)$$

With each additional machine being switched on/off periodically, it is also straightforward to find the computation cost for each function. If $\bar{m}_i + 1$ machines are on for a time $T_i^\text{on}$, and only $\bar{m}_i$ machines are on for a time $T_i^\text{off}$, then the cost $J_i^c$ of (4) becomes

$$J_i^c = \bar{f}_i \left( (T_i^\text{on} + \Delta_i) / T_i + \bar{m}_i \right) = \bar{f}_i \left( \bar{m}_i + \rho_i + \Delta_i / T_i \right) \quad (11)$$

if $T_i^\text{off} \geq \Delta_i$. If instead $T_i^\text{off} < \Delta_i$, that is if

$$T_i < T_i^* := \frac{\Delta_i}{1 - \rho_i},$$

then there is no time to switch the additional machine off and then on again before the next period start. Hence, we keep the last machine on, even if it is not processing packets, and the computing cost becomes

$$J_i^c = \bar{f}_i \left( \bar{m}_i + \rho_i + T_i^\text{off} / T_i \right) = \bar{f}_i (\bar{m}_i + 1). \quad (13)$$

Next, using this control-scheme, the optimization problem of (6) will be studied and solved under the assumption that every function will switch on/off its additional machine with the same period, $T$.

4. Design of machine-switching period

In this section we solve the optimization problem (6) under the assumption of a constant input. In order to somewhat reduce the complexity of the solution we also make the assumption of letting every function switch its additional machine on/off with the same period, $T_i = T$. The common period $T$ of the schedule, by which every function switches its additional machine on/off, is the only design variable in the optimization problem (6). In Lemma 1 and Lemma 2 below, the maximum queue size $q_i^\text{max}$ of any function $F_i$ and the end-to-end delay $D_{1,n}$ are both shown to be proportional to the switching period $T$. The intuition behind this fact is that the longer the period $T$ is, the longer a function will have to wait with the additional machine being off, before turning it on again. During this interval of time, each function is accumulating work and consequently both the maximum queue-size and the delay grows with $T$.

Figure 4 illustrate how two functions in a service-chain, $F_i$ and $F_{i-1}$, switch on/off their additional machine with the same period $T$. However, one should note that they are not on for the same duration. In this example, the input rate to the service-chain is $r = 17$, the nominal service-rate of the first and second function is $\bar{s}_{i-1} = 6$ and $\bar{s}_i = 8$ respectively. The machine-switching period is $T = 120$, and the on-time for the two additional machines are $T_{\text{on},i} = 100$ and $T_{\text{on},i-1} = 15$ respectively. The maximum queue-size needed for the second machine is $q_i^\text{max} = 90$.

In order to solve the optimization problem one need two ingredients. The first ingredient is the expression for the maximum queue-size needed for a given period $T$:

**Lemma 1.** With a constant input rate $r_0(t) = r$, along with all functions switching on/off their additional machine with a common period $T$, the maximum queue size $q_i^\text{max}$ at function $F_i$ is

$$q_i^\text{max} = T \times \alpha_i,$$  \quad (14)

where

$$\alpha_i = \max \{ \rho_i (\bar{s}_i (1 - \rho_i) - \bar{s}_{i-1} (1 - \rho_{i-1})),$$

$$(1 - \rho_{i-1}) (\bar{s}_{i-1} - \bar{s}_{i} - \bar{s}_i \rho_i),$$.  

$$\rho_{i-1} (\bar{s}_{i-1} (1 - \rho_{i-1}) - \bar{s}_i (1 - \rho_i)),$$

$$(1 - \rho_i) (\bar{s}_i \rho_i - \bar{s}_{i-1} \rho_{i-1}) \},$$

with $\rho_i$ as defined in (9), and $T$ being the period of the switching scheme, common to all functions.

**Proof.** Due to limited space the proof is shown in a technical report published at Lund University Publications, [23].

The expression of $q_i^\text{max}$ in Eq. (14) suggests a property that is condensed in the next Corollary.

**Corollary 1.** The maximum queue size $q_i^\text{max}$ at any function $F_i$ is bounded, regardless of the rate $r$ of the input.

**Proof.** From the definition of $\rho_i$ in Eq. (9), it always holds that $\rho_i \in [0, 1]$. Hence, from the expression of (14), it follows that $q_i^\text{max}$ is always bounded.

The second ingredient needed to solve the optimal design problem is the expression of how the end-to-end delay relate to the switching period $T$.

1. https://lup.lub.lu.se/search/publication/8c7b837c-bca3-4375-bb9d-28ce6bbcb889a
Lemma 2. With a constant input rate, \( r_0(t) = r \), the longest end-to-end delay \( \hat{D}_{1,n} \) for any request passing through functions \( F_1 \) thru \( F_n \) is

\[
\hat{D}_{1,n} = T \times \sum_{i=1}^{n} \delta_i, \tag{15}
\]

with \( \delta_i \) being an opportunity constant that depends on \( r, \bar{s}_i \), and \( \bar{s}_{i-1} \).

Proof. Due to limited space the proof is shown in a technical report published at Lund University Publications, [23]. \( \square \)

Solution to the optimization problem

With these hypothesis, the cost function of the optimization problem (6) becomes

\[
J(T) = aT + \sum_{i : T < T_i} f_i^j(1 - \rho_i) + \sum_{i : T \geq T_i} \frac{\Delta_i}{T} \bar{s}_i + J_i^\text{fb}, \tag{16}
\]

where \( J_i^\text{fb} \) is the lower bound given by (7) and \( a = \sum_{j=1}^{n} j_i \alpha_i \), where \( \alpha_i \) is given by Lemma 1. Furthermore, \( T_i \) (defined in (12)) represents the value of the period below which it is not feasible to switch the additional machine off and then on again \( (T < T_i \iff T_i < \Delta_i) \). In fact, \( \forall i \) with \( T < T_i \) we pay the full cost of having \( \bar{m}_i \) + 1 machines always on.

The deadline constraint in (6), can be simply written as

\[
T \leq c := \frac{D_i^{\max}}{\sum_{i=1}^{n} \delta_i},
\]

with \( \delta_i \) given in Lemma 2.

The cost \( J(T) \) of (16) is a continuous function of one variable \( T \). It has to be minimized over the closed interval \([0, c]\). Hence, by the Weierstraß’s extreme-value theorem, it has a minimum. To find this minimum, we just check all (finite) points at which the cost is not differentiable and the ones where the derivative is equal to zero. Let us define all points in \([0, c] \) in which \( J(T) \) is not differentiable:

\[
C = \{ T_i : T_i < c \} \cup \{ 0 \} \cup \{ c \}. \tag{17}
\]

We denote by \( p = |C| \leq n + 2 \) the number of points in \( C \). Also, we denote by \( c_k \in C \) the points in \( C \) and we assume they are ordered increasingly \( c_1 < c_2 < \ldots < c_p \). Since the cost \( J(T) \) is differentiable over the open interval \((c_k, c_{k+1})\), the minimum may also occur at an interior point of \((c_k, c_{k+1})\) with derivative equal to zero. Let us denote by \( C^* \) the set of all interior points of \((c_k, c_{k+1})\) with derivative of \( J(T) \) equal to zero, that is

\[
C^* = \{ c_k^* : k = 1, \ldots, p - 1, \ c_k < c_k^* < c_{k+1} \} \tag{18}
\]

with

\[
c_k^* = \frac{\sum_{i : T_i < c_{k+1}} f_i^j \Delta_i}{a}.
\]

Then, the optimal period is given by

\[
T^* = \arg \min_{T \in \{ C \cup C^* \}} \{ J(T) \}. \tag{19}
\]

Example. We use an example to illustrate the solution of the optimization problem of a service chain with two functions. The input rate of the service chain is \( r_0(t) = r = 17 \). Every request has an E2E-deadline of \( D_{\max} = 0.02 \). The parameters of the two functions are reported in Table 1.

![Table 1: Parameters of the example.](image)

The input \( r_0(t) = r \) can be seen as dummy function \( F_0 \), preceding \( F_1 \), with \( \bar{s}_0 = r, \bar{m}_0 = 1, \) and \( r_0 = 0 \). From (8) and (9) it follows that \( \bar{m}_1 = \bar{m}_2 = 2, \) and \( \rho_1 = \frac{7}{10}, \rho_2 = \frac{3}{7}, \) implying that both functions must always keep two machines on, and then periodically switch a third one on/off. This leads to \( T_1 = 60.0 \times 10^{-3} \) and \( T_2 = 11.4 \times 10^{-3}, \) where \( T_1 \) is the threshold period for function \( F_1 \), as defined in (12). From Lemma 1 it follows that the parameter \( a \) of the cost function (16) is \( a = 0.792 \), while from Lemma 2 the parameters \( \delta_i \) determining the queueing delay introduced by each function, are \( \delta_1 = 49.0 \times 10^{-3} \) and \( \delta_2 = 22.1 \times 10^{-3} \), which in turn leads to

\[
c = \frac{D_{\max}}{\delta_1 + \delta_2} = \frac{0.02}{71.1 \times 10^{-3}} = 281 \times 10^{-3}.
\]

Since \( T_2 < T_1 < c \), the set \( C \) of (17) containing the boundary is

\[
C = \{ 0, 0.00114, 0.060, 0.281 \}.
\]

To compute the set \( C^* \) of interior points with derivative equal to zero defined in (18), which is needed to compute the period with minimum cost from (19), we must check all intervals with boundaries at two consecutive points in \( C \). In the interval \( (0, T_2) \) the derivative of \( J \) is never zero. When checking the interval \( (T_2, T_1) \), the derivative is zero at

\[
c_1^* = \sqrt{\frac{f_1^j \Delta_2}{a}} = 0.318,
\]

which, however, falls outside the interval. Finally, when checking the interval \( (T_1, c) \) the derivative is zero at

\[
c_2^* = \sqrt{\frac{f_1^j \Delta_1 + f_2^j \Delta_2}{a}} = 0.421 > c = 0.281.
\]

Hence, the set of points with derivative equal to zero is \( C^* = \emptyset \). By inspecting the cost at points in \( C \) we find that the minimum occurs at \( T^* = c = 0.281 \), with cost \( J(T^*) = 34.7 \). To conclude the example we show the state-space trajectory for the two queues in Figure 5. Again, it should be noted that this example is meant to illustrate how one can use the design methodology of this section in order to find the best period \( T \). In a real setting the incoming traffic will likely be around millions of requests per second, [24].

![Figure 5: State-space trajectory for the example in Section 4. (on, off) correspond to \( F_1 \) having its additional machine on, while \( F_2 \) has its extra machine off.](image)
5. Summary

In this paper we have developed a general mathematical model for a service-chain residing in a Cloud environment. This model includes an input model, a service model, and a cost model. The input-model defines the input-stream of requests to each NFV along with end-to-end deadlines for the requests, meaning that they have to pass through the service-chain before this deadline. In the service-model, we define an abstract model of a NFV, in which requests are processed by a number of machines inside the service function. It is assumed that each function can change the number of machines that are up and running, but doing so is assumed to take some time. The cost-model defines the cost for allocating compute- and storage capacity, and naturally leads to the optimization problem of how to allocate the resources. We analyze the case with a constant input-stream of requests and derive control-strategies for this. This is a simplified case it will constitute the foundation of adaptive schemes to time-varying requests in the future.

We plan to extend this work by allowing for a dynamic input as well as uncertainties in the true performance of the machines running in the functions, leading to the need of using a more advanced feedback loop to guarantee the desired performance.

Acknowledgements. The authors would like to thank Karl-Erik Årzén and Bengt Lindoff for the useful comments on early versions of this paper.

Source code. The source code used to compute the solution of the example in Section 4 can be found on Github at https://github.com/vmillnert/REACTION-source-code.

References

DTFM: a Flexible Model for Schedulability Analysis of Real-Time Applications on NoC-based Architectures

Mourad Dridi*, Stéphane Rubini*, Frank Singhoff*, Jean-Philippe Diguë†,*
*Lab-STIC, CNRS, UMR6285, Univ. Bretagne Occidentale, 29100 Brest, France
†Lab-STIC, CNRS, UMR6285, Univ. Bretagne Sud, 56100 Lorient, France
{mourad.dridi, stephane.rubini, frank.singhoff}@univ-brest.fr, jean-philippe.diguet@univ-ubs.fr

Abstract—Many-core processors are expected to be hardware targets to support the execution of real-time applications. In a many-core processor, cores communicate through a Network-On-Chip (NoC), which offers high bandwidth and scalability, but also introduces contentions leading to additional variability to task execution times. Such contentions also strongly increase the pessimistic trend of worst case execution time estimation. Consequently, modeling and analysis of network contentions interferences on many-core processors is a challenge to support real-time applications. In this article, we formalize a dual task and flow model called DTFM. From the specification of a real-time application composed of a set of tasks and their communication dependencies, DTFM allows us to compute flow requirements and to assess predictability of the tasks. DTFM is extensible enough to be adapted to various NoCs and task models, allowing designers to compare candidate software and NoC architectures. Furthermore, we introduce an original validation approach based on the cross-use of a task level real-time scheduling analysis tool and a cycle-accurate SystemC NoC simulator.

I. INTRODUCTION

NoCs are widely used in many-core systems since they provide scalability, modularity, and communication parallelism. Many-core allows multiple applications to run at the same time in the same processor. These applications exchange messages through the communication infrastructure. Real-time applications have very stringent communication service requirements. The correctness of real-time communications depends not only on the communication result but also on the completion time bound [1].

In this article, we address temporal failures which occur when a real-time application is unable to meet its deadlines. To predict whether a given application is able to meet its deadline, usual schedulability analysis methods require to know the Worst Case Execution Time (WCET) of each task [2]. Then, schedulability analysis methods allow designers to investigate how the tasks will be scheduled on the hardware platform and how task executions will be delayed due to hardware contentions.

A. Problem Statement

Many task models were proposed to deal with real-time applications [2]. All these models are not immediately suitable with NoC architectures. They do not take into account the communications through the network, the task mapping and the latencies introduced by this new type of shared resources. Many traffic-flow models are also proposed to analyze the real-time communications for NoC architectures. All these models focus on real-time traffic-flows and they do not take into account the schedulability of tasks. A schedulable traffic-flow does not allow us to conclude that the whole system is schedulable. Moreover, they do not offer the flexibility, which is necessary to consider different communication protocols. Considering the incomplete state of the art, we investigate how to formalize and implement task and flow models that will provide a common platform for the evaluation of different solutions. Such a platform is required to run simulations and perform schedulability analysis with real-life cases in order to compare different solutions at hardware, software, scheduling or protocol levels.

B. Contributions

We first propose DTFM, a Dual Task and Flow Model, that allows designers to perform a complete schedulability analysis in order to assess the predictability of real-time tasks running over NoC-based architectures.

The second contribution is a unique validation environment, which combines a tick accurate real-time simulator and analysis tool that implements the DTFM model and produces a task scheduling, which is then used to feed a SystemC cycle accurate NoC simulator. Such a validation environment could be applied to investigate the temporal behaviour of real-time applications running on various other hardware components such as cache hierarchies. The analysis tool, that implements DTFM, allows us to evaluate and compare different NoC architectures, which can be standard or designed for mixed criticality systems. It also allows us to consider different types of task communication protocols.

The remainder of the paper is organized as follows. The next section presents related works. Section III introduces background elements about the NoC we consider. Then, section IV proposes our dual flow and task model. Implementation and evaluation of DTFM are explained in section V and section VI concludes the article.

II. RELATED WORKS

Several schedulability analysis approaches for real-time on-chip communication have been explored.

In [3], the authors propose an analysis approach for real-time on chip communication with wormhole switching and fixed priority scheduling. [1] focuses on real-time communication service with a priority share policy.

In all these works, the scheduling analysis of the real-time applications is computed for a given flow model and for a given task model and they do not take into account the schedulability of tasks. While in our work, we expect to compute the flow model from a given task model so we examine simultaneously task and flow schedulability. Moreover we provide a flexible model in order to consider different communication protocols and different NoC architectures.

There are also several existing tools and frameworks which support mapping of applications onto NoC-based architectures. [4] proposes a simulator developed in SystemC with a C++ plugin,
in order to model concurrent hardware modules at the cycle-level accuracy.

There are also other simulators such as [5] which allows users to setup and simulate a broad range of network configurations defined by network size, buffer size and routing algorithm.

In order to evaluate the usage of Many-Core architecture for critical applications, authors in [6] propose a simple one task to one core execution model and develops test drive programs. The developed programs evaluate the NoC latency performance.

All of these simulators provide Transaction-Level Model of NoC architectures, without any consideration for Task schedulability. On the other hand, DTFM, with the tools that support it, allows us to analyze the schedulability of real-time systems along with a cycle-accurate implementation of communications so that a real-time communication analysis can also be performed.

III. BACKGROUND

In this section, we introduce the NoC we assume in this work. Then, we identify the timing issues a real-time application faces when running over NoC architectures.

A. NoC

A NoC is a network of nodes which can be a processor, a memory a peripheral or a cluster. A node can access the network through a network interface (NI) that usually transmit and receives data in the form of packets.

Wormhole is largely adopted in NoCs because it does not require large capacity buffers. In a wormhole network, the packet is divided into a number of fixed size flits [7]. The packet is split into a header flit, one or several body flits and a tail flit. The header flit stores the routing information and builds the route. As the header flit moves ahead along the selected path, the remaining flits follow in a pipeline way and possibly span a number of routers. If the chosen link is busy, the header flit is blocked until the link becomes available. In this case, all the packets will be blocked in routers along the selected path. Then the network will introduce a new latency for the task that we cannot ignore during schedulability analysis.

We assume a standard NoC based on packet-based wormhole switching, with a 2D-mesh topology and XY deterministic routing. In this study we do not consider NoC with flit level preemption or with any specific protocol for real-time traffic.

B. Types of delays in a NoC

For each NoC configuration, we have different types of latency. In this section we focus on latency introduced by the type of NoC assumed in this paper. Let’s see the different types of delays that we must understood to assess schedulability of real-time tasks deployed on such NoC [3], [1].

1) Path delay: The first type of delay is straightforward and can be called the Path Delay:

Definition 1: The Path Delay $DP$ is the time required to send the packet from the transmitter to the receiver when no traffic flow contention exists.

We assume here that packets of a flow do not share any physical link with other flows along the path. This delay is variable and depends on several parameters such as the flow rate and the size of the message [3]. Note that with the path delay, we consider the worst case latency introduced by the network without any contention. If the real-time flow meets its deadline with the worst case then it will meet the deadline for any other scenario.

2) Delay due to Direct Interference: A second kind of delay occurs due to direct interference:

Definition 2: There is a direct interference when two flows share the same physical link at the same time. We called the delay caused by a direct interference between flows a Direct Delay $DD$.

In all situations, the direct interference can be presented as a non-deterministic source of delay. In our analysis, we consider the worst case latency introduced by the direct interference. We assume the packet from the observed traffic-flow is arrived just after other packets for each shared link. We assume also a maximum size of the packets [1].

3) Delay due to Indirect Interference: The last delay we address in this article is called Indirect Interference:

Definition 3: There is an indirect interference when two flows do not share the same physical link but they interfere directly with the same flows. We call the delay caused by an indirect interference, an Indirect Delay, noted $DI$.

We can have many cases where a flow is blocked by another flow which is also blocked by others and so on. In this case, the first flow must wait for a non-deterministic delay. Like the direct delay, an indirect delay is not deterministic because it depends on the history of network, on the flows rate, the messages size and the links states [1].

In this paper, we compute path delay and direct delays according to equations given in [1] and we consider that indirect interferences are unpredictable delays, we then do not handle indirect delays.

IV. THE DUAL TASK AND FLOW MODEL

In this section, we introduce our task and flow model. We also show how to calculate the flow model from the task model. A simple example is presented in order to illustrate those models. Finally, we explain how to compute flow parameters for a given communication protocol.

A. Presentation of the approach

The figure 1 shows the overall approach we propose in this article. From the task model, the mapping of the tasks on the processing units and the NoC model, DTFM allows us to generate the associated flow model. The flow model specifies the messages that have to be transmitted in the NoC to enforce the communications between the tasks. From a flow model and after identification of the different delays elapsed in the NoC, DTFM checks the predictability of the flows, i.e. detects when unpredictable delays may occur. If there is no unpredictable flows, the analysis tool implementing DTFM can compute the deadline and the communication delays of each message, and then, decides whether the system is schedulable. In the sequel, we introduce the task and flow notations used in this article.

B. Task and flow Models

The task model $\Gamma$ comprises $n$ periodic tasks:

$\Gamma = \{ \tau_1, \tau_2, \ldots, \tau_n \}$
Dual properties and timing requirements as follows:

- $O_i \in \mathbb{N}^+$ is the release time of the messages, i.e., the first time when a message of the flow becomes ready to be transmitted.
- $T_i \in \mathbb{N}^+$ is the period of the message. It is the fixed delay between releases of successive messages.
- $D_i \in \mathbb{N}^+$ is the deadline of the message, i.e., the time instant which allows both receiver and transmitter tasks to complete their execution prior their deadlines. Our model has the same restriction as [1] and [3]: each flow deadline must be less than or equal to its period.
- $\Pi_i \in \mathbb{N}^+$ is the priority of the messages. The value 1 denotes the highest priority level while a larger value indicates a lower priority.
- $Node_S i \in \mathbb{N}^+$ is the node running the transmitter task.
- $Node_D i \in \mathbb{N}^+$ is the node running the receiver task.
- $F_i : \psi \rightarrow \Omega^p$
  - $\rho_i \rightarrow F(\rho_i) = \{e_{p,k}, \ldots, e_{i,j}\}$
  - where $\Omega$ is the set of physical links in the NoC. If $a$ and $b$ are neighbor routers, $e_{a,b}$ (resp. $e_{b,a}$) is the physical link from the router $a$ (resp. $b$) to the router $b$ (resp. $a$).
  - $F_i$ is a function that computes the links used by a flow. Said differently, $F_i$ identifies the physical links that will be used by a given message.
  - The function $F_i$ helps us to understand the relationships between the various flows that transit through the network and to determine the interferences between messages sent by the tasks.

From the task model, we can compute the corresponding flow model. For each task given by the $E$ function, we have a flow $\rho$ where the transmitter is the task antecedent and the receiver is the task image.

We define the function $G$ with two variables to calculate the flow model from the task model:

- $G : \Gamma^2 \rightarrow \psi$
  - $(\tau_i, \tau_j) \rightarrow G((\tau_i, \tau_j)) = \rho$
  - $i \in [1 .. n]$; $j \in [1 .. n]$

If $E \left( \tau_i \right) = \tau_j$ then $G((\tau_i, \tau_j)) = \rho = \rho_{i,j}$

with:
- $\Pi(\rho_{i,j}) = \Pi(\tau_i)$
- $Node_S(\rho_{i,j}) = Node(\tau_i) = Node_S i$
- $Node_D(\rho_{i,j}) = Node(\tau_j) = Node_D j$

Function $G$ is surjective. Every element of the codomain is mapped to at least one element of the domain.

$\forall \rho \in \psi; \exists (\tau_i, \tau_j)$ such that $\rho = G((\tau_i, \tau_j))$

You should notice that $G$ depend on the communication protocol used by tasks to exchange their messages.  

Each task leads to a sequence of jobs. Each task $\tau_i$ has a set of properties and timing requirements as follows:

$\tau_i = \{O_i, T_i, C_i, D_i, \Pi_i, Node_S i, Node_D i, F_i\}$

where:

- $O_i \in \mathbb{N}^+$ is the first release time of the task $\tau_i$, i.e., the release time of the first job of $\tau_i$.
- $T_i \in \mathbb{N}^+$ is the period of the task, i.e., the fixed delay between two successive jobs of the task.
- $C_i \in \mathbb{N}^+$ specifies the worst case execution time of a job of the task.
- $D_i \in \mathbb{N}^+$ is the deadline. We note that each task deadline must be less than or equal to its period.
- $\Pi_i \in \mathbb{N}^+$ is the fixed priority of the task. The value 1 denotes the highest priority level while a larger value indicates a lower priority.
- $Node_S i \in \mathbb{N}^+$ identifies the node running the task. This parameter allows us to introduce the mapping configuration, i.e., on which processing unit each task is assigned to.
- $E_i : \Gamma \rightarrow \Gamma^p$
  - $\tau_i \rightarrow E(\tau_i) = \{\tau_j, \ldots, \tau_k\}$
  - The function $E$ allows us to introduce the precedence constraints in the task model. For a given task $\tau_i$, the function $E$ determines all the tasks $\tau_j$ that receive messages from the task $\tau_i$. Furthermore, we also assume that tasks use a given communication protocol to exchange their messages.

The flow model comprises $m$ periodic traffic flows:

$\psi = \{\rho_1, \rho_2, \ldots, \rho_m\}$

Each flow $\rho_i$ raises a sequence of messages in a similar way that a task raises a sequence of jobs. The flow is defined as follows:

$\rho_i = \{O_i, T_i, D_i, \Pi_i, Node_S i, Node_D i, F_i\}$

where:

Each task raises a sequence of jobs. The flow is defined as follows:

$\psi = \{\rho_1, \rho_2, \ldots, \rho_m\}$

Each flow $\rho_i$ raises a sequence of messages in a similar way that a task raises a sequence of jobs. The flow is defined as follows:

$\rho_i = \{O_i, T_i, D_i, \Pi_i, Node_S i, Node_D i, F_i\}$

where:

- $O_i \in \mathbb{N}^+$ is the release time of the messages, i.e., the first time when a message of the flow becomes ready to be transmitted.
- $T_i \in \mathbb{N}^+$ is the period of the message. It is the fixed delay between releases of successive messages.
- $D_i \in \mathbb{N}^+$ is the deadline of the message, i.e., the time instant which allows both receiver and transmitter tasks to complete their execution prior their deadlines. Our model has the same restriction as [1] and [3]: each flow deadline must be less than or equal to its period.
- $\Pi_i \in \mathbb{N}^+$ is the priority of the messages. The value 1 denotes the highest priority level while a larger value indicates a lower priority.
- $Node_S i \in \mathbb{N}^+$ is the node running the transmitter task.
- $Node_D i \in \mathbb{N}^+$ is the node running the receiver task.
- $F_i : \psi \rightarrow \Omega^p$
  - $\rho_i \rightarrow F(\rho_i) = \{e_{p,k}, \ldots, e_{i,j}\}$
  - where $\Omega$ is the set of physical links in the NoC. If $a$ and $b$ are neighbor routers, $e_{a,b}$ (resp. $e_{b,a}$) is the physical link from the router $a$ (resp. $b$) to the router $b$ (resp. $a$).
  - $F_i$ is a function that computes the links used by a flow. Said differently, $F_i$ identifies the physical links that will be used by a given message.
  - The function $F_i$ helps us to understand the relationships between the various flows that transit through the network and to determine the interferences between messages sent by the tasks.

From the task model, we can compute the corresponding flow model. For each task given by the $E$ function, we have a flow $\rho$ where the transmitter is the task antecedent and the receiver is the task image.

We define the function $G$ with two variables to calculate the flow model from the task model:

$G : \Gamma^2 \rightarrow \psi$

$(\tau_i, \tau_j) \rightarrow G((\tau_i, \tau_j)) = \rho$

$i \in [1 .. n]$; $j \in [1 .. n]$

If $E \left( \tau_i \right) = \tau_j$ then $G((\tau_i, \tau_j)) = \rho = \rho_{i,j}$

with:

$\Pi(\rho_{i,j}) = \Pi(\tau_i)$

$Node_S(\rho_{i,j}) = Node(\tau_i) = Node_S i$

$Node_D(\rho_{i,j}) = Node(\tau_j) = Node_D j$

Function $G$ is surjective. Every element of the codomain is mapped to at least one element of the domain.

$\forall \rho \in \psi; \exists (\tau_i, \tau_j)$ such that $\rho = G((\tau_i, \tau_j))$

You should notice that $G$ depend on the communication protocol used by tasks to exchange their messages.
Table I: Task Model

<table>
<thead>
<tr>
<th>Task</th>
<th>O(s)</th>
<th>T(s)</th>
<th>C(µ(s))</th>
<th>E(s)</th>
<th>H</th>
<th>Node</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₁</td>
<td>1</td>
<td>6</td>
<td>100</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>τ₂</td>
</tr>
<tr>
<td>τ₂</td>
<td>3</td>
<td>2</td>
<td>100</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>τ₃</td>
</tr>
<tr>
<td>τ₃</td>
<td>3</td>
<td>2</td>
<td>300</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>τ₄</td>
</tr>
<tr>
<td>τ₄</td>
<td>5</td>
<td>2</td>
<td>500</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>τ₅</td>
</tr>
<tr>
<td>τ₅</td>
<td>7</td>
<td>2</td>
<td>100</td>
<td>2</td>
<td>1</td>
<td>16</td>
<td>τ₁</td>
</tr>
</tbody>
</table>

Table II: Flow Model computed from the task model

<table>
<thead>
<tr>
<th>Flow</th>
<th>H</th>
<th>NodeS</th>
<th>NodeD</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>ρ₁</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>e(2,1) e(2,1) e(1,5)</td>
</tr>
<tr>
<td>ρ₂</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>e(2,2) e(2,6) e(6,10)</td>
</tr>
<tr>
<td>ρ₃</td>
<td>2</td>
<td>5</td>
<td>16</td>
<td>e(5,8) e(8,12) e(12,16)</td>
</tr>
<tr>
<td>ρ₄</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td>e(10,11) e(11,12) e(12,8)</td>
</tr>
<tr>
<td>ρ₅</td>
<td>1</td>
<td>10</td>
<td>16</td>
<td>e(10,11) e(11,12) e(12,16)</td>
</tr>
<tr>
<td>ρ₆</td>
<td>2</td>
<td>8</td>
<td>16</td>
<td>e(8,12) e(12,16)</td>
</tr>
<tr>
<td>ρ₇</td>
<td>1</td>
<td>16</td>
<td>3</td>
<td>e(16,15) e(15,11) e(11,7) e(7,3)</td>
</tr>
</tbody>
</table>

C. Example of a real-time system over NoCs

The following example illustrates our task model and shows how we can generate the corresponding flow model. For this example, we use a mesh 4 × 4 NoC and the routing algorithm is a XY routing. We consider a real-time application composed of 5 tasks as shown in Figure 2.

![Task Graph Model](image)

Figure 2: Task Graph Model

Table I presents the task model. As shown in Table I, the task τ₁ runs on node 10. Its parameters are priority P = 1, period T = 2 and deadline D = 2. E(τ₃) indicates that τ₃ sends messages to τ₄ and τ₅.

From this task model and with the G function, we can compute the flow model. As shown in Table II, the flow ρ₁ is sent from τ₁ to τ₂, then it is sent from the node 10 to the node 8 because NodeS(ρ₁) = Node(τ₃) = 10 and NodeD(ρ₁) = Node(τ₄) = 8. Then it will use the links e(10,11), e(11,12) and e(12,8).

D. Example of flow parameters for a given task communication protocol

You should notice that Oᵢ, Tᵢ and Dᵢ parameters of flow ρᵢ depend on the communication protocol used by tasks to exchange their messages. Then, to produce those parameters from the task model, the behavior of the task communication protocol has to be considered.

In this paper, we assume immediate data connection protocol which is defined in Architecture Analysis and Design Language (AADL) [8]. This protocol works as follows: tasks send their packets at the end of their execution and the packets are read before the execution of the receiving tasks. We assume the period of the flow is equal to the period of the transmitter task.

Let consider an example of the two tasks τ₁ and τ₂. τ₁ sends a flow ρ₁ to τ₂.

τ₁ = { O₁, T₁, C₁, D₁, Π₁, Node₁, E₁ }
τ₂ = { O₂, T₂, C₂, D₂, Π₂, Node₂, E₂ }
ρ₁ = { O₁, T₁, D₁, Node₁, Node₂, F₁ }

For such flows, the parameters Oᵢ, Dᵢ and Tᵢ can be computed as follows:

Oᵢ = Oᵢ + Cᵢ = O₁ + C₁
Tᵢ = T₁
Dᵢ = Dᵢ - (Cᵢ + Cᵢ + Oᵢ - Oᵢ)

V. IMPLEMENTATION AND EVALUATION OF DTFM

In this section, we first explain the implementation of DTFM. Second, we present its evaluation.

A. Implementation into Cheddar

We implemented DTFM into Cheddar [9]. Cheddar is a GPL framework that provides a scheduling simulator, schedulability tests and various features related to the design and the scheduling analysis of real-time systems. To perform analysis, Cheddar handles an architecture design language, called CheddarADL [10]. CheddarADL allows the users to describe both the software and the hardware parts of the system they expect to analyze. To implement DTFM into Cheddar, we extend CheddarADL to model the NoC and its size, the processor locations in the mesh, usual scheduling parameters, the tasks parameters, the tasks mapping and the dependencies between tasks. From the task set and the NoC model, DTFM produces the associated flow model, checks if deterministic communication delays can be produced by the NoC, and finally, computes those communication delays.

B. Evaluation with a multiscale toolset

We have produced several experiments in order to evaluate the correctness and the scalability of DTFM.

To perform such evaluations, we have implemented and used a multiscale toolset composed of the tick accurate real-time scheduling simulator of Cheddar that simulates the behavior of the task set, and of SHOC [11], a NoC cycle accurate SystemC-TLM simulator that allows us to observe the traffic in the NoC. SHOC supports different types of traffic generators and consumers; In our study we consider ad hoc producers and consumers. SHOC provides all NoC components, and IPs required for MPSoC simulations. All components can be parameterized. In this experiment we have designed and simulated

TABLE III: Evaluation of Path delay

<table>
<thead>
<tr>
<th>U</th>
<th>DP(ns) (SHOC)</th>
<th>DP(ns) (DTFM)</th>
<th>DP(ns) (SHOC)</th>
<th>DP(ns) (DTFM)</th>
<th>DP(ns) (SHOC)</th>
<th>DP(ns) (DTFM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>U = 0.3</td>
<td>300</td>
<td>303</td>
<td>300</td>
<td>303</td>
<td>300</td>
<td>303</td>
</tr>
<tr>
<td>U = 0.5</td>
<td>297</td>
<td>298</td>
<td>290</td>
<td>293</td>
<td>295</td>
<td>293</td>
</tr>
<tr>
<td>U = 0.7</td>
<td>292</td>
<td>293</td>
<td>297</td>
<td>298</td>
<td>295</td>
<td>293</td>
</tr>
</tbody>
</table>

In this section, we focus on different types of delay in the NoC. For each experiment, we compare the results computed by DTFM with the result of SHOC using the same settings.

We first evaluate the path delays computed by DTFM. In this experiment, we use a manual task mapping to enforce that flows generated by DTFM do not share any physical link. The utilization of each processor is set from 0.1 to 0.9 with a step of 0.2. For each processor utilization value, we generate 10 task mappings, and for every configuration we vary the period of tasks from 1000 ns to 1500 ns with a step of 20 ns. To generate each task set, we apply UUniFast [12].

Table III compares the delays produced by SHOC and DTFM for some representative analyzed task sets. We can see that DTFM is able to predict an upper bound of the path delays given by SHOC. Such results are consistent as the variation of processor utilization has no effect in the path delays, which is an expected behavior.

We now evaluate direct delays computed by DTFM. In this new experiment, we use a task mapping with more flows than the previous experiment. Then, some flows may share physical links. In order to study the delays due to direct interferences, a physical link is used at most by two flows. We generate 10 different task mappings with direct interferences. For every configuration, we vary the period of tasks from 1000 ns to 1500 ns with the step of 20 ns. Again, we apply UUniFast.

In this experiment, we change the release time of tasks in order to study different situations of contention between flows. We present in Figure 3 the difference between SHOC simulations and DTFM computations for some of the analyzed task sets. Each curve represents the difference for the same flow with different release times. We can see that DTFM is able to predict a upper bound of the direct delays given by SHOC.

2) Correctness of DTFM evaluations: In this experiment, we verify that any task set analyzed as schedulable by DTFM is actually schedulable.

To verify if a task set is actually schedulable, again, we run SHOC simulations. We vary the period of tasks from 1000 ns to 1500 ns with the step of 20 ns. For each configuration, we varied the task mapping such that the number of flows in the network changes (again, the more we have flows, the more we should have contentions). The number of flows in the network is selected from 3 to 30 with a step of 3.

Figure 4 presents the percent of correct analysis computed by DTFM. We can see that, if there is no indirect delays, any task sets said schedulable by DTFM are also said schedulable by SHOC. We also notice that flows with indirect delays are detected and rejected by DTFM: the corresponding task sets are said unschedulable by DTFM. But in some cases, indirect interference may only add a few extra delays and the system could still be schedulable. In such cases, DTFM
can make too conservative decisions by considering unschedulable cases which are schedulable.

3) Evaluation of the scalability of DTFM: In order to evaluate the scalability of DTFM, we measured the response time of DTFM to perform the analysis of a given task set.

For such evaluations, we call DTFM for several task sets with different numbers of dependencies ranging from 10 to 200. Figure 5 presents the response times of DTFM and shows how much time the tool needs to verify a task set. As shown in this figure, DTFM takes 5 seconds to analyze a task set with 10 dependencies and up to 9 seconds with 200 task dependencies. Cycle-accurate simulation of a few seconds of the system execution can take several hours, while the proposed model decides about the schedulability of the whole system faster with cycle accuracy limited to NOC communications.

VI. CONCLUSION

In this article, we introduced DTFM, a dual task and flow model in order to improve the predictability of real-time applications over NoC architectures. DTFM allows us, from the task model and the task mapping, to check the schedulability of the system. Sharing resources between tasks, contentions of the networks and interferences lead to non-deterministic communication delays which make complex the schedulability analysis of tasks. In DTFM, we take into account all of these factors in order to assess the schedulability of tasks.

We use a standard NoC architecture to illustrate DTFM. Such NoCs are unsafe platforms for real-time applications because they introduce various types of latencies which may prevent tasks to meet their deadlines.

We have implemented DTFM into Cheddar. From a set of tasks and their dependencies, we can automatically check and compute the various latencies in the NoC and perform schedulability analysis. Furthermore, when a task set leads to flows with indirect delays, such flows are detected by DTFM and the task set is said unschedulable. Besides DTFM and its corresponding tool, a second contribution is the tool-based validation approach which is based on a multiscale simulator toolset. The validation tool is composed of the Cheddar real-time scheduling simulator and SHOC, a SystemC NoC simulator. The task scheduling produced by Cheddar is used as an input of the SystemC simulator, which is used to measure delays in the NoC.

Such multiscale toolset simulator could be applied to investigate the temporal behavior of real-time applications running on various other hardware components such as cache hierarchies. With this multiscale simulator, we made experiments that evaluate DTFM correctness and scalability.

In future work, we will enrich DTFM with new delay models for different and usual task communication protocols and NoCs architectures. We also intend to use DTFM and its associated tools to investigate task mapping and arbitration techniques.

VII. ACKNOWLEDGEMENT

The authors would like to thank Dr Martha Johanna Sepúlveda for her help on the SHOC simulator. This work and Cheddar are supported by BMO, Ellidiss Technologies, CR de Bretagne, CG du Finistère, and Campus France.

REFERENCES

Go-RealTime: A Lightweight Framework for Multiprocessor Real-Time System in User Space

Zhou Fang\textsuperscript{1}, Mulong Luo\textsuperscript{1}, Fatima M. Anwar\textsuperscript{2}, Hao Zhuang\textsuperscript{1}, and Rajesh K. Gupta\textsuperscript{1}

\textsuperscript{1}Department of Computer Science and Engineering, UCSD
\textsuperscript{2}Department of Electrical Engineering, UCLA

Abstract—We present the design of Go-RealTime, a lightweight framework for real-time parallel programming based on Go language. Go-RealTime is implemented in user space with portability and efficiency as important design goals. It takes the advantage of Go language’s ease of programming and natural model of concurrency. Goroutines are adapted to provide scheduling for real-time tasks, with resource reservation enabled by exposing Linux APIs to Go. We demonstrate nearly full utilization on 32 processors scheduling periodic heavy tasks using Least Laxity First (LLF) algorithm. With its abstraction and system support, Go-RealTime greatly simplifies the set up of sequential and parallel real-time programs on multiprocessor systems.

Index Terms—Real-Time Systems; Multiprocessor Systems; Parallel Programming; Go Language; Scheduling Algorithms

I. INTRODUCTION

Multiprocessing is now the primary means of improving performance on diverse platforms from embedded systems to data-centers. Many frameworks and the associated tools make parallel programming easier, such as compiler support (OpenMP \cite{1}), language extension (Intel Cilk Plus \cite{2}) and library (Intel Thread Building Block (TBB) \cite{3}). These tools provide easy-to-use programming support to construct parallel tasks, and orchestrate execution details behind the scene. Yet, the goal of minimizing execution time is only partially achieved due to the lack of resource reservation. This is particularly limiting for real-time (RT) applications.

The majority of prior work on real-time systems is focused on operating system (OS) support for scheduling of sequential tasks on uniprocessor models. Multiprocessor systems are attracting increasing research interest, which have more complex schedulers. Among notable multiprocessor system implementations, Calandrino \textit{et. al.} proposed LITMUS\textsuperscript{RT} \cite{4}, a Linux based testbed for real-time multiprocessor schedulers. Based on LITMUS\textsuperscript{RT}, several different implementations of global Earliest Deadline First (EDF) algorithm \cite{5} were tested and compared \cite{6}. The drawback of global EDF algorithm on multiprocessor system is that schedulability of heavy utilization task is degraded dramatically \cite{7}. Dynamic priority algorithms, such as Least Laxity First (LLF) \cite{8,9}, and P-fairness \cite{10}, are able to achieve higher utilization than EDF on multiprocessor systems. However, they induce larger system cost due to frequent context switching and cache pollution.

Prior to scheduling, because there are many choices to decompose a parallel task into a group of sequential sub-tasks, parallel task scheduling algorithms have additional complexity on top of sequential scheduling. Lakshmanan \textit{et. al.} \cite{11} studied fork-join tasks and proposed the task stretch transform to reduce scheduling penalty. For a general class of parallel tasks, Saifullah \textit{et. al.} \cite{12} proposed to decompose a parallel task into several sequential sub-tasks with carefully computed deadlines, which are then scheduled using global EDF. This algorithm was then implemented by Ferry \textit{et. al.} \cite{13}. Due to lack of a direct API to make a thread switch, in \cite{13} it indirectly switches threads by changing their priority. This indirect approach results in a complex implementation and limits the performance.

In this work, we propose a new RT framework, Go-RealTime to solve the above challenges using Go language \cite{14}. It supports both real-time scheduling and parallel programming with a set of APIs. Go provides native support for concurrency by means of goroutines. We leverage this advantage of Go for real-time programs running on concurrency units in user space. OS thread scheduler is integrated into Go runtime and adopted for resource reservation. In Go-RealTime, we import OS APIs (thread scheduler, CPU affinity and timer) into Go, modify Go runtime to enable direct goroutine switch, and build external Go packages to support asynchronous events, parallel programming and real-time scheduling. Go-RealTime is able to make a given portion of processors (or a fraction of one processor) real-time and keep the rest unchanged, thus it supports implementation of mixed criticality systems. This approach is also safer and more portable than earlier works \cite{4,6} because no modifications to the OS kernel are needed.

In Go-RealTime, a user can program a parallel task as Directed Acyclic Graph (DAG) \cite{15} of sub-tasks. Sub-tasks are executed by subroutines so the overhead of process-level concurrency is saved. It supports multiple scheduling algorithms for different tasks running at the same time. Different types of tasks are scheduled by the suitable algorithm (EDF \cite{5}, LLF \cite{8}). The total processor resources are partitioned dynamically among all schedulers by the resource balancing mechanism. Implementation results show that our framework is lightweight, flexible and efficient to deploy real-time programs on platforms ranging from embedded devices to large-scale servers.

The main contributions of our work are: (1) we design a real-time parallel framework using Go language without modifying OS kernel; (2) we present the resource balancing approach to execute multiple scheduling algorithms in parallel; (3) we implement system support for handling asynchronous event in user space; (4) we implement a multiprocessor LLF scheduler with a simple tie breaking method, which can achieve near full utilization for randomly generated heavy task set $(0.5 \leq \mu_{task} \leq 0.9)$ on a 32-processor machine; (5) we evaluate the implementation of EDF and LLF in Go-RealTime through analyzing system overhead and taking schedulability tests.

The rest of the paper is organized as follows. We introduce the system architecture of Go-RealTime in Section II. It presents Go-RealTime’s APIs, system model, resource balancing, and asynchronous event handling. Programming parallel task in the framework is presented in Section III. The design, implementation and evaluation of real-time schedulers are presented in Section IV. Section V gives the conclusion and future works.

II. THE GO-REALTIME ARCHITECTURE

In this section we introduce the overall system architecture, APIs and data structure of Go-RealTime. It is currently implemented on top of Linux version of Go 1.4. Go-RealTime works concurrently with the original runtime of Go language. It creates RT goroutines which run real-time tasks. The threads carrying RT goroutines are separated from native Go threads (no goroutine migration). In the following discussion, threads and goroutines are created by Go-RealTime unless otherwise stated.
Go-RealTime is targeting for applications which require millisecond level timing precision. The requirement is given by $\delta_{async}$, the upper bound of tolerable asynchronous event timing uncertainty. A real-time asynchronous event handler is designed to meet the requirement dynamically. The framework turns off Go’s garbage collector to reduce timing uncertainty. In the current implementation, memory usage is managed by recycling dynamically allocated objects. Replacing the Go garbage collector by a more controllable approach which meets real-time requirement lies in the future work.

A. Go-RealTime’s APIs

Go language’s concurrency is enabled through goroutines and invoked with keyword go. A user creates a goroutine and associates it with a program using go func(arg). After creation, go runtime scheduler automatically allocates goroutines to run on OS threads. Each thread created by the Go runtime keeps a runnable queue of goroutines. In the original Go runtime, a simple scheduling algorithm is adopted: each thread keeps a runnable queue in First-In-First-Out (FIFO) order. It tries to make a goroutine switch per 10ms. User can yield the execution of a goroutine using the API GoSched(). Compared to Linux’s Completely Fair Scheduler (CFS), this design is more lightweight and scalable. It is also much easier to use than thread libraries such as Pthread. However, the Go runtime completely hides system details from the programmer, thus making it difficult to carry out RT task scheduling due to lack of control over threads and processors. Go-RealTime modifies Go by adding resource and scheduling APIs (Table I). These include:

- SetSched: use Linux system call sched_setscheduler to change the scheduling policy of a Go thread from time-sharing (CFS as default) to RT (FIFO, Round-Robin).
- BindCPU: bind a Go thread to a processor using sched_setaffinity system call. Then Go thread runs exclusively on the processor.
- GoSwitch: directly switch to a specific target goroutine. It is implemented by modifying Go runtime source code.

These modifications and additions make the execution of goroutines fully controllable in user space. The APIs can reserve a fraction $f$ of one processor as well: two Linux timers are used to set Go thread RT at $t_{rt}$ and set it back to time-sharing at $t_s$. Given the period of timers $t_{timer}$, it satisfies $t_{ts} - t_{rt} = f \cdot t_{timer}$, so the Go thread occupies fraction $f$ of the processor deterministically. It is useful to run RT tasks on uniprocessor systems. In this work we focus on taking $N_{cpu}$ processors completely for RT tasks.

Go-RealTime relies on these APIs to implement RT schedulers in Section IV. As an alternative to directly using these APIs, the users can also use the easier Task APIs (Table I), which assign the timing specifications and submit tasks to the RT scheduler.

B. Go-RealTime’s System Model

The system model of Go-RealTime consists of three objects: worker, task and goroutine, as shown in Figure 1. A worker is a thread created by Go-RealTime. It is an abstraction of reserved processor resource. A Go-RealTime program creates a group of workers via NewWorker($N_w$) method. The number of workers ($N_w$) is usually equal to $N_{cpu}$ to maximize parallelism.

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>SetSched(thread, policy, priority)</td>
<td>Linux thread scheduler processor affinity</td>
</tr>
<tr>
<td></td>
<td>BindCPU()</td>
<td>processor affinity</td>
</tr>
<tr>
<td>Scheduling</td>
<td>SetTimerFd(file description, time)</td>
<td>Linux timer</td>
</tr>
<tr>
<td>Task</td>
<td>GoSwitch(goroutine)</td>
<td>switch to a goroutine</td>
</tr>
<tr>
<td></td>
<td>NewWorker($N_w$)</td>
<td>create a RT worker</td>
</tr>
<tr>
<td></td>
<td>task.SetTimeSpec(t_s, t_p, t_d, t_r)</td>
<td>set timing specification</td>
</tr>
<tr>
<td></td>
<td>task.Run(func, arg)</td>
<td>start a RT task</td>
</tr>
</tbody>
</table>

Fig. 1. System model of Go-RealTime. A worker is the abstraction of a thread with one processor resource reservation. Goroutines are sorted in a few priority queues. A parallel task is decomposed into DAG of sequential sub-tasks.

Task is the object to describe a RT program. It has the following features:

- Time specification: starting time $t_s$, period $t_p$, deadline $t_d$, and worst case running budget $t_r$, set by task.SetTimeSpec($t_s$, $t_p$, $t_d$, $t_r$).
- Utilization: $\mu_{task} = t_r/t_p$. ($\mu_{task}$ can be larger than 1 for parallel tasks).

Goroutine is the concurrency unit. A new RT goroutine is created by calling task.Run(func, arg). task then starts to run on the new goroutine. A group of workers shares some runnable queues of goroutines. $N_{queue}$, the number of queues, is a parameter of Go-RealTime scheduler. Priority queue is used to sort goroutines by a key ("deadline" in EDF, "laxity" in LLF). We use the lock free priority queue data structure [16] to reduce the cost of parallel access contention.

Workers fetch the head goroutine in queue to execute. When a worker becomes idle, it fetches the first goroutine in queue. The scheduler is preemptive: if a goroutine $g_0$ becomes runnable and its priority is higher than $g_1$ which is running, the scheduler switches off $g_1$ and let $g_0$ run. A sequential task is executed by one goroutine. A parallel task is decomposed into sequential sub-tasks described by DAG model, executed by a group of goroutines. Since no preemption happens among sub-tasks of the same parallel task, they are implemented simply as subroutines.

C. Resource Balancing in Go-RealTime

Go-RealTime is able to keep more than one goroutine queue to support multiple scheduling algorithms in parallel. This approach has two benefits: (1) it supports partitioned scheduling policy, and different algorithms can be deployed for different sets of tasks, (2) scheduling algorithm can be changed on-the-fly without interrupting running tasks.

In an example with two queues is shown in Figure 2, one queue uses EDF algorithm, and the other uses LLF. Load imbalance happens when the total task utilization of EDF and LLF queues does not match their allocated processor resources, which induces utilization loss. Load balancing technique such as work stealing [17] is a solution. Whereas for RT scheduling, work stealing becomes complex because it must respect task priority during stealing. Otherwise it may run lower priority tasks but leave higher priority tasks waiting. Go-RealTime uses resource balancing instead of load balancing. Because all tasks are periodic and their utilization $\mu_{task}$ are known, total utilization of a queue is simply the sum $\mu_{queue} = \sum \mu_{task}$. It is the amount of processor resource the queue should get.

The total resources, $N_w$ workers, are allocated to the queues proportionally according to $\mu_{queue}$.

When the load of one queue ($\mu_{queue}$) changes, the number of workers allocated to each queue ($N_{queue}$) is recomputed. Go-RealTime dynamically assigns workers to queues according to $N_{queue}$. In most cases, $N_{queue}$ is not an integer, so a queue may be assigned a fraction of a worker. In this example (Figure 2), worker 1 is allocated to both queues. A fraction $f_{EDF}$ of this worker should be allocated for EDF queue, and $f_{LLF}$ for LLF queue ($f_{EDF} + f_{LLF} = 1$). Worker 1 tries to give the right fractions of total running time to both
queues. It records the total running time of each queue. When it becomes idle and is going to grab a goroutine, it firstly checks the current fraction of total running time of each queue, and selects the queue which has received the smallest fraction. Go-RealTime asynchronously checks and enforces resource balancing while goroutine is running, repeated by a given time interval $t_{balance}$. The implementation will be explained in the next section.

Fig. 2. Resource balancing: allocating the correct fraction of the workers to goroutine queues.

D. Handling of Asynchronous Events

Go-RealTime handles asynchronous events via check_async method. It checks all asynchronous events (timeout, message, etc.) and responds to the events which should have occurred. The challenge is how to call check_async method repeatedly in an asynchronous manner. Our current implementation is to insert check_async calls in the program until the interval between two consecutive checks is less than $\delta_{async}$. By skipping some of these checks, we can control the timing precision. Algorithm 1 gives an example of instrumenting a ConvertGray function [18] with check_async. The function converts a colored image in memory to gray scale. The three locations to insert check_async ($@out$, $@lpy$, $@lpx$) result in different timing precisions and overheads. To tune precision in a finer way, it can skip a fixed number of check_async calls using a counter. For example, it can call check_async once every ten times when the program runs to location $lpy$ (denote as "lpy, 1/10").

Algorithm 1 Instrumented ConvertGray

```plaintext
1: function CONVERTGRAY
2: @out: check_async()
3: for Loop over Y coordinate do
4: @lpy: check_async()
5: for Loop over X coordinate do
6: @lpx: check_async()
7: SetPixelGray(image, x, y)
8: end for
9: end for
10: end function
```

The check_async facility is used in a number of components in Go-RealTime as discussed below.

1) Timer: Go-RealTime implements a software timer upon check_async. It keeps a priority queue of active timers at each worker. The queue uses “time” value to assign priorities. The current running goroutine on a worker is responsible for checking the timers. check_timer method gets called inside check_async. It compares the head timer in queue with the current clock time. If any timeout is due, it calls the handler function of the timer and removes the timer from the queue. We use timer uncertainty to evaluate timing precision of the framework. Timer uncertainty is computed as $\Delta_{timer} = t_{notify} - t_{expect}$. $t_{expect}$ is the expected timeout of timer. $t_{notify}$ is when the program is notified and responds to the timer.

We compare timers of Linux (notify via file descriptor), Go and Go-RealTime. $\Delta_{timer}$ is measured by repeatedly setting a timer and computing $t_{notify} - t_{expect}$. All tests are running on real-time threads on a server with 32 Intel Xeon E5-2640 2.6GHz CPUs running CentOS 6.6. The statistics of $\Delta_{timer}$ is shown in Figure 3. The $x$-coordinate is the value of $\Delta_{timer}$ and the $y$-coordinate gives the cumulative probability function. The overhead of Go-RealTime timer is given near the curve. The results show that the uncertainty of the Linux timer is around $100\mu$s. The native Go timer daemon is running on a goroutine, which may be influenced by other goroutines. In order to test its performance under load, we create a few dummy goroutines running on the same thread as the timer goroutine (load=1 means one dummy goroutine). We see that 10 goroutines increase the uncertainty to sub-second level, which means the native Go timer can not guarantee its precision. The precision of Go-RealTime timer depends on program details (check_async location). We use the program in Figure 1 for the test. Because all Go-RealTime goroutines are checking the timers generated by other goroutines as well, loading goroutines do not influence the timer precision. The result shows check_async method is able to achieve millisecond-level precision with small overhead ($2.9ms/s$ for $lpy$, $3.3ms/s$ for $lpy$ 1/10). Go-RealTime timer does not rely on OS timer. Compared to the Linux timer, it provides the ability to tune precision and overhead in a large range.

Fig. 3. Timer uncertainty comparison of Linux, Go and Go-RealTime. Each test contains 500 samples.

2) Messages: Passing asynchronous messages and getting on-time responses is important in RT systems. In Go-RealTime, each worker has a message queue to be updated by asynchronous events. The current running goroutine checks the messages when check_async runs. In this way the running goroutine can respond to asynchronous messages quickly. As an instance, for EDF and LLF scheduling, an asynchronous message is sent to the running task when a new task becomes runnable. Upon receiving the message, the running task compares its priority with the new one, then the task with higher priority continues to run.

3) Other Events: Go-RealTime implements a few other asynchronous events based on check_async. Two representative examples are (i) checking resource balancing is done in check_async method every $t_{balance}$, and (ii) in LLF scheduling, all goroutines actively update and compare laxity in check_async method every $t_{lif}$. III. PARALLEL TASKS IN GO-REALTIME

Go language provides goroutine and channel constructs [14] to build scalable concurrent programs. Parallel programming in Go is studied in [19] using a dynamic programming example that we will use to illustrate the advantages of Go-RealTime. The example addresses a search method: given $N$ keys and the probability of each key, find the optimal binary search tree that minimize average search time. The graph of computing nodes is shown in Figure 5 (a). It starts from the diagonal nodes and ends at the right upper corner node. The program is parallelized by grouping nodes as sub-task as Figure 5 (b). In the reference implementation in [19], each sub-task is executed as a goroutine. Because there is a fixed dependency among
all sub-tasks, there is no need for each sub-task to exclusively occupy a concurrency unit, which costs resource and increases switch time. Executing each sub-task as a subroutine is more efficient.

Go-RealTime models a parallel task as a DAG of sub-tasks. It stores runnable sub-tasks in a global pool protected by a spin lock. A sub-task is pushed into the runnable pool after all its predecessors have completed. When a goroutine is idle, it fetches a sub-task to execute in the pool of its associated parallel task. A goroutine sleeps if the pool is empty and is woken up when new sub-tasks are runnable. Go-RealTime creates a group of goroutines for a parallel task. It includes an initializing goroutine, a finalizing goroutine, and a few goroutines to execute the parallel section, constructed in the fork-join pattern, as shown by Figure 4. A goroutine is blocked until all its predecessors finish. The initializing goroutine is responsible for preparing for parallel computation, such as loading data and initializing sub-task pool. The finalizing goroutine does clearing work and handles computation result. The API to construct a parallel task is the subtask object: users create a set of sub-tasks, assign each sub-task a sequential program to run and a list of predecessor/follower in the DAG.

Fig. 4. In this example, a parallel reduction task is decomposed into a DAG of sequential sub-tasks. It is executed by a group of goroutines constructed in the fork-join pattern.

The topology of a DAG is decided by the decomposition of a parallel task. A finer grain of decomposition leads to better parallelization. But it also induces larger system overhead of sub-task switching and contention at spin lock. For the example in Figure 5, we denote the number of sub-tasks on the diagonal as \( N_p \), the total number of sub-tasks is \( N_p \cdot (N_p + 1) / 2 \). Figure 5 (b) illustrates that \( N_p = 8 \) leads to better utilization of parallel resource than \( N_p = 4 \). We test the program on 16 processors using different values of \( N_p \) with \( N_{seq} = 1024 \). As shown in Figure 6 (a), when \( N_p = 16 \) the parallel resource can not be effectively utilized. When \( N_p \) becomes 128, as Figure 6 (b) shows, the parallel resource is well utilized. The span is largely reduced by around 4 times. User should decide the most suitable granularity considering both span and cost. The scheduling problem of a set of finely parallelized tasks, for which sequential section are ignorable, can be simplified as uniprocessor scheduling problem, thus global EDF allows us to achieve optimal scheduler implementation.

IV. REAL-TIME SCHEDULER IN GO-REALTIME

Our goal was to design a RT scheduler for Go-RealTime that manages both sequential and parallel tasks concurrently. With the help of the resource balancing design, several different scheduling algorithms can work in parallel with correct portions of processors. To simplify the implementation of parallel scheduling, Go-RealTime uses EDF to schedule all parallel tasks. It is optimal for finely parallelized programs. In this section, we focus on the sequential scheduling problem. We present our EDF/LLF implementation, and demonstrate the gains due to on-the-fly scheduling algorithm change in Go-RealTime.

A. Scheduling Algorithms

EDF is a fixed priority scheduling algorithm. Given a set of \( N_{task} \) tasks, the total times of switching is at most \( N_{task} \).

However, for multiprocessor system, a single sequential task can not utilize parallel resources. Keeping a high number of concurrent runnable tasks is important to better utilize parallel resources. LLF uses laxity instead of deadline as the metric of priority. Laxity \( t_{laxity} \) is the latest starting time to meet deadline, calculated by \( t_{laxity} = t_d - t_e + t_e \) (This follows the same definition in [9]). \( t_e \) is the time that a task has already been executed. LLF scheduler uses the worst case of \( t_e \) given by user. When a task is running, its laxity increases. Laxity ties happen when two tasks have the same laxity and repeatedly switch to each other. Theoretically LLF may induce infinite task switching because of tie. Previous work on LLF proposed several strategies to break laxity ties and bound switching times [9], but they are designed for single processor and do not utilize parallel resources. Go-RealTime breaks laxity ties in a simple way: it compares laxities of the running task with the head task in queue in check async method. A switch is allowed only after \( t_{llf} \) since the previous. Therefore \( t_{llf} \) is the minimal interval of task switching caused by LLF. \( t_{llf} = 10\text{ms} \) is used by default.

We test the EDF and LLF algorithms using 4 and 32 processors, respectively. The test program is a simple infinite loop. It checks timer queue every \( 1\text{ms} \). Period and utilization of tasks are randomly generated from an uniform distribution. Period \( t_p \) ranges in \( 100\text{ms} \leq t_p \leq 300\text{ms} \). Two types of tasks are considered: light task \((0.1 \leq \mu_{task} \leq 0.5)\) and heavy task \((0.5 \leq \mu_{task} \leq 0.9)\). In the test we run each set of tasks for \( 10\text{s} \), which has a random total utilization \( \mu_{set} \). We calculate the ratio of successfully scheduled sets (sets without deadline miss) \( r_{scheduled} \) for small ranges of \( \mu_{task} \). The result is given in Figure 7. It shows that for sets of light tasks (Figure 7 (a)(c)), EDF and LLF can achieve high utilization close to upper bound, on both 4 and 32 processors. Performance of EDF is slightly worse than LLF. For heavy tasks, LLF still achieves high utilization, whereas a large ratio of task sets is not schedulable using EDF (Figure 7 (b)(d)). The result confirms with the bad performance of EDF for high utilization tasks.

Fig. 5. Parallel programming example. (a) dynamic programming grids are group into boxes as sub-tasks, (b) DAG of sub-tasks and (c) impact of decomposition granularity.

Fig. 6. Parallelize dynamic programming on 16 processors with \( N_p \) as (a) \( N_p = 16 \) and (b) \( N_p = 128 \).
scenarios. However, frequent goroutine switching induced by LLF may incur large system overhead. Next we quantify the overhead to comprehensively understand the performance of schedulers.

![Image](image-url)

**Fig. 7.** Real-Time scheduler performance. The tests for EDF on heavy tasks contain 400 samples. The other tests contain 100 samples each. (a) 4-processor, light tasks, (b) 4-processor, heavy tasks, (c) 32-processor, light tasks and (d) 32-processor, heavy tasks.

**B. System Cost**

System cost is an important consideration for the timing overhead it represents. The cost includes direct and indirect components. Direct cost consists of time consumed by the framework code and goroutine switch. Indirect cost that we consider is mainly cache pollution.

1) **Direct Cost:** We classify direct cost into three sources: queueing, switching and timer. Queueing cost is the time consumed by scheduler code, the majority comes from the cost of *insert* and *fetch* operations on goroutine queues. Increasing number of concurrent tasks leads to higher queueing cost. Shorter period of tasks also increases the queueing cost. This is because goroutine queueing operation is more frequent, which results in intense contention of parallel access. Switching cost is the direct cost of goroutine switch. The overhead of timer is mainly induced by *check_timer* call, decided by its calling interval. It also includes the cost of operations on timer queue such as insertion.

The histogram of direct cost on 32-processor tests is given in Figure 8. Beside light and heavy tasks, we consider ultra light tasks which has $0.01 \leq \mu_{\text{task}} \leq 0.1$, to study the queueing cost with a large number of tasks in queue. The result shows that direct cost is still small when 500 ultra light tasks run on 32 processors (smaller than 0.055%, shown in (a)). As the number of tasks reduces and period increases (like most application cases), the direct cost becomes even smaller and can be ignored.

2) **Cache Pollution:** When goroutine switch happens, the current cache status may be invalid due to context change of data and code. The time required to update cache from memory is called cache pollution. The cost highly depends on programs and machines, thus it is hard to be exactly quantified. The cost reported by [20] is in the range of milliseconds. Therefore it should be considered as a major cost. We use the number of goroutine switch as the metric of indirect cost. Figure 9 gives the histogram of average number of goroutine switches per second-processor. It shows the switch induced by LLF is evidently larger than EDF. Because of similar performance in scheduling ultra light and light tasks, EDF is preferred in these cases to reduce switching cost.

**C. Schedulability Evaluation**

We give detailed schedulability evaluation of the Go-RealTime for EDF and LLF. We use the convention in [21], [22], [23], [24] to generate the test sets. That is, starting with zero task in the task set, we randomly generate tasks with uniformly distributed period and utilization, then add this task to the current task set. We check the total utilization of this task set, if it is less than the lower bound of the predetermined utilization, we continue to add new task into this task set until the total utilization is in the range, then this task set is tagged as valid for schedulability test. If the total utilization exceeds the upper bound, we abandon this task set and start with a new task set with zero task. For the total utilization of each task set, we set it to be within 2.0 to 3.9. The period of each task is uniformly distributed between 100ms to 300ms. For light tasks, the utilization per task is uniformly distributed between 0.1 to 0.5, for heavy tasks the utilization per task is uniformly distributed between 0.1 to 0.9. We generate 5000 task sets for each case of testing.

Figure 10 shows the results of schedulability experiments. To be specific, Figure 10(a) shows the number of schedulable task sets with only light tasks. For light tasks, the number of tasks for each total utilization is almost uniformly distributed. We can see that for both EDF and LLF algorithms, all the task sets are schedulable. Figure 10(b) shows the number of schedulable task sets with heavy tasks. For heavy tasks, the distribution of tasks under different total utilization is not uniform. It first increases with the total utilization then decreases with it. We see that for EDF, the number of successful task sets drops quickly with the increase of total utilization. While for LLF, the number of successful task sets decreases slowly. This implies that LLF performs better than EDF for this settings in Go-RealTime.

**D. On-the-fly Scheduling Algorithm Change**

The support for multiple schedulers in Go-RealTime is the key to combine advantages of different algorithms, such as
check_async method based on timing profile of programs, (2) design a controllable approach to run garbage collection which respects the priority of RT tasks.

REFERENCES


V. CONCLUSIONS AND FUTURE WORK

In this paper, we present the design and implementation of Go-RealTime, a real-time parallel programming framework implemented in user space using Go programming language. Important design choices related to resource reservation, asynchronous execution, and runtime coordination make it possible for the application developer to embed strong timing requirements and ensure their satisfaction through a flexible runtime scheduler. It also supports DAG-based parallel programming to deploy parallel program on multiprocessor system.

Go-RealTime is implemented by modifying open source Go runtime, implementing Go packages and by importing important Linux system calls into Go. It uses Linux thread scheduler for resource reservation. Goroutine running on top of thread is the unit of concurrency. Our framework greatly simplifies the implementation and deployment of RT programs, and improves the flexibility in system extension.

Our prototype of Go-RealTime is moving forward in the following directions: (1) design a strategy to automatically place
Author Index

Afanasov, Mikhail 19
Astrinaki, Maria 13

Becker, Matthias 1
Belham, Moris 1
Bini, Enrico 31

Christoforakis, Ioanins 13

Dasari, Dakshina 1
Diguet, Jean-Philippe 43
Domínguez-Poblete, Jorge 7
Dridi, Mourad 43

Eker, Johan 31

Fang, Zhou 37

García-Valls, Marisol 7
Gupta, Rajesh 37

Iavorskii, Aleksandr 19

Kornaros, Georgios 13

Leva, Alberto 25
Luo, Mulong 37

M. Anwar, Fatima 37
Millnert, Victor 31
Mottola, Luca 19
Mubeen, Saad 1

Nolte, Thomas 1

Papadopoulos, Alessandro Vittorio 25
Prandini, Maria 25

Rubini, Stéphane 43

Singhoff, Frank 43

Terraneo, Federico 25
Touahria, Imad Eddine 7
Zhuang, Hao 37
### Keyword Index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptive software</td>
<td>19</td>
</tr>
<tr>
<td>architectural support for QoS</td>
<td>13</td>
</tr>
<tr>
<td>Automotive</td>
<td>1</td>
</tr>
<tr>
<td>Cheddar</td>
<td>43</td>
</tr>
<tr>
<td>Clock Synchronization</td>
<td>25</td>
</tr>
<tr>
<td>Cloud computing</td>
<td>31</td>
</tr>
<tr>
<td>Communication delay</td>
<td>43</td>
</tr>
<tr>
<td>context oriented</td>
<td>19</td>
</tr>
<tr>
<td>Control Theory</td>
<td>25</td>
</tr>
<tr>
<td>cyber-physical systems</td>
<td>19</td>
</tr>
<tr>
<td>Data Age</td>
<td>1</td>
</tr>
<tr>
<td>DDS</td>
<td>7</td>
</tr>
<tr>
<td>Design Phase</td>
<td>1</td>
</tr>
<tr>
<td>End-to-End</td>
<td>1</td>
</tr>
<tr>
<td>Go Language</td>
<td>37</td>
</tr>
<tr>
<td>guaranteeing on-chip throughput</td>
<td>13</td>
</tr>
<tr>
<td>memory bandwidth shaping</td>
<td>13</td>
</tr>
<tr>
<td>Middleware</td>
<td>7</td>
</tr>
<tr>
<td>mixed-critical embedded device</td>
<td>13</td>
</tr>
<tr>
<td>Multiprocessor Systems</td>
<td>37</td>
</tr>
<tr>
<td>Network function virtualization (NFV)</td>
<td>31</td>
</tr>
<tr>
<td>Network-On-Chip</td>
<td>43</td>
</tr>
<tr>
<td>Parallel Programming</td>
<td>37</td>
</tr>
<tr>
<td>Partitioned systems</td>
<td>7</td>
</tr>
<tr>
<td>programming paradigm</td>
<td>19</td>
</tr>
<tr>
<td>Publish-subscribe</td>
<td>7</td>
</tr>
<tr>
<td>Real-time</td>
<td>7</td>
</tr>
<tr>
<td>Real-Time Systems</td>
<td>37</td>
</tr>
<tr>
<td>Real-time systems</td>
<td>43</td>
</tr>
<tr>
<td>resource management</td>
<td>13</td>
</tr>
<tr>
<td>Scheduling Algorithms</td>
<td>37</td>
</tr>
<tr>
<td>service chain</td>
<td>31</td>
</tr>
<tr>
<td>Switched control</td>
<td>25</td>
</tr>
<tr>
<td>SystemC NoC simulator</td>
<td>43</td>
</tr>
<tr>
<td>Keyword</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>task model</td>
<td>43</td>
</tr>
<tr>
<td>time-sensitive adaptation</td>
<td>19</td>
</tr>
<tr>
<td>traffic-flow model</td>
<td>43</td>
</tr>
<tr>
<td>Virtual network function(s) (VNF)</td>
<td>31</td>
</tr>
<tr>
<td>Wireless Sensor Network</td>
<td>25</td>
</tr>
</tbody>
</table>