UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Control Performance Optimization for Application Integration on Automotive Architectures

Minaeva, A.; Roy, D.; Akesson, B.; Hanzálek, Z.; Chakraborty, S.

DOI
10.1109/TC.2020.3003083

Publication date
2021

Document Version
Author accepted manuscript

Published in
IEEE Transactions on Computers

Link to publication

Citation for published version (APA):
Minaeva, A., Roy, D., Akesson, B., Hanzálek, Z., & Chakraborty, S. (2021). Control Performance Optimization for Application Integration on Automotive Architectures. *IEEE Transactions on Computers*, *70*(7), 1059-1073. https://doi.org/10.1109/TC.2020.3003083

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:28 Aug 2022

Control Performance Optimization for Application Integration on Automotive Architectures

Anna Minaeva[1,2], Debayan Roy[3], Benny Akesson[4,5], Zdeněk Hanzálek[1], Samarjit Chakraborty[6]
[1]Czech Institute of Informatics, Robotics, and Cybernetics, Czech Technical University in Prague
[2]Faculty of Electrical Engineering, Czech Technical University in Prague
[3]Institute for Real-Time Computer Systems, Technical University of Munich, Germany
[4]ESI (TNO), Eindhoven, The Netherlands
[5]University of Amsterdam, The Netherlands
[6]Department of Computer Science, The University of North Carolina at Chapel Hill, USA

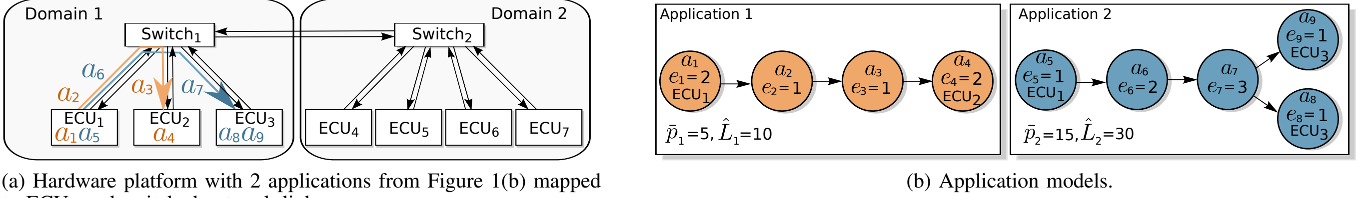
Abstract*—Automotive software implements different functionalities as multiple control applications sharing common platform resources. Although such applications are often developed independently, the control performance of the resulting system depends on how these applications are integrated. A key integration challenge is to efficiently schedule these applications on shared resources with minimal control performance degradation. We formulate this problem as that of scheduling multiple distributed periodic control tasks that communicate via messages with non-zero jitter. The optimization criterion used is a piecewise linear representation of the control performance degradation as a function of the end-to-end latency of the application. The three main contributions of this article are: 1) a constraint programming (CP) formulation to solve this integration problem optimally on time-triggered architectures, 2) an efficient heuristic called *Flexi*, and 3) an experimental evaluation of the scalability and efficiency of the proposed approaches. In contrast to the CP formulation, which for many real-life problems might have unacceptably long running times, *Flexi* returns nearly optimal results (0.5% loss in control performance compared to optimal) for most problems with more acceptable running times.

I. INTRODUCTION

With increasing user demands, the complexity of modern automotive systems is growing rapidly. These functionalities are often implemented as subsystems from different suppliers, which are integrated by the original equipment manufacturer (OEM). Some of them are implemented as control applications, such as engine management systems, steer-by-wire, autonomous cruise control, or pre-crash safety systems [20], [21]. A control application typically involves a sequence of sensing, computation, and actuation tasks, where sensor values and control signals need to be transmitted over an in-vehicle communication network comprising CAN, FlexRay and Ethernet.

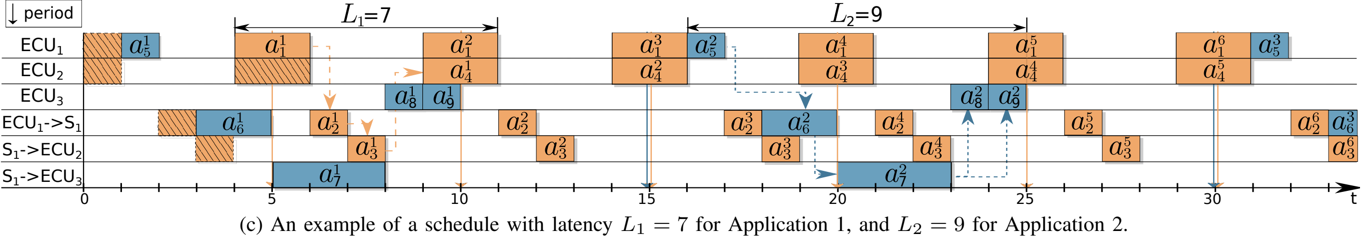
To reduce cost, automotive applications typically share platform resources on which they are executed, e.g., ECUs and network links [36]. Therefore, car manufacturers face the problem of integrating subsystems from different suppliers comprising one or multiple applications, such that their control and timing requirements are guaranteed [22], [24], [39]. The control performance of an application depends strongly on the timing behavior of its implementation on the shared platform [9], [10]. The relationship between control performance and the timing behavior the control application experiences heavily depends on the application [12]. This needs to be accounted for during the application integration stage. Towards this, the problem addressed in this article is *to schedule multiple control applications on a shared platform, while minimizing the maximum control performance degradation among them*. This has to be done while satisfying hard real-time constraints, such as deadlines, jitter, data dependencies, and end-to-end latency constraints. For this purpose, we consider time-triggered scheduling [18], in which distinct *activities* (viz., tasks and messages) are executed at predefined times on different resources. This problem turns out to be computationally hard [7]. However, given its practical relevance, it is important to get a good quality solution within a reasonable time. Efficient solutions to this application integration problem will also have use in setups where applications are dynamically downloaded from the cloud and deployed at runtime [28], [38].

The three main contributions of this article are: 1) A constraint programming (CP) formulation that solves the problem optimally and exploits properties of the problem to reduce the computation time. 2) A heuristic called *Flexi* that first constructs a schedule and then improves it via a large neighborhood search. It reduces the problem complexity by introducing a coarser scheduling granularity where possible. This approach provides a reasonable trade-off between computation time and solution quality compared to the optimal approach. 3) An experimental evaluation of the proposed approaches on datasets generated using a benchmark generation tool developed by Bosch [20]. The evaluation examines the scalability of the proposed approaches and quantitatively compares the computation time, control performance, and resource utilization of the heuristic and the optimal CP formulation. While the CP formulation would produce the optimal results if allowed to run sufficiently long, for most real-life problems, the time required by CP to generate all optimal schedules would be unacceptable. In comparison, *Flexi* is able to produce valid schedules for many problems within a small fraction of the time required by CP, with only a small degradation in control performance. For example, using the tool from Bosch, we generated 100 synthetic benchmarks with parameters that represent realistic systems. Each such benchmark involved 1000 tasks that were mapped onto 16 ECUs. With a timeout set to 50 mins, *Flexi* produced feasible schedules for 93 systems, whereas the CP formulation could produce schedules only for 62 cases within that time. Further, for the cases where both approaches were able to produce a solution, *Flexi* could generate schedules 5$\times$ faster (i.e., took less time) with only a 0.5% degradation in the control



(a) Hardware platform with 2 applications from Figure 1(b) mapped to ECUs and switched network links.

(b) Application models.



(c) An example of a schedule with latency $L_1 = 7$ for Application 1, and $L_2 = 9$ for Application 2.

Fig. 1. Periodic scheduling problem description with an example solution. There are two applications with periods of 5 and 15 time units, respectively. The first application involves task a_1 sending a message to task a_4 with messages a_2 and a_3 mapped to links between $\text{ECU}_1 \rightarrow \text{Switch}_1$ and $\text{Switch}_1 \rightarrow \text{ECU}_2$, respectively. The second application comprises tasks a_5, a_8, and a_9 and messages a_6 and a_7.

performance compared to the solutions returned by the CP formulation.

The rest of this article is organized as follows: the platform, application, and control models are presented in Section II, followed by the problem formulation in Section III. Section IV describes the CP formulation. Next, Section V introduces the Flexi heuristic for scheduling periodic activities, and Section VI presents the experimental evaluation. The related work is discussed in Section VII, before concluding in Section VIII.

II. System Model

This section introduces the platform, application, and control models used in this article.

A. Platform Model

We consider a platform comprising multiple ECUs connected by a switched network based on the *time-triggered automotive Ethernet* with a tree topology, similar to the one in Figure 1(a). Such a distributed architecture is commonly used for automotive systems [21]. ECUs are typically grouped into multiple domains, where the ECUs in a domain are interconnected by links to a switch. The switches are connected in chains by network links, e.g., Switch_1 and Switch_2 in Figure 1(a). Thus, the number of resources m is computed as the sum of the number of ECUs m_E and the number of network links, i.e., $m = m_E + (2 \cdot m_E + 2 \cdot (m_D - 1))$, where m_D is the number of domains.

B. Application Model

The system functionality is realized by a *set of applications* App. Each application app_w runs periodically with a certain period $\overline{p}_w$. The application model is based on the characteristics of realistic benchmarks of modern automotive software systems, obtained using [20]. An application is composed of a set of *tasks* T that communicate via a set of messages transmitted over the switched network. Note that a data transmission between a pair of tasks is realized using more than one message depending on the number of links between the ECUs hosting the tasks. For the example in Figure 1(a), there are two messages, a_2 and a_3, representing a data transmission from task a_1 to task a_4. The *set of all messages* is denoted by M while the *set of all tasks* is represented by T. The *set of activities* is, therefore, given by $A = T \cup M$.

Each activity $a_i \in A$ is characterized by the tuple $\{map_i, p_i, e_i, pred_i, succ_i\}$ representing its mapping to the resource, period, processing time (either execution or transmission time), the set of direct predecessors and the set of direct successors, respectively. The *mapping* for each activity is given as $map_i : A \rightarrow \{1, 2, \cdots, m\}$. Whereas the mapping of tasks is given by application developers, the mapping of messages is straightforwardly derived from the task mapping for the tree topology of the switched network. There is only one path between each pair of ECUs in this topology.

We assume that time is discretized with sufficient precision. The applications are heterogeneous with respect to the volumes of data they transfer over the network, which is common in modern automotive systems. For instance, an engine management system transfers sensor values, and the autonomous driving system transfers video or lidar data. Thus, *network traffic comprising both small- and large-sized data packets is present in the system.*

Considering that the bandwidth of the network links is provided, the transmission time for each message $a_i \in M$ is calculated as $e_i = \frac{sz_i}{bnd} + ovr$. Here, sz_i is the size of the transmitted data as given in the application specification, bnd is the bandwidth of the network link, and ovr is its communication overhead given by the platform specification. Note that the proposed model can also be used when the bandwidth on different network links is not the same. In that case, we obtain different transmission times e_i for the same data in different network links.

Activities in an application are data-dependent with the dependencies represented using a general directed acyclic graph (DAG) of *precedence relations*. We show examples of such graphs for a set of two applications in Figure 1(b). Note that we do not make any assumption on the structure of the DAGs.

Precedence relations apply only to activities with the same period, which is common in the automotive domain [11], [14].

Control performance of an application typically degrades with sampling and actuation jitters [8]. Sampling and actuation are performed by certain application tasks and we implement these tasks with zero jitter. We also extend the zero-jitter restriction to other tasks to reduce the number of design variables in the schedule optimization problem. Communication is often the bottleneck in automotive systems, as demonstrated in the experiments in Section VI. We hence relax the jitter constraints for message transmissions on the network links to obtain a higher utilization of the communication resources, as shown in [27]. When the time difference between the executions of a data-dependent pair of tasks is larger than the time required to send the data over the network links, it provides flexibility to schedule the intermediate messages with bounded jitter. For the example in Figure 1(c), tasks a_1 and a_4 are scheduled with a time difference of 3, whereas it takes 2 time units in total to transmit messages a_2 and a_3. This enables us to schedule a_2 and a_3 with jitters that can add up to 1.

We define the end-to-end latency L_w of an application app_w as the time from the start of the first activity until the completion of the last activity in an application. Note that the end-to-end latency of a control application is also equal to the sensing-to-actuation delay in the control loop. Considering that the control performance typically degrades with an increase in sensing-to-actuation delay, the maximum possible latency for application app_w, denoted as $\hat{L}_w$, is determined based on the minimum performance requirement and it is assumed to be provided by the application developer.

An example is shown in Figure 1. In Figure 1(b), two applications with periods 5 and 15 respectively comprise tasks a_1, a_4, a_5, a_8 and a_9 that are mapped on to ECUs 1, 2, and 3. The tasks communicate via messages a_2, a_3, a_6, and a_7 over the links of a switched time-triggered network [36]. As shown in Figure 1(c), the tasks executing on the ECUs are scheduled with zero jitter, whereas the messages on the network links have non-zero jitter. For example, activity a_2 is scheduled at times 2, 6, and 11, i.e., not always with the same time offset relative to its period of 5. Furthermore, the end-to-end latency L_2 of Application 2 in this schedule is 9, where the first activity a_5 starts at time 1, while the last activity a_9 finishes at time 10.

C. Control Model

It is common practice in the automotive domain that the application period is given [23], [37] and is equal to the sampling period of the controller. The control performance may degrade with sampling and actuation jitters. In this article, we consider a zero-jitter implementation of a controller and schedule the sensing and actuation tasks accordingly. We furthermore consider physical plants that are controlled by software applications. Here, we only study linear time-invariant (LTI) systems, as they are common in practice. For LTI plants, the continuous-time mathematical model can be written as

$$\dot{x}(t) = A \cdot x(t) + B \cdot u(t),$$
$$y(t) = C \cdot x(t),$$

where $x(t)$, $u(t)$, and $y(t)$ represent the system states, control input and system output, respectively. A, B, and C are constant system matrices.

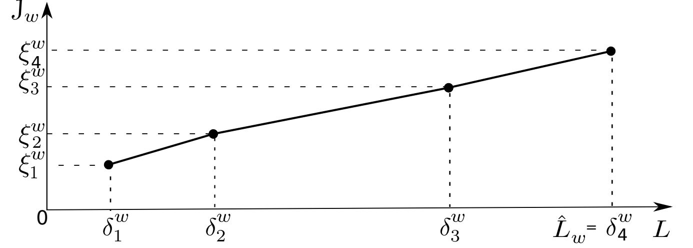


Fig. 2. Piecewise linear control performance function for app_w.

Traditionally, a software-based implementation of a controller for an app_w considers a constant sampling period $\overline{p}_w$. That is, the sensing task of a control application reads the system state $x(t)$ at time instants t_0, t_1, $\cdots$, t_n, where the sampling period is given by $\overline{p}_w = t_{k+1} - t_k$. We further assume that the control applications are implemented based on the *Logical Execution Time* (LET) paradigm [1]. The LET implementation paradigm provides a fixed sensing-to-actuation delay to realize a more predictable control [30]. Corresponding to the end-to-end latency L_w of the application app_w, the control input $u[k]$ calculated based on the state $x[k]$, i.e., $x(t_k)$ sensed at time t_k, is applied at time $t_k + L_w$.

With the above considerations, the equivalent sampled-data model is derived in [4] and can be written as

$$x[k+1] = \phi \cdot x[k] + \Gamma_0 \cdot u\left[k - \left\lfloor \frac{L_w}{\overline{p}_w} \right\rfloor\right] + \Gamma_1 \cdot u\left[k - \left\lceil \frac{L_w}{\overline{p}_w} \right\rceil\right],$$
$$y[k] = C \cdot x[k], \tag{1}$$

where ϕ, Γ_0 and Γ_1 for $\tau = L_w - \left\lfloor \frac{L_w}{\overline{p}_w} \right\rfloor$ are given by

$$\phi = e^{A \cdot \overline{p}_w}, \;\; \Gamma_0 = B \cdot \int_0^{\overline{p}_w - \tau} e^{At} dt, \;\; \Gamma_1 = B \cdot \int_{\overline{p}_w - \tau}^{\overline{p}_w} e^{At} dt.$$

It is assumed that the control input $u[k]$ in Equation (1) is computed according to the feedback control law [3] given by

$$u[k] = K \cdot x[k] + F \cdot r, \tag{2}$$

where r is the reference input and K and F are feedback and feedforward gains, respectively. These gains are designed by control engineers to satisfy certain control performance requirements assuming ideal implementation conditions, such as zero delay.

In this work, we consider *the settling time as the performance metric* for an application. It is defined as the time taken by the closed-loop system to reach and stay within a threshold of the reference input. Given the control law, continuous-time system matrices, sampling period and end-to-end latency, the closed-loop system can be simulated according to Equations (1) and (2), and the settling time can be calculated. Note that in most cases, the settling time is expected to increase with an increase in end-to-end latency.

Now, we compute the minimum possible end-to-end latency δ_1^w for an application app_w based on the precedence relations between its constituent tasks and messages. We further compute the maximum possible value of the end-to-end latency δ_N^w for which the settling time requirement is met. In addition, we consider $N-2$ discrete values of end-to-end latency between

δ_1^w and δ_N^w to obtain a set $\{\delta_1^w, \delta_2^w, \cdots, \delta_d^w, \cdots, \delta_N^w\}$ such that $\delta_{d+1}^w - \delta_d^w = \frac{\delta_N^w - \delta_1^w}{N-1}$. We construct a look-up table for each application that contains the values of the settling time ξ_d^w for each discrete value of end-to-end latency δ_d^w. Such a look-up table can be represented as $LUT_w = \{(\delta_d^w, \xi_d^w) | d = 1, \cdots, N\}$. Here, $\xi_d^w \leq \xi_{d+1}^w$ for $\delta_d^w < \delta_{d+1}^w$. LUT_w defines the control performance of an application app_w as a piecewise linear function of the end-to-end latency, as shown in Figure 2. Here, for the range of end-to-end latencies $[\delta_1^w, \delta_N^w]$, the designed controller is stable and satisfies the worst-case performance requirement.

Note that when an application app_w has exclusive access to resources, it is possible to achieve the lowest settling time ξ_1^w corresponding to the implementation with the minimum possible end-to-end latency δ_1^w. However, when multiple applications are sharing the resources, it may not be possible to implement all of them with their respective minimum end-to-end latencies. Here, the settling times of certain applications may be higher than the minimum. Our goal is to minimize such performance degradation while scheduling the applications. The table LUT_w is, hence, used to formulate the objective function for the scheduling problem. Note that this technique elevates scheduling to consider the real application performance, which is a novelty of this work.

III. Problem Formulation

We study *the non-preemptive scheduling problem on dedicated resources*. The aim is to find a schedule with a hyperperiod $H = lcm(p_i \in P)$, where lcm is the least common multiple function and P is a set of distinct values of activity periods. The schedule is defined by the start times $s_i^j \in \mathbb{Z}$ of each activity $a_i \in A$ for every occurrence $j = 1, 2, \cdots, n_i$ within a hyperperiod, where $n_i = \frac{H}{p_i}$. The schedule must consider the periodicity and the jitter limitation of each activity, while satisfying the precedence relations and the end-to-end latency constraint for each application. Moreover, it minimizes the control performance degradation. The periodic monoprocessor scheduling without jitter and precedence constraints is proven to be NP-hard by reduction from the 3-Partition problem in [16]. As the schedule optimization problem under consideration comprises the problem considered in [16] as a sub-problem, we conclude that it is also NP-hard.

A. Scheduling Constraints

The solution space is defined by four sets of constraints. We first consider a set of *jitter constraints* that requires each task to have zero jitter while running on ECUs. This is given by Equation (3). Note that there are no jitter constraints for messages, as previously explained in Section II-B.

$$\begin{aligned} s_i^j &= s_i^1 + (j-1) \cdot p_i, \\ a_i &\in T, \; j = 1, \cdots, n_i. \end{aligned} \tag{3}$$

The second set of constraints defines the *precedence relations* that ensure that all data dependencies are respected, as shown in Equation (4).

$$\begin{aligned} s_i^j - s_k^j &\geq e_k, \\ a_i, a_k \in A : a_k \in pred_i, \; & j = 1, \cdots, n_i. \end{aligned} \tag{4}$$

We further consider a set of *resource constraints* that prevents simultaneous scheduling (or collision) of two activities on a resource, as given by Equation (5). Such a constraint must be considered for each pair of occurrences within a hyperperiod for two different activities. Here, either occurrence j of activity i is executed after occurrence l of activity k or vice versa.

$$\begin{aligned} s_i^j - s_k^l \geq e_k \; XOR \;\; s_k^l - s_i^j \geq e_i, \\ \{a_i, a_k\} \in A : map_i = map_k, \\ j = 1, \cdots, n_i, \; l = 1, \cdots, n_k. \end{aligned} \tag{5}$$

We also consider a set of *end-to-end latency constraints*, as given by Equation (6). The end-to-end latency of an application is the maximum time duration between the start of a root activity and the completion of a leaf activity in the application. Note that a root activity has no predecessor while a leaf activity has no successor.

$$\begin{aligned} s_i^j + e_i - s_k^j \leq L_w, \\ a_i, a_k \in app_w : \; succ_i = \emptyset, \; pred_k = \emptyset, \\ j = 1, \cdots, n_i, \; app_w \in App, \end{aligned} \tag{6}$$

Note that $L_w \in \mathbb{Z}$ is a variable determined by the start times of activities. Equation (7) further constrains the end-to-end latency based on the maximum permissible value $\hat{L}_w$.

$$L_w \leq \hat{L}_w, \; app_w \in App. \tag{7}$$

B. Minimizing Control Performance Degradation

To compare the performance degradation of different control applications in response to scheduling decisions, we normalize the performance (i.e., the settling time of the closed-loop system) through division by the best possible value. For example, if the settling time of an application corresponding to the obtained schedule is $3\,\text{s}$ and the minimum possible settling time is $2\,\text{s}$, the normalized settling time is 1.5. Now, consider another application that has the minimum possible settling time of $100\,\text{ms}$ while the obtained settling time is $150\,\text{ms}$. Here, the normalized settling time is also 1.5. Thus, for both applications the performance degradation is the same, i.e., $50\,\%$ higher than the minimum possible value. Note that the higher the value of the normalized settling time is, the worse is the control performance.

We can update the look-up table LUT_w, defined in Section II-C, based on the normalization technique to obtain $\overline{LUT}_w = \{(\delta_d^w, \overline{\xi}_d^w) \mid d = 1, \cdots, N\}$, where $\overline{\xi}_d^w = \frac{\xi_d^w}{\xi_1^w}$. Here, $\overline{\xi}_d^w$ is the normalized settling time obtained with an end-to-end latency of δ_d^w. The normalized settling time J_w can be represented as a piecewise linear function of the end-to-end latency L_w as follows:

$$J_w = \begin{cases} F\big((\overline{\xi}_1^w, \delta_1^w), (\overline{\xi}_2^w, \delta_2^w), L_w\big), & \text{if } \delta_1^w \leq L_w \leq \delta_2^w; \\ F\big((\overline{\xi}_2^w, \delta_2^w), (\overline{\xi}_3^w, \delta_3^w), L_w\big), & \text{if } \delta_2^w \leq L_w \leq \delta_3^w; \\ \vdots & \\ F\big((\overline{\xi}_{N-1}^w, \delta_{N-1}^w), (\overline{\xi}_N^w, \delta_N^w), L_w\big), & \text{if } \delta_{N-1}^w \leq L_w \leq \delta_N^w. \end{cases} \tag{8}$$

Here, $F((x_1, y_1), (x_2, y_2), x_3)$ is the functional value at $x_3 \in [x_1, x_2]$ where $F(\cdot)$ is a line passing through the points (x_1, y_1) and (x_2, y_2). The piecewise linear function is thus similar to

the one shown in Figure 2. However, we plot the normalized settling time instead of the absolute value on the y-axis. This piecewise linearization allows us to implement the application for any end-to-end latency in the interval $[\delta_1^w, \delta_N^w]$. Thus, we are not restricted to the N end-to-end latency values for which we simulate the closed-loop system. This enhances the feasibility of the scheduling problem. Furthermore, note that the settling time is given by $N-1$ component functions. With a higher value of N, the scheduling problem becomes more complex, while with a lower value, the accuracy in the computation of settling time is compromised.

In this article, we define the objective function for the scheduling problem as follows:

$$\textit{Minimize:} \max_{app_w \in App} J_w. \quad (9)$$

That is, we minimize the maximum normalized settling time among all the control applications. This is equivalent to minimizing the maximum performance degradation, since the best normalized performance is unity for each application. The optimization approaches proposed in this article also work for other objective functions, such as *min* $\sum_{app_w \in App} J_w$ (see [25] for details and results).

IV. Constraint Programming Formulation

We have studied both Constraint Programming (CP) and Integer Linear Programming (ILP) formulations of our scheduling problem. Since the CP formulation outperforms the ILP formulation, we discuss only former and refer the reader to [25] for details on the ILP formulation.

A. Decision Variables

The *start time of occurrence j of activity a_i* is denoted by s_i^j, which is a decision variable of the CP problem. Here, we assume that the j-th period of an application app_w spans from time $(j-1)\cdot\overline{p}_w$ to time $j\cdot\overline{p}_w$. Accordingly, we allow the activities of an application to span over several periods in the resulting schedule, as in the case of Application 1 in Figure 1(c). Hence, we constrain start time variables as in Constraint (10), and we call them *time window* constraints.

$$LB_i^j = (j-1)\cdot p_i \le s_i^j \le j\cdot p_i - 1 + \hat{L}_w - e_i = UB_i^j. \quad (10)$$

To explain the intuition behind the upper bound computation, let us state two considerations: 1) Root activities (without predecessors) in the application are always zero-jitter (ZJ) tasks, i.e., they are scheduled at the same time in each period. 2) For ZJ tasks, the schedule with start time $s_i^1 = p_i$ is equivalent to the schedule with $s_i^1 = 0$. From these considerations, the j-th occurrence of the root activity a_i starts at the latest before the end of the corresponding period, i.e., at time $j\cdot p_i - 1$. Therefore, the last activity in the j-th occurrence of the application must finish before $j\cdot p_i - 1 + \hat{L}_w$ such that the maximum permissible value of end-to-end latency $\hat{L}_w$ is not exceeded. We further tighten the bounds on the start time variables using the knowledge about the precedence relations as described in [25].

As per Constraint (10), we might schedule an occurrence of an activity beyond the hyperperiod H. However, note that if we schedule an activity at time t then it repeats at time $t+H$. And therefore, we cannot schedule another activity at time $t+H$ on the same resource. Accordingly, we update Constraint (5) to prevent collisions of activities in a resource as in Constraint (11).

TABLE I
Functions used to formulate the model in CP Optimizer.

Constraint	*Reference*	*Function*
Jitter	(3)	*startAtStart*$(s_i^j, s_i^1, (j-1)\cdot p_i)$
Time window	(10)	*setStartMin*$(s_i^j,\ LB_i^j)$
		setEndMax$(s_i^j,\ UB_i^j)$
Precedence	(4), (12), (13)	*endBeforeStart*$(s_i^j,\ s_k^j)$
Resource	(11)	*noOverlap*(A_r), $A_r = \{a_i : map_i = r\}$
		startAtStart$(s_i^{j+f\cdot n_i},\ s_i^j,\ H)$
Latency	(6)	*span*(L_w, A_w), $A_w = \{a_i \in app_w\}$
	(7)	*setLengthMax*$(L_w,\ \hat{L}_w)$

$$\begin{gathered}(s_i^j + f\cdot H) - (s_k^l + h\cdot H) \ge e_k \\ XOR\ (s_k^l + h\cdot H) - (s_i^j + f\cdot H) \ge e_i, \\ f,h \in \{1,2,\cdots,n_H^{max}\},\ n_H^{max} = \left\lceil \frac{\max_{a_i\in A}\ UB_i^{n_i}}{H} \right\rceil. \end{gathered} \quad (11)$$

Furthermore, we must also consider constraints to ensure that occurrences of the same activity with non-zero jitter (i.e., a message) do not collide. Thus, we introduce Constraint (12) to guarantee the precedence relation between each pair of consecutive occurrences of an activity. In addition, we need to consider a constraint between the first and the last occurrences as in Constraint (13). This constraint also reduces symmetry in the solution space by preventing permutations of the start time of different occurrences of one activity.

$$s_i^j + e_i \le s_i^{j+1}, \quad (12)$$
$$s_i^{n_i} + e_i \le s_i^1 + H, \quad (13)$$
$$a_i \in M,\ j = 1,\cdots,n_i - 1.$$

B. Model Formulation

According to the experimental results, using solver-specific constraint prototypes to formulate the CP problem reduces the computation time significantly. In this section, we provide guidelines to formulate the schedule optimization problem under study in CP optimizer [15] using Java for interfacing. Decision variables s_i^j and L_w are implemented as interval variables, as they are typically well-suited to formulate scheduling problems. We further implement all constraints using specific functions available for interval variables as listed in Table I.

In our formulation, Constraint (11), i.e., the set of resource constraints, influences the computation time the most. This is because of (i) the large number of constraints and (ii) the non-convex nature of the constraints. Note that for a resource where n_r occurrences of activities need to be scheduled, the number of constraints that must be formulated is given by $\frac{n_r\cdot(n_r-1)}{2}$. Experimental evaluations have demonstrated that the most efficient formulation of these constraints uses the *noOverlap*(A_r) function. Here, A_r is the set of activities mapped on resource r and is given by $A_r = \{a_i \in A : map_i = r\}$ for $r = 1,2,\cdots,m$. Towards formulating Constraint (11), we consider a set of variables $\{s_i^j \,|\, j = 0,\cdots,n_H^{max}\cdot n_i\}$ for an

activity a_i. We further relate variables in the first hyperperiod with the variables in the subsequent hyperperiods using the function *startAtStart*($s_i^{j+f\cdot n_i}$, s_i^j, H), where the third parameter is the difference between the first two parameters. We use the same constraint prototype to formulate zero-jitter constraints (3) for the tasks. Note that the CP formulation in this section can be easily adapted for any combination of zero-jitter and non-zero jitter activities, i.e., it is not limited to zero-jitter tasks on ECUs (or any processing units) and non-zero jitter messages on networks.

We formulate the objective function using a built-in function *piecewiseLinearFunction*. To further reduce the computation time of the solver, we set the parameter *Workers* of the CP optimizer solver to 1. This implies that we do not want parallel threads for the computation. This setting has experimentally shown significant reduction in the computation time. From our experience, in the current version of CP optimizer (IBM ILOG CPLEX Optimization Studio 12.8), computation using only one thread is typically faster. The source codes for the problem formulation can be found in [26].

V. Heuristic Approach

This section introduces a heuristic approach called *Flexi* that first constructs a feasible solution and then optimizes it. Here, we also prove the property that we employ in the heuristic for a faster and near-optimal solution.

A. Overall Approach

Flexi comprises two stages, namely, the feasibility and the optimization stages, respectively. The feasibility stage is illustrated in Figure 3. In the *internal loop* (Steps 2 and 3), it constructs a schedule by inserting elements (activities or activity occurrences) one-by-one. The construction is based on an ordered set that is referred to as the priority queue Q. If a feasible schedule is obtained, this stage is complete. However, when an element ϵ_i cannot be scheduled, it is split into occurrences if it is a message and some occurrences can be scheduled (Step 5). Then, the set of elements D preventing ϵ_i from being scheduled, is found (Step 6). Note that the elements of D are already in the schedule, whereas ϵ_i is not. Finally, the problematic occurrences of ϵ_i with its predecessors are put before all elements of D in Q (Step 7). We refer to Steps (2-8) as the *external loop* of our algorithm. The feasibility stage stops when all occurrences are scheduled, an infinite loop is detected, or an iteration budget is exhausted.

Note that the proposed feasibility stage is an improvement over the 3-LS heuristic presented in [27], where a schedule is also constructed by assigning start times to activities. Among other differences, the 3-LS heuristic considers only a single granularity level, i.e., scheduling and removing activities only. In contrast, Flexi *adjusts the granularity of the elements in the priority queue*. In the beginning, elements are activities. However, the message is split into occurrences if only some occurrences fail to be scheduled. Note that tasks on ECUs are always scheduled as one entity because they are assumed to be zero-jitter activities. According to Constraint (3), the schedule for all their occurrences can be derived from the first one straightforwardly. Considering that we do not increase the number of scheduling entities when it is not necessary, we

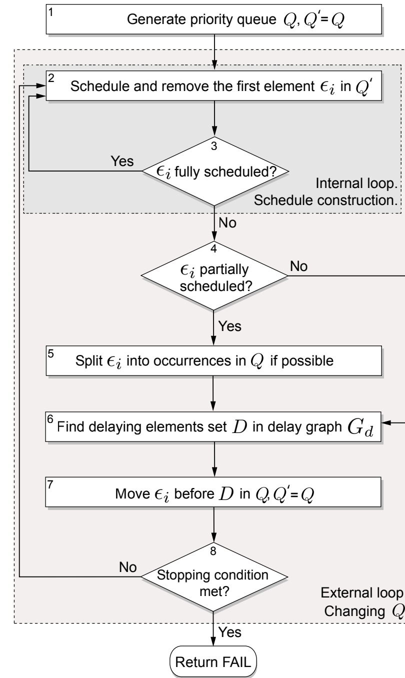


Fig. 3. Outline of the feasibility stage of Flexi.

ensure the coarsest possible level of scheduling granularity at each phase of the algorithm.

Finally, when the feasibility stage finds a solution, the optimization stage iteratively applies a *Large Neighborhood Search* technique [29] to improve the solution. In each iteration, Large Neighborhood Search solves the optimization problem described in Section III for a chosen set of applications (called a *neighborhood*), while fixing the schedule for the rest of the applications to the best one found thus far. Therefore, it looks *locally* in the neighborhood defined by the chosen set of applications for a better solution. This strategy reduces the computation time by limiting the search space and considering only a subset of the decision variables. The optimization stage uses the CP (or the ILP) problem formulation from Section IV (or [25]).

B. Feasibility Stage

Here, we first explain the concept of a *delay graph*. This graph is used to find the set of prescheduled elements that prevent the current element from being scheduled. We also formulate and prove a necessary and sufficient condition for scheduling a zero-jitter activity given a set of prescheduled activities that we exploit to accelerate the search for a solution. We further outline the *sub-problem* that schedules a single element, respecting a partial schedule of the higher priority elements. Finally, we present the algorithm implementing the feasibility stage.

1) Delay Graph to Modify Priority Queue: A *delay graph* is used to find a set of elements D that prevents the current element ϵ_i from being scheduled. A delay graph G_d is an acyclic directed graph constructed using the current element ϵ_i and the existing schedule. Nodes of G_d are elements, while edges directed from one node to another indicate that the former node prevents the latter node from starting earlier due to the resource or precedence constraints. Note that

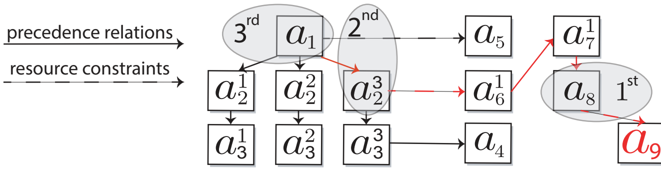


Fig. 4. Example of a delay graph with the coarsest level of granularity for the applications in Figure 1(b) that are scheduled as in Figure 1(c). Elements on different delay levels for a_9 are marked using gray ovals.

nodes corresponding to non-zero jitter messages can be split into occurrences during the algorithm run, whereas nodes corresponding to zero-jitter tasks cannot.

To make the scheduling process more straightforward, we do not allow an element to be considered by the scheduler before its predecessors $pred_i$ in the priority queue Q. Therefore, the earliest time $\check{s}_i$ for the start time of ϵ_i is the maximum of the completion times of its predecessors, as in Equation (14).

$$\check{s}_i = \max_{\epsilon_k \in pred_i} (s_k^j + e_k) \tag{14}$$

If no activity prevents ϵ_i from being scheduled directly after the latest predecessor(s) (i.e., at time $\check{s}_i$), we add edge(s) from its latest predecessor(s) to ϵ_i. Thus, we can only have an edge from an element on another resource if it is a predecessor. If an element cannot be scheduled at $\check{s}_i$ due to the resource constraints, the element that causes the delay is always found. In this case, the conflicting element is mapped on the same resource as ϵ_i.

Figure 4 shows a delay graph for the applications in Figure 1(b) that are scheduled as in Figure 1(c). Here, the priority queue is set as $Q = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9\}$. For all the elements except a_5, a_6, and a_9, a corresponding parent is its predecessor. Observe that the activity a_5 has no predecessor, and thus, can start at time 0. However, it is scheduled at time 1 because a_1 is scheduled on the same resource at time 0. Therefore, we draw an edge from a_1 to a_5. The same reason holds for the pairs $\{a_2^3, a_6^1\}$ and $\{a_8, a_9\}$. Note that the DAG of precedence relations only slightly influences the complexity of the delay graph. Considering that we target problem instances with reasonably high utilization, delaying elements for most of the activities are not their predecessors, but conflicting activities on the same resource.

An element ϵ_k on *delay level* l^d for the element ϵ_i in G_d has the shortest distance (in terms of the number of edges) equal to l^d, i.e., dist($\epsilon_i, \epsilon_k, G_d$) = l^d. Note that we count only edges between elements that are not predecessors of ϵ_i because no element in the priority queue can be before its predecessors. For the simple example in Figure 4, the element on the 1st delay level for the activity a_9 is a_8, on the second level is a_2^3, and on the third level is a_1.

2) Computation Time Improvement: Here, we formulate and prove the necessary and sufficient condition for schedulability of two zero-jitter tasks. We use this result to reduce the computation time of Flexi.

We first provide the necessary background for, and the intuition behind, the formulation. As stated in Equation (5) in Section III, two activities do not collide when the difference between their start times is larger than or equal to the processing time of the activity that is scheduled earlier, i.e., if $s_i^j \geq s_k^l$,

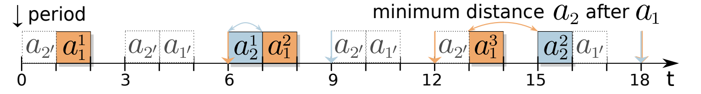


Fig. 5. A schedule of two ZJ tasks a_1 and a_2 with periods $p_1 = 6$ and $p_2 = 9$ with the minimum start-to-start interoccurrence distance 1 from a_2 to a_1 and 2 from a_1 to a_2. Dotted lines are the ZJ tasks $a_{1'}$ and $a_{2'}$ with $p_{1'} = p_{2'} = \gcd(p_1, p_2) = 3$ from Property 1.

Equation (15) holds:

$$s_i^j - s_k^l \geq e_k. \tag{15}$$

For two zero-jitter tasks, possibly with different periods, we are interested in all the differences given by:

$$\begin{aligned} s_i^j - s_k^l &= (s_i^1 + (j-1)\cdot p_i) - (s_k^1 + (l-1)\cdot p_k) = \\ &= (s_i^1 - s_k^1) + ((j-1)\cdot p_i - (l-1)\cdot p_k),\ j,\ l \in \mathbb{Z}^+, \end{aligned} \tag{16}$$

according to Equation (3). We derive the minimum among these differences using the Bezout identity [6]. It states that if p_i and p_k are integers with *greatest common divisor* $g_{i,k} = \gcd(p_k, p_i)$, then the integers of the form $j \cdot p_i + l \cdot p_k$ is a multiple of $g_{i,k}$. Then, in Equation (16), we can substitute integers j and l for a variable $z_{i,k} \in \mathbb{Z}$ for each ordered pair of tasks and it can be rewritten as $s_i^1 - s_k^1 + z_{i,k} \cdot g_{i,k}$. The minimum value of the difference is, therefore, $(s_i^1 - s_k^1) \bmod g_{i,k}$ over all possible values of $z_{i,k}$. Thus, the resource constraints (5) for a pair of zero-jitter tasks on an ECU can be formulated as in Equation (17). Note that the modulo operator is defined as $a \bmod c = a + q \cdot c$ with $q \in \mathbb{Z}$ such that $0 \leq a + q \cdot c < c$, which makes these constraints work with arbitrary order of s_i^1 and s_k^1, i.e., with a negative value of a.

$$\begin{aligned} (s_i^1 - s_k^1) \bmod g_{i,k} \geq e_k, \\ (s_k^1 - s_i^1) \bmod g_{i,k} \geq e_i, \end{aligned} \tag{17}$$

This result also implies that a necessary condition for schedulability of two ZJ tasks is $e_i + e_k \leq g_{i,k}$, as proven in [19].

The example in Figure 5 presents a schedule of two ZJ tasks a_1 and a_2 with periods $p_1 = 6$ and $p_2 = 9$, respectively. We can see that the value $(s_1^1 - s_2^1) \bmod g_{1,2} = (1-6) \bmod 3 = 1$ is the minimum distance among all occurrences j and l when a_2^j is before a_1^l in time, whereas it is $(s_2^1 - s_1^1) \bmod g_{1,2} = (6-1) \bmod 3 = 2$ when a_2^j is after a_1^l.

Next, we present the necessary and sufficient condition for schedulability of two zero-jitter tasks.

Property 1 (Schedulability of two ZJ tasks): Let $a_{i'}$ and $a_{k'}$ are tasks with processing times $e_{i'} = e_i$ and $e_{k'} = e_k$, and periods $p_{i'} = p_{k'} = g_{i,k}$. Then, two ZJ tasks a_i and a_k can be scheduled without collisions if and only if tasks $a_{i'}$ and $a_{k'}$ can be scheduled without collisions.

Proof: Let the original tasks a_i and a_k be schedulable. Then, Equations (17) hold for some s_i^1 and s_k^1. We set $s_{i'}^1 = s_i^1 \bmod g_{i,k}$ and $s_{k'}^1 = s_k^1 \bmod g_{i,k}$. Due to the additive property of modular arithmetic, $(s_{i'}^1 - s_{k'}^1) \bmod g_{i,k} = (s_i^1 \bmod g_{i,k} - s_k^1 \bmod g_{i,k}) \bmod g_{i,k} = (s_i^1 - s_k^1) \bmod g_{i,k}$, Equations (17) also hold for $s_{i'}^1$ and $s_{k'}^1$.

In the other direction, let Equations (17) hold for $a_{i'}$ and $a_{k'}$ for some $s_{i'}^1$ and $s_{k'}^1$. Then, we set $s_i^1 = s_{i'}^1$ and $s_k^1 = s_{k'}^1$. Then, Equations (17) trivially hold for s_i^1 and s_k^1. □

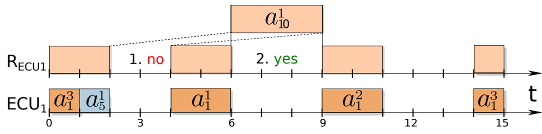


Fig. 6. Example of a R_{ECU_1} set for the given schedule on ECU$_1$ before scheduling new activity a_{10}^1.

Thus, we can find non-conflicting s_i and s_k for two ZJ tasks with arbitrary periods if and only if we can find $s_{i'}$ and $s_{k'}$ for two ZJ tasks with an identical period equal to the greatest common divisor of p_i and p_k and with initial processing times.

In the insert element procedure described next, we only need to go over time intervals for the first task occurrence instead of looking at time intervals for all occurrences for ZJ tasks using this property. It results in significantly reduced computation time since there are many ZJ tasks with distinct periods.

3) Insert Element Procedure: The insert element procedure takes the current element that needs to be scheduled and the resource on which it is mapped. Using the precedence relation and the end-to-end latency constraint, it determines the minimum and the maximum permissible values for the start time of the current element. It also finds the time intervals for which the resource is occupied, and therefore, it computes the earliest possible start time for the element.

Algorithm 1 implements the procedure. As inputs, it takes (i) the element ϵ_i to be scheduled; (ii) $\check{s}_i$ and $\hat{s}_i$, the earliest and the latest possible start times of this element based on the precedence relation and the end-to-end latency constraint, computed according to Equations (14) and (18), respectively. We compute $\hat{s}_i$ in Equation (18) as the maximum of three values: 1) its lower bound LB_i^j defined by Equation (10), i.e., start of the corresponding period; 2) completion time of the previous element occurrence if it is a message and $j > 1$; and 3) start time of the earliest activity in the application plus the end-to-end latency bound $\hat{L}_w$ minus processing time of the element e_i to satisfy end-to-end latency constraints.

$$\hat{s}_i^j = \max(LB_i^j,\ s_i^{j-1} + e_i,\ \min_{a_k \in app_w}(s_k^j) + \hat{L}_w - e_k) \quad (18)$$

Finally, the algorithm also considers the set of predecessor elements $\overline{pred}_i$ finishing at $\check{s}_i$, and the set of time intervals where resource r is occupied, R_r, which is the union of time intervals in which prescheduled occurrences are running on resource $r = map_i$.

The set R_r is a union of time intervals, for which resource r is occupied by already scheduled activities. For the example in Figure 6, $R_{ECU_1} = \{[0, 2], [4, 6], [9, 11], [14, 15]\}$. It is introduced to reduce the computation time of the procedure. This is achieved by iterating over intervals in R_r, instead of going over all scheduled elements and checking that no element is already scheduled at a given time. Formally, R_r is a union of nonintersecting intervals, sorted in ascending order, i.e., $R_r = \bigcup_{b=1}^{n_{int}} [l_b, u_b]$ with $l_b, u_b \in \mathbb{N}$ due to discrete-time and integer processing times and $l_b < u_b < l_{b+1}$.

To schedule ZJ task a_i, we use Property 1 as follows. For each prescheduled task a_k on the same ECU, we add processing time intervals of task $a_{k'}$ with the same processing time, but with a period equal to $p_{k'} = g_{i,k}$ to R_r (Line 7). The main loop of the insert element procedure iterates over intervals in

Input: ϵ_i, $\check{s}_i$, $\hat{s}_i$, $\overline{pred}_i$, R_r

```
1  if ϵ_i ∈ T then // Consider ZJ tasks
2      for ϵ_k ∈ T : map_k = map_i do
3          s^1_k' = s_k mod g_i,k;
4          j = 2;
5          while s_k' ≤ min{ŝ_i, H} do
6              s^j_k' = s^1_k' + (j − 1) · g_i,k;
7              R_r.add([s^j_k', s^j_k' + e_k]);
8              j = j + 1;
9          end
10     end
11 end
12 for b = 1 to n_int do // Main loop to find earliest s_i
13     if min(l_b, ŝ_i) − max(u_b−1, š_i) ≥ e_i then
14         s_i = max(u_b, š_i);
15         if s_i = š_i then // Construct delay graph
16             D = pred_i;
17         else
18             D = {ϵ_k ∈ A : s_k ≤ š_i < s_k + e_k};
19         end
20     end
21 end
22 if ϵ_i is not scheduled then
23     D = {ϵ_k ∈ A : s_k < š_i + e_i, s_k + e_k > š_i, map_i = map_k};
24 end
```

Output: s_i, D

Algorithm 1: Insert element procedure

R_r until it finds free space in the schedule after $\check{s}_i$. Then, if there is a free time interval of length e_i starting at $\check{s}_i$, we set the parents in the delay graph to elements in $\overline{pred}_i$ (i.e., to the predecessors finishing at $\check{s}_i$) (Line 16). Otherwise, the delaying element is the element scheduled at the earliest possible start time $\check{s}_i$ (Line 18). Finally, if ϵ_i was not scheduled, we define the set of parents D in the delay graph G_d for ϵ_i. This set is defined as the elements scheduled on the same resource that start before the earliest possible completion due to the precedence constraints $\check{s}_i$ and end after its earliest possible start time (Line 23). In other words, the elements that are scheduled in the time interval $[\check{s}_i, \check{s}_i + e_i)$.

The insert procedure always schedules an element as soon as possible. The motivation is twofold: first, it produces schedules with smaller fragmentation in each iteration. This is an especially sensitive issue, since we have both small (control) and large (video traffic) transmission times of frames on the network. Thus, frames with large transmission times have higher risk of not being scheduled given a schedule with high fragmentation, whereas frames with small transmission times can heavily fragment the schedule. The second reason is that this strategy results in a solution with lower end-to-end latency of the applications and, therefore, potentially a better objective value, i.e., lower control performance degradation.

It may be necessary to go beyond one hyperperiod because the start time variables can be larger than one H as per Equation (10). For this purpose, we look at R_r in the hyperperiods, to which the interval $[\check{s}_i^j, \hat{s}_i^j]$ belongs. Due to the periodicity of the schedule, we just add the corresponding number of hyperperiods to the interval bounds l_b and u_b. This is not included in Algorithm 1 for better readability.

4) Feasibility Stage Algorithm: The feasibility stage of Flexi is implemented by Algorithm 2. The inputs are the set of activities A, the rule to set the priority queue Pr_{feas}, and the iteration budget of the feasibility stage it^{max}. After populating

```
Input: A, Pr_feas, it^max
1  Q = sort(A, Pr_feas);
2  l^d = 1, it = 0;
3  while it < it^max and Q ∉ Q^prev and S not found do
     // External loop. Changing Q
4    Q' = Q;
5    Q^prev.add(Q);
6    while Q' ≠ ∅ and ε_i is scheduled do // Internal loop
7      ε_i = Q'.pop();
8      s_i^j, D = InsertElement(ε_i, š_i^j, ŝ_i^j, pred_i^j (overlined), R_r);
9    end
10   if ε_i is not scheduled then
11     D = {ε_k ∈ A : dist(ε_i, ε_k, G_d) = l^d};
12     if (ε_i, D) ≠ (ε_i^prev, D^prev) then
13       l^d = l^d + 1;
14       D = {ε_k ∈ A : dist(ε_i, ε_k, G_d) = l^d};
15     else
16       l^d = 1;
17     end
18     Q.splitAndPutBefore({ε_i, all_pred_i}, D);
19     D^prev = D, ε_i^prev = ε_i;
20   end
21   it = it + 1;
22 end
Output: S
```

Algorithm 2: Feasibility stage of Flexi

the priority queue Q (Line 1), the algorithm iterates over the external loop, where it adjusts the priority queue according to the feedback it gets from the internal loop. If some element ϵ_i failed to be scheduled by the insert element procedure (Line 10), we first find the set of delaying elements D (Line 11) on the current delay level l^d, as described in the previous subsection.

If both the problematic element ϵ_i and its set of parents G_d in the delay graph are the same as in the previous iteration (Line 12), we increase the delay level by one and assign the new delaying elements set (Line 14). In contrast, when either ϵ_i or D have changed, i.e., when either the problematic element ϵ_i from the previous iteration has been successfully scheduled or we have a different set of the delaying elements, the delay level is reset to 1 (Line 16). Finally, the current problematic element is split if necessary, and is promoted in Q with all predecessors to be immediately before all elements in D. This strategy prevents us from getting stuck with the same problematic element again. Moreover, this strategy aims not to disturb the prescheduled elements, rescheduling them only if necessary. The algorithm is terminated in three cases: 1) a complete solution is found, 2) an infinite loop over iterations of the external loop is detected, or 3) the iteration budget for the external loop is exhausted. Considering that both internal and external loops are deterministic, we identify an infinite loop when we encounter the same priority queue Q for the second time. However, we additionally set the iteration budget for the external loop because we aim for problem instances with more than 100,000 occurrences.

As the feasibility stage takes only a short time to run, we employ two different strategies to set priorities Pr_{feas} as follows:

1) in increasing order of the slack values $UB_i - LB_i$,
2) in decreasing order of potential to improve the objective value (normalized settling time), i.e., $\overline{\xi}^w_{N^w} - \overline{\xi}^w_1$.

We then choose the solution with the best objective value. This strategy has paid off, since for problem instances with high utilization we need to target feasibility with Strategy 1. On the other hand, for less-utilized systems, it is beneficial to use Strategy 2 to get a better criterion value.

The asymptotic complexity of the feasibility stage is $\mathcal{O}(n_{el} \cdot \frac{H}{2} \cdot it^{max})$, where n_{el} is the maximum number of elements to schedule (equal to the sum of number of tasks and number of message occurrences), and $\frac{H}{2}$ is the maximum number of intervals to explore in R_r for any resource r. This is the worst case that happens when the schedule consists of activities with a unit processing time scheduled with a gap of one time unit.

C. Optimization Stage

In the optimization stage, an iterative search is performed in the neighborhood of solution S_{best} with the best objective value Φ_{best} found so far. The neighborhood is given by the solutions with different start times of activities in only a subset of applications.

Algorithm 3 implements the optimization stage of Flexi, where the input values are: the schedule obtained by the feasibility stage S with objective value Φ_S, the number of applications considered in the neighborhood N_{apps}, the number of solutions (neighbors) to consider N_{sol}, the improvement tolerance τ_{opt} to stop the search, the priority rule Pr_{opt} to choose the neighborhood, and the maximum computation time t^{opt}_{max} for one run of the optimal model, which is used to evaluate a neighbor.

Start times of activities of applications that are not in the current neighborhood are fixed to the corresponding values in S_{best}, whereas start times of activities of applications chosen to be in the neighborhood can be changed. For each iteration of the inner loop, the neighborhood set $S_{neigh} \in App$ consisting of N_{sol} solutions, is generated (Line 7). Then, the resulting problem is solved by the CP formulation from Section IV or the ILP formulation from [25] (Line 8). The computation time is limited by t^{opt}_{max} to address the time complexity of the optimal approach.

While looking at the neighbors of the current solution, the neighbor with the best objective value is stored (Line 9). After evaluating all neighbors, S_{best} is updated (Line 11). Neighbors of the new solution are generated in the next iteration if the improvement over the previous iteration was larger than τ_{opt}. Otherwise, the search process is discontinued, and the best solution is returned.

```
Input: S, N_apps, N_sol, τ_opt, Pr_opt, t^opt_max
1  Φ_best = Φ_S;
2  Φ_cur = Φ_best + τ_opt;
3  S_best = S;
4  while Φ_cur − Φ_best > τ_opt do
5    Φ_cur.initialize();
6    for b = 1 to N_sol do
7      S_neigh = App.getApps(N_apps, N_sol, Pr_opt);
8      S = solve(S_best, S_neigh, t^opt_max);
9      Φ_cur.update(), S_cur.update();
10   end
11   Φ_best.update(), S_best.update();
12 end
Output: S_best
```

Algorithm 3: Optimization stage of Flexi

TABLE II
PLATFORM-GENERATION PARAMETERS.

Set	*P [ms]*	n_T	n_e	m_E	U_{min}
1	$1, 2, 5, 10$	30	15	2	0.5
2	$1, 2, 5, 10, 20, 50, 100$	50	30	2	0.6
3	$1, 2, 5, 10, 20, 50, 100$	100	30	3	0.65
4	$1, 2, 5, 10, 20, 50, 100$	500	50	8	0.7
5	$1, 2, 5, 10, 20, 50, 100$	1000	100	16	0.7

The priority rule Pr_{opt} to set applications in neighborhood is based on the application's *potential for improvement*, which is defined as the difference between the current control performance value and the best value that can be obtained for the application, i.e., $J_w - \overline{\xi}_1^w$. The applications with the maximum difference (potential) are chosen. We have experimentally compared different Pr_{opt} in [25], and this strategy has shown the best results since it exhibits a reasonable trade-off between computation time and solution quality.

Considering that the optimization stage uses a CP solver or an ILP solver, its worst-case asymptotic time complexity is not polynomial.

VI. EXPERIMENTAL RESULTS

We quantify the trade-off between the computation time and the quality of the solution for the proposed Flexi and CP approaches, and the existing 3-LS heuristic and ILP approaches.

A. Experimental Setup

Experiments are performed on problem instances generated by a tool developed by Bosch [20]. The applications are assumed to be executed on a platform similar to the one in Figure 1(a), i.e., the ECUs are connected by a time-triggered network with a tree-based topology, as previously described in Section II. Moreover, we synthetically generate control performance degradation values by simulating realistic control applications. There are five sets, each comprising 100 problem instances of different sizes (accessed at [26]). The generation parameters for each set are presented in Table II, and the granularity of the timer is set to be 1 μs. Each generated set of problem instances comprises a given number of tasks n_T, and we set the expected number of tasks executed on one ECU to n_e. We compute the number of ECUs as $m_E = \lceil \frac{n_T}{n_e} \rceil$.

The mapping of tasks to ECUs is done in the following way. The probability of interdomain communication is set to 0.2, i.e., 20% of communicating tasks are situated on ECUs in different domains. Note that setting this parameter too high results in unschedulable instances with overloaded links between switches. The mapping of tasks to ECUs is performed such that the load is balanced across the ECUs, i.e., the resulting mapping utilizes all ECUs approximately equally. The mapping of messages to the links, on the other hand, follows straightforwardly from the platform topology and the task mapping, as there is always exactly one route between each pair of ECUs.

Each application represents a control function for which the plant model is derived from the automotive domain. These plants represent DC motor speed control [32], DC motor position control [31], car suspension [32] and cruise control systems [32]. Given a sampling period (equal to the application task and message repetition periods), we design a controller assuming a specific delay. Now, for given maximum and minimum possible delay values and the granularity of discretization, we compute a set of delay values. For each delay value δ_d^w and the given sampling period p_w, we simulate the closed-loop system (i.e., the controller and the plant) for step response, according to Equations (1) and (2), to analyze the settling time ξ_d^w. Thus, for each application, we compute a table showing the variation in the settling time with the delay.

We assume a time-triggered Ethernet with a bandwidth of 100 Mbps [36]. For messages that transmit video content, we set a maximum desirable utilization of one video message in its period to 0.1 to still be schedulable, while reflecting typical message sizes. This corresponds to one message transmitting maximally 10 Mb per second, which is a realistic assumption since data that exceed this value are typically split into multiple messages due to decreasing reliability of the transmission with increasing message sizes.

While generating a problem instance, if the utilization of one of the resources is greater than 100%, or the utilization of each resource is less than U_{min} given in Table II, the problem instance is discarded and generated again. The resulting problem instances on average contain 82, 168, 421 and 6,276 and 12,552 activities (tasks and messages) for Sets 1-5, respectively. We set the latency bound for each application to twice its period, i.e., $\hat{L}_w = 2 \cdot \overline{p}_w$, as it allows for some flexibility of the solution while not jeopardizing its control performance [35]. For each application, we have assumed $N = 20$. Thus, for 20 equally spaced discrete values of latency within the bound, we perform closed-loop simulations and note the settling times. Moreover, the parameters for the optimization stage of the Flexi are set in the following manner: $N_{apps} = (2, 2, 2, 3, 3)$, $N_{sol} = (3, 3, 3, 2, 2)$, $\tau_{opt} = 10^{-2}$, $it^{max} = (100, 300, 500, 1000, 3000)$, $t_{max}^{opt} = (10, 20, 30, 300, 600)$ seconds given for Sets 1 to 5. These values have experimentally demonstrated reasonable results in terms of computation time, feasibility, and solution quality.

We performed the experiments on a server equipped with a 2x Intel Xeon E5-2690 v4 CPU, 3 Cores/CPU processor, and 64 GB memory. The ILP and CP models were implemented in IBM ILOG CPLEX Optimization Studio 12.8 and solved with the CPLEX and CP optimizer solvers using concert technology. The ILP, CP, and heuristic approaches were implemented in the Java programming language.

B. Results

The experiments compare the ability of the proposed Flexi CP (Flexi with CP model used in the optimization stage), Flexi ILP (Flexi with ILP model used in the optimization stage), Optimal CP, Optimal ILP, and the 3-LS heuristic from [27] to find a feasible solution to the generated sets of problem instances. Moreover, the trade-off between the computation time and the solution quality is evaluated.

We set a time limit of 3,000 seconds per problem instance to obtain the results in a reasonable time, and we use the best solution found thus far if the time limit is hit. Note that the optimal approach can stop either when an optimal solution is found or the time limit is reached. In the latter case, the optimal approach has either found a feasible solution or not.

TABLE III
THE NUMBER OF PROBLEM INSTANCES WHERE THE APPROACHES FAILED TO FIND A FEASIBLE SOLUTION WITHIN A TIME-LIMIT OF 3,000 SECONDS. THE NUMBER OF TIME OUTS IS AFTER THE SLASH.

Approach	Set 1	Set 2	Set 3	Set 4	Set 5
3-LS	7	58	71	44	68
Flexi	1	9	8	6	7
Optimal CP	0/7	2/22	0/50	13/100	38/100
Optimal ILP	0/13	78/100	-	-	-

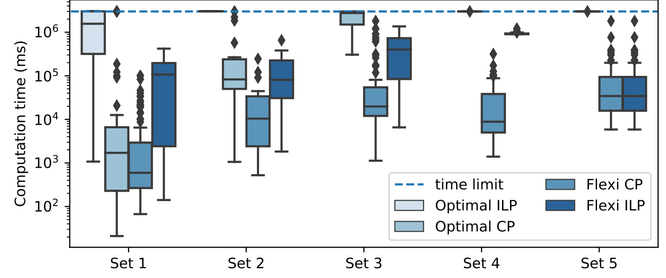


Fig. 7. Computation time (in a logarithmic scale) of the Optimal CP, Optimal ILP, Flexi CP, and Flexi ILP for Sets 1-5.

Flexi finishes either when it is not able to find a feasible solution during the feasibility stage or when it is done with the optimality stage, in which case we have a solution. Note that the experimental results are dependent on the value of the time limit used for the optimal approaches, as it affects the computation time as well as the quality of the obtained solution.

Figures in this section show the distribution in the form of box plots, indicating the first quartile, median, and third quartile together with outliers (diamonds) [17]. Outliers are numbers that lie outside $1.5\times$the interquartile range (third quartile value minus first quartile value) away from the top or bottom of the box that are represented by the top and bottom whiskers, respectively. Note that all outliers were also successfully solved within the time limit.

The only existing approach solving the schedule optimization problem under consideration or its generalization is the 3-LS heuristic [27]. The main reason is the difference in the jitter constraints on the ECUs (zero-jitter) and on the network links (non-zero jitter) that we consider in this article. Therefore, in the following section, we compare Flexi and the 3-LS heuristic approaches to show that the strategies to improve flexibility, described in Section V, have resulted in better efficiency of Flexi.

1) Feasibility Evaluation: Table III presents the number of problem instances for which the 3-LS heuristic, Flexi, and the optimal approaches were unable to find a feasible solution within the given time limit of 3,000 seconds. It also presents the number of instances, for which the two optimal approaches failed to prove the optimality of the solution (after the slash), both out of 100 instances. The results show that Flexi is significantly more successful in finding feasible solutions for our problem than the 3-LS heuristic and that it shows non-decreasing quality with increasing problem size, unlike CP.

Regarding the optimal approaches, *Optimal CP significantly outperforms Optimal ILP*: ILP fails to find a solution for more than three-quarters of problem instances already for Set 2, whereas CP finds solutions for most of the instances in the largest Set 5. Thus, whereas *Optimal ILP is only suitable for small-sized problems with hundreds of activities, Optimal CP handles medium-sized problems with thousands of activities well. Finally, Flexi finds feasible solutions for more problem instances in the largest problems with tens of thousands of activities.*

2) Trade-Off between Computation Time and Solution Quality: Figure 7 shows the computation time comparison for the problem instances from Sets 1 to 5, for which all approaches were able to find a feasible solution. The results of Optimal ILP are not shown for Sets 2-5, as already for Set 2 there are only 22 instances for which all approaches found a solution and Optimal ILP times out in all cases.

The computation time of all approaches increases exponentially with increasing problem size. This tendency is less visible for Optimal CP due to the growing number of time outs with increasing problem size. Furthermore, the optimality stage of Flexi using both ILP (Flexi ILP) and CP (Flexi CP) was not run on Set 5, since it did not result in significant criterion improvement for the smaller Set 4. Thus, only the computation time of the feasibility stage is shown in Figure 7 for Set 5. Note that for Flexi ILP this happens already for Set 4, where it cannot find any solution within the given time limit of 300 seconds per one neighbor for all problem instances. Flexi exhibits significantly lower computation times than Optimal CP, on average: 3, 2.5, 5, 3 and $5\times$ less for Flexi ILP and 60, 39, 20, 7, $5\times$ less for Flexi CP for Sets 1 to 5, respectively. The difference does not decrease monotonically due to 1) the optimality stage not running on instances of Set 5 (decreasing its computation time) and 2) Set 2 containing instances of lower scheduling complexity.

To justify that the long computation time for ILP is not a result of an inefficient solver, we also implemented the Optimal ILP model using Gurobi Optimizer 8.0 [13], and it failed to find a feasible solution for 39 instances out of 100 in Set 2 within the time limit. This is better than the results of CPLEX stated in Table III, but still not as good as Optimal CP. The reason is a large number of resource constraints given by Equation (5). In particular, Optimal ILP has as many constraints as there are pairs of message occurrences. In contrast, Optimal CP uses a single built-in constraint $\mathrm{noOverlap}$ for one resource in CP Optimizer. This tremendously reduces the computation time due to the elaborate constraint propagation techniques in the solver. It finds a solution even for Set 5 (for 62 instances out of 100), with up to 25,000 activities within the given time limit, although failing to prove an optimal solution for all 100 problem instances, as shown in Table III. Due to the significantly better results, only Optimal CP is used for further comparison with Flexi.

Figure 8 shows the relative difference in the objective values of Flexi ILP, Flexi CP, and Optimal CP approaches for problem instances where all approaches were able to solve within the time limit. Note that there are certain instances where the objective value obtained using Flexi can be as high as a few hundred percent compared to the Optimal CP formulation. This can be attributed to a few physical plant models (considered in the experiments) that are very sensitive to the end-to-end

latency. For the most latency-sensitive plant, the settling time varies between $0.1988\,\mathrm{s}$ to $8.8374\,\mathrm{s}$ for a latency variation from $5\,\mathrm{ms}$ till $100\,\mathrm{ms}$, i.e., the normalized settling time varies from 1 till 44.45. As we are trying to minimize the maximum among all the normalized settling times, even an end-to-end latency slightly higher than the minimum for such latency-sensitive applications can lead to a very high objective value. For several highly constrained problem instances, Flexi may not be able to find a solution where all such applications are implemented using a very short end-to-end latency. Therefore, we get a few outliers in Figure 8. Nevertheless, the median values for Flexi CP for all sets range from 0.5% for Set 5 to 38% for Set 3. We can see that Flexi CP slightly outperforms Flexi ILP in terms of the objective value. Thus, *Flexi CP is more efficient than Flexi ILP, as it shows better results faster*.

Additionally, whereas for smaller Sets 1, 2, and 3 the average relative difference grows, for larger Sets 4 and 5, it decreases. For these sets, Optimal CP starts performing poorly, which is demonstrated by the presence of negative difference values. This is because it hits the time out and has to use a sub-optimal solution.

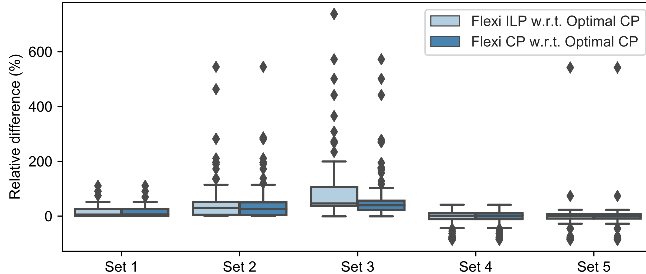


Fig. 8. The relative difference in objective values of Flexi CP and Flexi ILP compared to Optimal CP on Sets 1-5 with a time limit of 3,000 seconds. Higher values mean that Optimal CP is better.

Optimal CP shows the best objective value when there is no restriction on time. However, since some outliers can take weeks to solve till optimality, one needs to set a time limit. In reality, other parts of the development team may be dependent on the schedule, and it may have to be recomputed several times as requirements change, as they always do. The limit hence cannot be too high. A low limit may also be needed to explore many potential system configurations during design-space exploration.

To justify the independence of the results on the time limit, we run Optimal CP and Flexi for Set 5 until the first feasible solution is found, which takes on average 1,533 and 330 seconds, respectively. Flexi results in on average 14% better objective value than Optimal CP. This demonstrates the clear advantage of Flexi CP over Optimal CP for larger problem instances. Moreover, we run Optimal CP on Set 5 with a time limit of 12 hours, which resulted in an insignificant change in the objective value relative to the value found after 3,000 seconds (less than 5% on average).

Finally, *we conclude that the CP formulation scales significantly better than the ILP formulation, both for optimal and heuristic solutions. Additionally, Flexi runs many times faster than Optimal CP, while obtaining solutions with reasonable control performance degradation. However, we leave the exploration of how alternative application and network topologies influence the results of optimal and heuristic approaches as future work.*

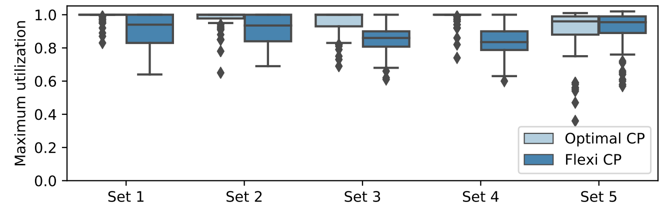


Fig. 9. Maximally achievable utilization obtained via Optimal CP and Flexi CP for Sets 1-5.

3) Comparison of Maximum Achievable Utilization: To show that the proposed approaches can handle non-trivial problem instances with high utilization, we evaluate their ability to find a feasible solution with growing maximum resource utilization among all resources. We also compare the Flexi CP and Optimal CP approaches. The metric is the maximum utilization for which the problem instance is still schedulable. The experimental settings are described in [25].

Figure 9 shows the distribution of the maximum utilization values for Optimal CP and Flexi CP that were cut off after the time limit of 3,000 seconds. The difference in the maximum utilization is on average 9, 7, 10, 12, and -1% for problem instances from Sets 1 to 5, respectively. Note that for Sets 3, 4, and 5, there are certain instances for which Flexi could find a feasible solution while Optimal CP failed to determine the feasibility within the time limit. Due to the time limit, the heuristic yields the maximum achievable utilization 1% better than Optimal CP for the largest set of instances.

Thus, *Flexi exhibits a small degradation in the maximum achievable utilization compared to the Optimal CP formulation for smaller sets, whereas for the largest set, it achieves better utilization than the Optimal CP formulation with the given time limit*.

C. Engine Management System Case Study

We proceed by demonstrating the practical applicability of our proposed heuristic and optimal approaches on an Engine Management System (EMS). This system is responsible for controlling the time and amount of air and fuel injected by the engine, and it is one of the most sophisticated units in a car. It comprises thousands of tasks interacting over tens of thousands of variables. A detailed description of such an application is presented by Bosch in [20], along with a problem instance generator that creates input EMS models according to the characterization.

We assume an automotive architecture similar to the one in Figure 1(a) and the time granularity is 1 μs. Moreover, we consider ECUs to be similar to an Infineon AURIX Family TC27xT with a processor frequency of 125 MHz and time-triggered Ethernet with a bandwidth of 1 Gb/s. Finally, the control performance values for the applications are obtained by simulating the control dynamics of a given application in MATLAB, i.e., in the same way as reported previously in this section for the synthetic problem instances.

We consider such a generated EMS problem instance with 17 applications comprising 2,000 tasks with periods 1, 2, 5, 10, 20, 50, 100, 200 and 1,000 ms and 33,693 messages, for a total of 348,458 occurrences. The target platform consists of 20 ECUs connected by 3 switches, in total 64 resources, whereas the resulting utilization ranges from 14 to 91% with an average utilization around 40%. We run the proposed approaches on the considered scheduling problem. Here, due to the significantly higher complexity of the problem instance, ILP failed to even generate the model with the available 128 GB of memory. In contrast, Flexi CP found a solution in 2 hours with a criterion of 1.6. Optimal CP obtained the first feasible solution with criterion 2.1 after 6 hours, and had found a solution with the criterion value 1.99 when the time-out was triggered after 2 days. This reduction of time from 2 days to 2 hours can be useful during design-space exploration phase when multiple designs can be evaluated in a more reasonable fashion.

Thus, the experimental results on the EMS case study shows that *on large problem instances Flexi can be significantly more efficient than Optimal CP as it finds a solution of better quality faster. Finally, the ILP formulation is significantly less efficient than the CP formulation both in terms of computation time and criterion value on problem instances of all sizes and need not be considered further.*

VII. Related Work

Scheduling of control applications with the goal of control performance optimization has drawn significant attention in recent times [2], [31], [32].

To guarantee the required performance using time-triggered scheduling of control applications, zero jitter and fixed end-to-end latency have been the typical assumptions [32]. Moreover, zero-jitter scheduling and bounded end-to-end latency are also considered for real-time applications in, e.g., [23] and [34]. In [27], jitter-constrained scheduling has been considered and it has been shown that strict jitter requirements may result in a significant underutilization of system resources.

The schedule integration problem has been also studied in the literature, in which multiple applications sharing resources are integrated into one system after being developed separately. In [33], Sagstetter et al. have proposed to integrate schedules of applications while keeping the end-to-end latency of the applications unchanged. In [5], new applications are integrated considering a set of preconfigured applications on the resources. Here, the new applications are scheduled with bounded end-to-end latency, while the schedules of existing applications are fixed. These approaches do not consider control applications and the impact of schedule integration on control performance. If they are applied to integrate control applications, the control performance might not deviate a lot, however, the integration problem might become infeasible for resource-constrained systems owing to inflexible end-to-end latency of the applications.

Furthermore, there have been works that address the co-scheduling of tasks and messages in TTEthernet-based distributed systems. Whereas Craciunas and Oliver in [11] minimize the end-to-end latency of applications, Zhang et al. in [37] propose multi-objective optimization minimizing the end-to-end latencies and response times of applications. Note that unlike our work, they do not consider control performance optimization. To the best of our knowledge, this is the first article that addresses control performance optimization during time-triggered scheduling of tasks and messages while considering different sensitivity of applications to their end-to-end latencies and relaxing jitter constraints on messages.

VIII. Conclusion

This article presents two approaches to determine a feasible time-triggered schedule configuration for control applications. The proposed approaches aim to minimize the control performance degradation of the applications due to resource sharing. In this context, we studied how control performance varied with varying end-to-end latency of an application.

We first proposed a constraint programming (CP) formulation. Further, we devised a two-stage heuristic approach called *Flexi* to solve larger problem instances faster than the CP approach.

To demonstrate the practical relevance of our techniques, we applied the proposed approaches to a case study involving an automotive engine management system. The schedule obtained by Flexi within 6 hours results in a 20% better control performance when compared to the schedule synthesized by the CP solver after running for 48 hours. Thus, it supports the conclusion that over limited time horizons, *Flexi* shows better results than the optimal CP approach on larger problem instances.

Acknowledgments

This work was supported by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466) and by the Netherlands Organisation for Applied Scientific Research TNO.

References

[1] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[2] A. Aminifar, S. Samii, P. Eles, Z. Peng, and A. Cervin. Designing high-quality embedded control systems with guaranteed stability. In *Real-Time Systems Symposium (RTSS)*, 2012.

[3] K. J. Åström and B. Wittenmark. *Computer-controlled systems: theory and design*. Courier Corporation, 2013.

[4] M. Balszun, D. Roy, L. Zhang, W. Chang, and S. Chakraborty. Effectively utilizing elastic resources in networked control systems. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

[5] S. Beji, A. Gherbi, J. Mullins, and P.-E. Hladik. Model-driven approach to the optimal configuration of time-triggered flows in a TTEthernet network. In *International Conference on System Analysis and Modeling (SAM)*, 2016.

[6] M. Bullynck. Modular arithmetic before CF Gauss: Systematizations and discussions on remainder problems in 18th-century Germany. *Historia Mathematica*, 36(1):48–72, 2009.

[7] Y. Cai and M. Kong. Nonpreemptive scheduling of periodic tasks in uni-and multiprocessor systems. *Algorithmica*, (15):572–599, 1996.

[8] A. Cervin. Stability and worst-case performance analysis of sampled-data control systems with input and output jitter. In *American Control Conference (ACC)*, 2012.

[9] S. Chakraborty, M. A. A. Faruque, W. Chang, D. Goswami, M. Wolf, and Q. Zhu. Automotive cyber-physical systems: A tutorial introduction. *IEEE Design and Test*, 33(4):92–108, 2016.

[10] S. Chakraborty, M. D. Natale, H. Falk, M. Lukasiewycz, and F. Slomka. Timing and schedulability analysis for distributed automotive control applications. In *International Conference on Embedded Software (EMSOFT)*, 2011.

[11] S. S. Craciunas and R. S. Oliver. Combined task- and network-level scheduling for distributed time-triggered systems. *Real-Time Systems*, 52(2):1–40, 2015.

[12] D. Goswami, R. Schneider, A. Masrur, M. Lukasiewycz, S. Chakraborty, H. Voit, and A. Annaswamy. Challenges in automotive cyber-physical systems design. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2012.

[13] Gurobi Optimization. Gurobi optimizer reference manual, 2018.

[14] M. Hu, J. Luo, Y. Wang, and B. Veeravalli. Scheduling periodic task graphs for safety-critical time-triggered avionic systems. *Transactions on Aerospace and Electronic Systems*, 51(3):2294–2304, 2015.

[15] IBM ILOG CPLEX. 12.2 user manual. 2010.

[16] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Real-Time Systems Symposium*, 1991.

[17] P. Kampstra et al. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Sftware*, 28(1):1–9, 2008.

[18] H. Kopetz. Time-triggered real-time computing. *Annual Reviews in Control*, 27(1):3–13, 2003.

[19] J. Korst, E. Aarts, J. Lenstra, and J. Wessels. Periodic multiprocessor scheduling. In *Parallel Architectures and Languages Europe*, volume 505. 1991.

[20] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmark for free. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems*, 2015.

[21] H.-T. Lim, L. Völker, and D. Herrscher. Challenges in a future IP/Ethernet-based in-car network for real-time applications. In *Design Automation Conference (DAC)*, 2011.

[22] M. Lukasiewycz, M. Glaß, J. Teich, and S. Chakraborty. Exploration of distributed automotive systems using compositional timing analysis. In *Embedded Systems Development, From Functional Models to Implementations*, pages 189–204. Springer, 2014.

[23] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty. Modular scheduling of distributed heterogeneous time-triggered automotive systems. In *Asia and South Pacific Design Automation Conference*, 2012.

[24] M. Lunt. E/E-architecture in a connected world. https://www.asam.net/index.php?eID=dumpFile&t=f&f=798&token=148b5052945a466cacfe8f31c44eb22509d5aad1, 2017.

[25] A. Minaeva. *Scalable Scheduling Algorithms for Embedded Systems with Real-Time Requirements*. PhD thesis, Czech Technical University in Prague, 2019.

[26] A. Minaeva. Source codes for Periodic Scheduling with Control Performance. https://github.com/CTU-IIG/PSCP, 2020.

[27] A. Minaeva, B. Akesson, Z. Hanzlek, and D. Dasari. Time-triggered co-scheduling of computation and communication with jitter requirements. *IEEE Transactions on Computers*, 67(1):115–129, 2018.

[28] P. Mundhenk, G. Tibba, L. Zhang, F. Reimann, D. Roy, and S. Chakraborty. Dynamic platforms for uncertainty management in future automotive E/E architectures. In *Design Automation Conference (DAC)*, 2017.

[29] A. Novak, Z. Hanzalek, and P. Sucha. Scheduling of safety-critical time-constrained traffic with F-shaped messages. In *International Workshop on Factory Communication Systems (WFCS)*, 2017.

[30] P. Pazzaglia, L. Pannocchi, A. Biondi, and M. Di Natale. Beyond the weakly hard model: Measuring the performance cost of deadline misses. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[31] D. Roy, W. Chang, S. K. Mitter, and S. Chakraborty. Tighter dimensioning of heterogeneous multi-resource autonomous CPS with control performance guarantees. In *Design Automation Conference (DAC)*, 2019.

[32] D. Roy, L. Zhang, W. Chang, D. Goswami, and S. Chakraborty. Multi-objective co-optimization of FlexRay-based distributed control systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[33] F. Sagstetter, S. Andalam, P. Waszecki, M. Lukasiewycz, H. Stähle, S. Chakraborty, and A. Knoll. Schedule integration framework for time-triggered automotive architectures. *Design Automation Conference*, 2014.

[34] K. Schild and J. Würtz. Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357, 2000.

[35] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegla, and G. Mühl. ILP-based joint routing and scheduling for time-triggered networks. In *International Conference on Real-Time Networks and Systems (RTNS)*, 2017.

[36] W. Steiner, G. Bauer, B. Hall, and M. Paulitsch. Time-triggered Ethernet. In *Time-Triggered Communication*, pages 209–248. CRC Press, 2011.

[37] L. Zhang, D. Goswami, R. Schneider, and S. Chakraborty. Task- and network-level schedule co-synthesis of Ethernet-based time-triggered systems. *Asia and South Pacific Design Automation Conference*, 2014.

[38] L. Zhang, D. Roy, P. Mundhenk, and S. Chakraborty. Schedule management framework for cloud-based future automotive software systems. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2016.

[39] L. Zhang, R. Schneider, A. Masrur, M. Becker, M. Geier, and S. Chakraborty. Timing challenges in automotive software architectures. In *International Conference on Software Engineering (ICSE)*, 2014.