Time-Triggered Co-Scheduling of Computation and Communication with Jitter Requirements

Anna Minaeva[1], Benny Akesson[2], Zdeněk Hanzálek[1], Dakshina Dasari[3]
[1]Faculty of Electrical Engineering and Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical University in Prague
[2]CISTER/INESC TEC and ISEP
[3]Corporate Research, Robert Bosch GmbH, Germany

***Abstract*—The complexity of embedded application design is increasing with growing user demands. In particular, automotive embedded systems are highly complex in nature, and their functionality is realized by a set of periodic tasks. These tasks may have hard real-time requirements and communicate over an interconnect. The problem is to efficiently co-schedule task execution on cores and message transmission on the interconnect so that timing constraints are satisfied. Contemporary works typically deal with zero-jitter scheduling, which results in lower resource utilization, but has lower memory requirements. This article focuses on jitter-constrained scheduling that puts constraints on the tasks jitter, increasing schedulability over zero-jitter scheduling.**

The contributions of this article are: 1) Integer Linear Programming and Satisfiability Modulo Theory model exploiting problem-specific information to reduce the formulations complexity to schedule small applications. 2) A heuristic approach, employing three levels of scheduling scaling to real-world use-cases with 10000 tasks and messages. 3) An experimental evaluation of the proposed approaches on a case-study and on synthetic data sets showing the efficiency of both zero-jitter and jitter-constrained scheduling. It shows that up to 28% higher resource utilization can be achieved by having up to 10 times longer computation time with relaxed jitter requirements.

***Cite as*: Anna Minaeva, Benny Akesson, Zdeněk Hanzálek, Dakshina Dasari, Time-Triggered Co-Scheduling of Computation and Communication with Jitter Requirements, *IEEE Transactions on Computers*, 0018-9340, 10.1109/TC.2017.2722443.**

***Source code*: https://github.com/CTU-IIG/CC_Scheduling_WithJitter**

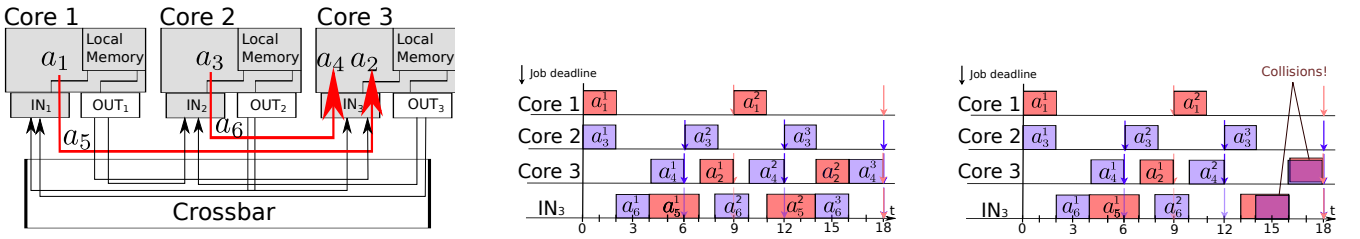
***Index Terms*—Schedules; Jitter; Processor scheduling; Job-shop scheduling; Resource management; Ports (Computers)**

I. Introduction

The complexity of embedded application design is increasing as a multitude of functionalities is incorporated to address growing user demands. The problem of *non-preemptive co-scheduling* of these applications on multiple cores and their communication via an interconnect can be found in automotive [4], [43], avionics [20] and other industries. For instance, automotive embedded systems (e.g. contemporary advanced engine control modules) are highly complex in nature, and their functionality is realized by a set of tightly coupled periodic tasks with *hard real-time requirements* that communicate with each other over an interconnect. These tasks may be activated at different rates and execute sensing, control and actuation functions. Additionally, these embedded applications are required to realize many end-to-end control functions within predefined time bounds, while also executing the constituent tasks in a specific order. To reduce the cost of the resulting system, it is necessary to allocate resources efficiently.

The considered problem is illustrated in Figure 1a, where tasks a_1, a_2, a_3, a_4 are mapped to Cores 1 to 3, where each core has its local memory and communicate via a crossbar switch. This architecture is inspired by Infineon AURIX TriCore [1]. The crossbar switch is assumed to be a point-to-point connection that links an output port of each core with input ports of the remaining cores. Although there is *no contention on output ports* since tasks on cores are statically scheduled, *scheduling of the incoming messages on the input ports* must be done to prevent contention. Moreover, there are two chains of dependencies, indicated by thicker (red) arrows, i.e. $a_1 \rightarrow a_5 \rightarrow a_2$ and $a_3 \rightarrow a_6 \rightarrow a_4$. Note that although this example contains 6 resources to be scheduled, the only input port that must be scheduled in this case is the one of Core 3, since there are no incoming messages to other cores.

The time-triggered approach, where the schedule is computed offline and repeated during execution is commonly used in scheduling safety-critical systems. However, contemporary time-triggered works mostly consider *zero-jitter (ZJ) scheduling* [11], [40] also called strictly periodic scheduling, where the start time of an *activity*, (i.e., task or message) is at a fixed offset (instant) in every period. If there are two consecutive periods in which the activity is scheduled at different times (relative to the period), we call it *jitter-constrained (JC) scheduling*. On one hand, ZJ scheduling results in lower memory requirements, since the schedule takes less space to store and typically needs less time to find an optimal solution. On the other hand, it puts too strict requirements on a schedule causing many problem instances to be infeasible, as we later show in Section VI. This may lead to increasing requirements on the number of employed cores for the given application, and thus makes the system more expensive. Even though some applications or even systems are restricted to being ZJ, e.g. some systems in the avionics domain [3], many systems in the automotive domain allow JC scheduling [16], [36]. Therefore, this article explores the trade-off between JC and ZJ scheduling. Although not all activities have ZJ requirements, some of them are typically sensitive to the delay between consecutive occurrences, since it greatly influences the quality of control [13]. Assuming



(a) Problem description (b) Schedule with jitter-constrained (c) Schedule with zero-jitter

Fig. 1. Multi-periodic scheduling problem description with examples of ZJ and JC solution, where a_5 is a message between a_1 and a_2 and a_6 is a message between a_3 and a_4.

constrained jitter instead of ZJ scheduling allows the resulting schedule to both satisfy strict jitter requirements of the jitter-critical activities and to have more freedom to schedule their non-jitter-critical counterpart.

An example of the JC schedule for the problem in Figure 1a assumes that activities a_1, a_2 and a_5 have a required period of 9 time units, while a_3, a_4, and a_6 must be scheduled with a period of 6. The resulting JC schedule is shown in Figure 1b with a hyper-period (length) of 18 time units, which is the least common multiple of both periods. Hence, activities a_1, a_2 and a_5 are scheduled 2 times and activities a_3, a_4, and a_6 are scheduled 3 times during one hyper-period, defining its number of *jobs*, i.e. activity occurrences. Note that activities a_2 and a_5 are not scheduled with zero-jitter since a_2 in the first period is scheduled at time 7, while in the second period at time 5 (+9). Similarly, a_5 is scheduled at different times in the first and second periods (4 and 2(+9), respectively). In contrast, Figure 1c illustrates that using ZJ scheduling results in collisions between a_2 and a_4 on Core 3, and between a_5 and a_6 in the crossbar switch. Moreover, an exact approach (see SMT formulation in Section IV) can prove that the instance is infeasible with ZJ scheduling. Thus, if an application can tolerate some jitter in the execution of activities a_2 and a_5 without unacceptable quality degradation of control, then the system resources could be utilized more efficiently, as shown in Figure 1b.

The three main contributions of this article are: 1) Two models, one Integer Linear Programming (ILP) formulation and one Satisfiability Modulo Theory (SMT) model with problem-specific improvements to reduce the complexity and the computation time of the formulations. The two models are proposed due to significantly different computation times on problem instances of low and high complexity, respectively. The formulations optimally solve the problem for smaller applications with up to 50 activities in reasonable time. 2) A heuristic approach, called 3-LS, employing a three-step approach that scales to real-world use-cases with more than 10000 activities. 3) An experimental evaluation of the proposed solution for different jitter requirements on a synthetic data sets that quantifies the computation time and resource utilization trade-off and shows that relaxing jitter constraints allows to achieve on average up to 28% higher resource utilization for the price of up to 10 times longer computation time. Moreover, the 3-LS heuristic is demonstrated on a case study of an engine management system, which it successfully solves in 43 minutes.

The rest of this article is organized as follows: the related work is discussed in Section II. Section III proceeds by presenting the system model and the problem formulation. The description of the ILP and SMT formulations and their computation time improvements follow in Section IV. Section V introduces the proposed heuristic approach for scheduling periodic activities, and Section VI proceeds by presenting the experimental evaluation before concluding the article in Section VII.

II. Related Work

There are two approaches to solve the periodic scheduling problem with hard real-time requirements: 1) *Event-Triggered (ET) Scheduling* [12], where scheduling is performed during run-time of a system, triggered by events, and 2) The *Time-Triggered (TT) Scheduling* that builds schedules offline that are provably correct by construction. The TT scheduling is commonly adopted in safety-critical systems, due to the highly predictable behavior of the scheduled activities, simplifying design and verification [27].

Even though this article targets the TT approach, the survey of related work would not be complete without mentioning articles that consider the ET paradigm. A broad survey of works related to periodic (hard real-time) scheduling is provided by Davis and Burns in [12]. Next, Baruah et al. [6] introduce the notion of Pfair schedules, which relates to the concept of ZJ scheduling while scheduling preemptively, i.e. where execution of an activity can be preempted by another activity. Similarly to the ZJ approach that requires the time intervals equal to execution times of activities to be scheduled equidistantly in consecutive periods as a whole, Pfair requires equidistant allocation, while scheduling by intervals of one time instant. On the non-preemptive scheduling front, Jeffay at al. [23] propose an approach to schedule periodic activities on a single resource with precedence constraints. The problem of co-scheduling tasks and messages in an event-triggered manner is considered in [22], [25], [42]. However, these works do not consider jitter constraints, as done in this article.

The TT approach attracted the attention of many researchers over the past twenty years for solving the problem of periodic scheduling. The pinwheel scheduling problem [19] can be viewed as a relaxation of the jitter-bounded scheduling concept, where each activity is required to be scheduled at least once during each predefined number of consecutive time units. If minimizing number of jobs, the solution of the pinwheel problem approaches the ZJ scheduling solution, since it tends to have an equidistant schedule for each activity. Moreover, the Periodic Maintenance Scheduling Problem [5] is identical to ZJ scheduling, as it requires jobs to be executed exactly a predefined number of time units apart.

Considering works that formulate the problem similarly, some authors deal with scheduling only one core [17], [33], while others focus only on interconnects [7], [14]. These works neglect precedence constraints and, in terms of scheduling, consider each core or interconnect to be scheduled independently, unlike the co-scheduling of cores and interconnects in this article. The advantage of co-scheduling lies in synchronization between tasks executing on the cores and messages transmitted through an interconnect that results in high efficiency of the system in terms of resource utilization. Steiner [40] introduces precedence dependencies between activities, while dealing with the problem of scheduling a TTEthernet network. However, Steiner assumes that all activities have identical processing times, which in our case will increase resource utilization significantly.

Some works deal with JC scheduling without any constraints on jitter requirements, which is not realistic in the automotive domain, since there can be jitter-sensitive activities. Puffitsch et al. in [36] assume a platform with imperfect time synchronization and propose an exact constraint programming approach. Abdelzaher and Shin in [2] solve a similar problem by applying both an optimal and a heuristic branch-and-bound method. Furthermore, the authors in [35] consider the preemptive version of our problem that makes it impossible to apply their solution to problem considered in this article, since some activities can be scheduled with interruption.

Jitter requirements are not considered in the problem formulations of [32] and [37], where the authors propose heuristic algorithms to deal with the co-scheduling problem. Finally, in [13] the authors solve the considered problem with an objective to minimize the jitter of the activities using simulated annealing, while we rather assume jitter-constrained activities with no objective to optimize. Note that these approaches with JC assumption are heuristics and the efficiency of the proposed methods have not been compared to optimal solutions.

There also exist works that schedule both tasks and messages, while assuming ZJ scheduling. Lukasiewycz and Chakraborty [29] solve the co-scheduling problem assuming the interconnect to be a FlexRay bus, which results in a different set of constraints. Their approach involves decomposing the initial problem and solving the smaller parts by an ILP approach to manage scalability. Besides, Lukasiewycz et al. in [30] solve the co-scheduling problem by introducing the preemption into the model formulation. Moreover, Craciunas and Oliver [11] consider an Ethernet-based interconnect and solve the problem using both SMT and ILP. However, ZJ scheduling results in a larger number of required cores, as shown in Section VI. In summary, *this work is different in that it is the first to consider the periodic JC co-scheduling problem with jitter requirements and solves it by a heuristic approach, whose quality is evaluated by comparing with the exact solution for smaller instances*.

III. System Model

This section first introduces the platform and the application models used in this article. Then, the mapping of activities to resources is described, concluded by the problem statement.

A. Platform Model

The considered platform comprises a set of homogeneous *cores* on a single multi-core Electronic Control Unit (ECU) with a *crossbar switch*, as shown in Figure 1a. This is similar to the TriCore architecture [1]. The crossbar switch provides point-to-point connection between the cores, and input ports act as communication endpoints and can receive only a single message at a time. We assume that tasks on different cores communicate via the crossbar switch that writes variables in local memories of the receiving cores. On the other side, intra-core communication is realized through reading and writing variables that are stored in the local memory of each core. The set of m resources that include $\frac{m}{2}$ cores and $\frac{m}{2}$ crossbar switch input ports is denoted by $U = \{u_1, u_2, \cdots, u_m\}$. Moreover, the cores are characterized by their clock frequency and available memory capacity.

Although this work focuses on multi-core systems with crossbar switches, the current formulation is easily extensible to distributed architectures with multiple single-core processing units, connected by a bus, e.g. CAN [8]. Furthermore, assuming systems with fully switched networks, e.g. scheduling of time-triggered traffic in TTEternet [39] leads to a similar scheduling problem. However, scalability of the solution presented below may be problematic in such case due to the increased number of entities to schedule, since each message needs to be scheduled on every network segment. The possible extension of this article in this direction can be found in [18], where scheduling of only communication is done unlike the co-scheduling considered in this article.

B. Application Model

The application model is based on characteristics of realistic benchmarks of a specific modern automotive software system, provided in [28]. We model the application as a set of periodic *tasks* T that communicate with each other via a set of *messages* M, transmitted over the crossbar switch. Then $A = T \cup M$, denotes the set of activities, which includes both the incoming messages on the input ports of the crossbar switch and the tasks executed on the cores. Each activity a_i is characterized by the tuple $\{p_i, e_i, jit_i\}$ representing its period, execution time and jitter requirements, respectively. Its execution may not be preempted by any other activity, since *non-preemptive scheduling* is considered. The release date of each activity equals the start of the corresponding period and the deadline is the end of the next period. This deadline prolongation extends the solution space. The period of a message is set to the period of the task that sends the message. Additionally, execution time of messages on the input ports correspond to the time it takes to write the data to the local memory of the receiving core. Thus, since the local memories are defined by both their bandwidth and latency, execution time for each message $a_i \in M$ is calculated as $e_i = \frac{sz_i}{bnd} + lat$, where sz_i is the size of the corresponding transmitted variable given by the application model, while bnd is the bandwidth of the memory and lat is its latency given by the platform model. This is similar to latency-rate server concept [31].

Cause-effect chains are an important part of the model. A cause-effect chain comprises a set of activities that must be executed in a given order within predefined time bounds to guarantee correct behavior of the system. As one activity can be a part of more than one cause-effect chain, the resulting dependency relations are represented by a directed acyclic graph (DAG) that can be very complex in real-life applications [34],

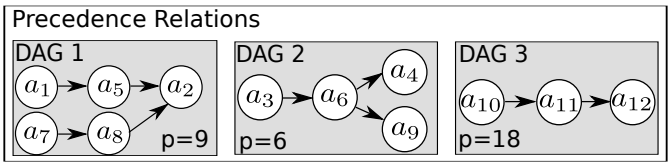


Fig. 2. An example of the resulting precedence relations, where activities in DAG 1 have period 6, activities in DAG 2 have period 9 and activities in DAG 3 have period 18.

such as automotive engine control. Similarly to [11] and [21], activities of one cause-effect chain are assumed to have the same period. More generalized precedence relations with activities of distinct periods being dependent on each other can be found in e.g. [15] . Thus, the resulting graph of precedence relations consists of distinct DAG's for activities with different periods, although not necessarily only one DAG for each unique period. The set of *precedence relations* between the activities is characterized by an adjacency matrix with elements $A_G = \{g_{i,l}\}$ of dimension $n \times n$ where $g_{i,l} = 1$ if activity a_i must finish before activity a_l can start. An example of a precedence relation is shown in Figure 2, which includes the activities from Figure 1. Note that many activities may not have any precedence constraints, since they are not part of any cause-effect chain. For instance, it could be simple logging and monitoring activities.

Lastly, each cause-effect chain has an *end-to-end deadline* constraint, i.e. the maximum time that can lapse from the start of the first activity till the end of the execution of the last activity in each chain equal to two corresponding periods. However, as the first activity in each chain can be scheduled at the beginning of the period at the earliest and the last activity of the chain at the end of the next period at the latest, the end-to-end latency constraint is automatically satisfied due to release and deadline constraints of the activities. Therefore, end-to-end latency constraints do not add further complexity to the model.

C. Mapping of Tasks to Cores

The mapping $map : A \rightarrow U$, $map = \{map_1, \cdots, map_n\}$ of tasks to cores and messages to the memories is assumed to be given by the system designer, which reflects the current situation in the automotive domain, for e.g. engine control. Note that for the previously discussed extension to fully switched network systems, both mapping and routing (i.e. define path through the network for each message) that respect some locality constraints are necessary. Since mapping influences routing and therefore message scheduling, for such systems it is advantageous to solve the three steps at once, as it is done e.g. in [41].

To get a mapping for the problem instances used to validate the approaches in this article, a simple ILP model for mapping tasks to cores is formulated the following way. The variables $q_{i,j} \in \{0, 1\}$ indicate whether task $i = 1, \cdots, |T|$ is mapped to resource $j = 1, \cdots, \frac{m}{2}$ ($q_{i,j}$ = 1) or not ($q_{i,j}$ = 0). Note that we consider only Cores as resources and tasks on cores as activities while mapping. The mapping tries to balance the load, which is formulated as a sum of absolute values of utilization differences on two consecutive resources in Equation (1). Since the absolute operator is not linear, it needs to be linearized by introducing the load variables $b_j \in \mathbb{R}$ in Equations (2) and (3).

$$\textit{Minimize:} \sum_{j \in 1,..,\frac{m}{2}} b_j \tag{1}$$

subject to:

$$b_j \geq \sum_{i=1,\cdots,|T|} \frac{e_i}{p_i} \cdot q_{i,j} - \sum_{i=1,\cdots,|T|} \frac{e_i}{p_i} \cdot q_{i,j+1},\ j = 1, \cdots, \frac{m}{2} \tag{2}$$

$$b_j \geq \sum_{i=1,\cdots,|T|} \frac{e_i}{p_i} \cdot q_{i,j+1} - \sum_{i=1,\cdots,|T|} \frac{e_i}{p_i} \cdot q_{i,j},\ j = 1, \cdots, \frac{m}{2}. \tag{3}$$

Moreover, each task must be mapped to a single resource, as is stated in Equation (4).

$$\sum_{j=1,\cdots,\frac{m}{2}} q_{i,j} = 1,\ i = 1, \cdots, |T|. \tag{4}$$

Note that this mapping is not considered a contribution of this article, but only a necessary step to provide a starting point for the experiments, since the benchmark generator does not provide the mapping.

D. Problem Statement

Given the above model, the goal is to find a schedule with a hyper-period $H = lcm(p_1, p_2, \cdots, p_n)$ with lcm being the least common multiple function, where the schedule is defined by start times $s_i^j \in \mathbb{N}$ of each activity $a_i \in A$ in each period $j = 1, 2, \cdots, n_i$, where $n_i = \frac{H}{p_i}$. The schedule must satisfy the periodic nature of the activities, the precedence relations and the jitter constraints. The considered scheduling problem can be categorized as *multi-periodic non-preemptive scheduling of activities with precedence and jitter constraints on dedicated resources*.

The formal definition of a zero-jitter schedule is the following:

Definition 1 (Zero-jitter (ZJ) schedule): The schedule is a ZJ schedule if and only if for each activity a_i Equation (5) is valid, i.e. the difference between the start times s_i^j and s_i^{j+1} in each pair of consecutive periods j and $j + 1$ over the hyper-period is the same.

$$s_i^{j+1} - s_i^j = p_i,\ j = 1, 2, \cdots, n_i - 1. \tag{5}$$

Zero-jitter scheduling deals exclusively with ZJ schedules. If for some activity and some periods j and $j + 1$ Equation (5) does not hold in the resulting schedule, we call it *jitter-constrained (JC) schedule*.

The scheduling problem, where a set of periodic activities are scheduled on one resource is proven to be NP-hard in [9] by transforming from the 3-Partition problem. Thus, the problem considered (both ZJ and JC) here is also NP-hard, since it is a generalization of the aforementioned NP-hard problem.

IV. Exact Models

Due to significantly different timing behavior of the models on problem instances with varying complexity (see Section VI), both SMT and ILP models are formulated in this article. Moreover, the NP-hardness of the considered problem justifies using these approaches, since no polynomial algorithm exists

to optimally solve the problem unless P=NP. This section first presents a minimal SMT formulation to solve the problem optimally, then continues with a linearization of the SMT model to get an ILP model. It concludes by providing improvements to both models that exploit problem-specific knowledge, reducing the complexity of the formulation and thus the computation time.

A. SMT Model

The SMT problem formulation is based on the set of variables $s_i^j \in \{1, 2, \cdots, H\}$, indicating a start time of job j of activity a_i. Following the problem statement in Section III-D, we deal with a decision problem with no criterion to optimize. The solution space is defined by five sets of constraints. The first set of constraints is called *release date and deadline constraints* and it requires each activity to be executed in a given time interval of two periods, as stated in Equation (6).

$$\begin{aligned}(j-1)\cdot p_i \le s_i^j \le (j+1)\cdot p_i - e_i,\\ a_i \in A,\ j = 1, \cdots, n_i.\end{aligned} \tag{6}$$

The second set, Constraint (7), ensures that for each pair of activities a_i and a_l mapped to the same resource ($map_i = map_l$), it holds that either a_i^j is executed before a_l^k or vice-versa. These constraints are called *resource constraints*. Note that due to the extended deadline in Constraint (6), the resource constraints must be added also for jobs in the first period with jobs of the last period, since they can collide.

$$\begin{aligned}s_i^j + e_i \le s_l^k \vee s_l^k + e_l \le s_i^j,\\ s_i^1 + e_i + H \le s_l^{n_l} \vee s_l^{n_l} + e_l \le s_i^1 + H,\\ a_i,\ a_l \in A : map_i = map_l,\ j = 1, ..., n_i,\ k = 1, ..., n_l.\end{aligned} \tag{7}$$

For the ZJ case, it is enough to formulate Constraints (7) for each pair of activities only for jobs in the least common multiple of their periods, i.e. $j = 1, \cdots, \frac{lcm(p_i,p_j)}{p_i}$ and $k = 1, \cdots, \frac{lcm(p_i,p_j)}{p_l}$. Moreover, the problem for ZJ scheduling is formulated using n variables. One variable s_i^1 is defined for the first job of each activity and other jobs are simply rewritten as $s_i^j = s_i^1 + p_i \cdot (j-1)$.

The next set of constraints is introduced to prevent situations, when two consecutive jobs of one activity collide. Thus, Constraint (8) introduces precedence constraints between each pair of consecutive jobs of each activity, considering also the last and the first job.

$$\begin{aligned}s_i^j + e_i \le s_i^{j+1},\\ s_i^{n_i} + e_i \le s_i^0 + H,\\ a_i \in A,\ j = 1, \cdots, n_i - 1.\end{aligned} \tag{8}$$

Next, due to the existence of cause-effect chains, *precedence constraints* that are based on the previously mentioned A_G matrix are formulated in Equation (9).

$$\begin{aligned}s_i^j + e_i \le s_l^j,\\ a_i,\ a_l \in A : g_{i,l} = 1,\ j = 1, \cdots, n_i.\end{aligned} \tag{9}$$

The *jitter constraints* can be formulated either in terms of *relative jitter*, where we bound only the difference in start times of jobs in consecutive periods or in terms of *absolute jitter*, bounding the start time difference of any two jobs of an activity. Experiments have shown that defining jitter as absolute or relative does not significantly influence the resulting efficiency. The difference in terms of maximal achievable utilization is less than 1% on average with relative jitter showing higher utilization. Therefore, further in the paper we use the relative definition of jitter. Note that the results for absolute jitter formulation do not differ significantly from the results presented in this article. The formulation of relative jitter is given in Equation (10), where the first constraint deals with jitter requirements of jobs inside one hyper-period and the second one deals with jobs crossing a border between two hyper-periods.

$$\begin{aligned}|s_i^j - (s_i^{j-1} + p_i)| \le jit_i,\\ |s_i^1 + H - p_i - s_i^{n_i}| \le jit_i\\ j = 2, \cdots, n_i :\ j > k,\ a_i \in A.\end{aligned} \tag{10}$$

B. ILP Model

The formulation of the ILP model is very similar to the SMT model described above. The main difference in formulation is caused by the requirement of linear constraints for the ILP model. Thus, since Equations (6), (8) and (9) are already linear, they can be directly used in the ILP model. However, resource Constraints (7) are non-linear and to linearize them, we introduce new set of decision variables that reflect the relative order of each two jobs of different activities:

$$x_{i,l}^{j,k} = \begin{cases}1, & \text{if } a_i^j \text{ starts before } a_l^k;\\ 0, & \text{otherwise.}\end{cases}$$

Therefore, resource constraints are formulated by Equation (11), which ensures that either a_i^j is executed before a_l^k (the first equation holds and $x_{i,l}^{j,k} = 1$) or vice-versa (the second equation holds and $x_{i,l}^{j,k} = 0$). However, exactly one of these equations must always hold due to binary nature of $x_{i,l}^{j,k}$, which prevents the situation where two activities execute simultaneously on the same resource. Note that we use $2 \cdot H$ in the right part of the Constraints, since the maximum difference between two jobs of distinct activities can be maximally $2 \cdot H$ due to release date and deadline constraints.

$$\begin{aligned}s_i^j + e_i \le s_l^k + 2 \cdot H \cdot (1 - x_{i,l}^{j,k}),\\ s_l^k + e_l \le s_i^j + 2 \cdot H \cdot x_{i,l}^{j,k},\\ a_i,\ a_l \in A, j = 1, ..., n_i,\ k = 1, ..., n_l.\end{aligned} \tag{11}$$

Furthermore, to formulate the jitter constraints (10) in a linear form, the absolute value operator needs to be eliminated. As a result, Equation (12) introduces four sets of constraints, two for the jobs inside one hyper-period and two for the jobs on the border.

$$\begin{aligned}s_i^j - (s_i^k + (j-k)\cdot p_i) \le jit_i,\\ s_i^{j+1} - (s_i^j + (j-k)\cdot p_i) \ge -jit_i\\ (s_i^1 + H - p_i) - s_i^{n_i} \le jit_i\\ (s_i^1 + H - p_i) - s_i^{n_i} \ge -jit_i\\ j, k = 1, \cdots, n_i :\ j > k,\ a_i \in A.\end{aligned} \tag{12}$$

Unlike the time-indexed ILP formulation [26], where each variable $y_{i,j}$ indicates that the activity i is scheduled at time

j (having $H \cdot n$ variables), the approach used here can solve problems with large hyper-periods when there are fewer jobs with longer execution time. Hence, it utilizes only $n_{jobs} + \frac{n_{jobs} \cdot (n_{jobs}-1)}{2}$ with $n_{jobs} = \sum_{i=1}^{n} n_i$ variables, which is a fraction of the variables that the time-indexed formulation requires for this problem.

C. Computation Time Improvements

While the basic formulations of the SMT and ILP models were presented previously, four computation time improvements for the models are introduced here in order to reduce the complexity of the formulation and computation time of the solver. Note that the improvements do not break the optimality of the solution.

The first improvement *removes redundant resource constraints*. Due to the release date and deadline constraints (6), it is known that $s_i^j \in [(j-1) \cdot p_i, (j+1) \cdot p_i - e_i]$ and $s_l^k \in [(k-1) \cdot p_l, (k+1) \cdot p_l - e_l]$. Therefore, it is necessary to include resource constraints only if the intervals overlap. This improvement results in more than 20% of the resource constraints being eliminated, reducing the computation time significantly since the number of resource constraints grows quadratically with the number of activities mapped to a given resource.

Instead of setting the release date and deadline Constraint (6), the second improvement provides this information directly to the solver. Thus, each constraint is substituted by setting the lower bound of s_i^j on $(j-1) \cdot p_i$ and the upper bound on $(j+1) \cdot p_i - e_i$. Hence, instead of assuming the variables s_i^j in interval $[1, \cdots, H]$ and pruning the solution space by the periodicity constraints, the solver starts with tighter bounds for each variable. This significantly cuts down the search space, thereby reducing computation time. Due to the different solver abilities for SMT and ILP, this optimization is only applicable to the ILP model.

We can further *refine the lower and upper bounds* of the variables by exploiting the knowledge about precedence constraints, which is the third improvement. For each activity the length of the longest critical path of the preceding and succeeding activities that must be executed *before* and *after* the given activity, t^b and t^a respectively, are computed. First, the values of $\hat{t_i^b}$ and $\hat{t_i^a}$ are obtained by adding up the execution times of the activities in the longest chain of successors and predecessors of the activity a_i, respectively, as proposed by [10]. For the example in Figure 2, assuming the execution times of all activities are equal to 1, $\hat{t_1^b} = 0$, $\hat{t_1^a} = 2$, $\hat{t_6^b} = 1$, $\hat{t_6^a} = 1$, $\hat{t_2^b} = 2$, $\hat{t_2^a} = 0$. Additionally, the bounds can be improved by computing the sum of execution times of all the predecessors, mapped to the same resource, i.e.

$$t_i^b = \max(\sum_{l:\ l \in Pred_i,\ map_l = map_i} e_l, \hat{t_i^b})$$

$$t_i^a = \max(\sum_{l:\ l \in Succ_i,\ map_l = map_i} e_l, \hat{t_i^a}),$$

where $Pred_i$ and $Succ_i$ denote the set of all predecessors and all successors of activity a_i, respectively.

For the example in Figure 2 and a single core, the resulting values are the following: $t_1^b = 0$, $t_1^a = 2$, $t_6^b = 1$, $t_6^a = 2$, $t_2^b = 4$, $t_2^a = 0$. Hence, the lower bound of s_i^j can be refined by adding t_i^b and the upper bound can be tightened by subtracting t_i^a, i.e. $s_i^j \in [(j-1) \cdot p_i + t_i^b, (j+1) \cdot p_i - e_i - t_i^a]$. This can also be used in the first improvement, eliminating even more resource constraints.

The fourth and final improvement *removes jitter constraints* (12) *for activities with no freedom to be scheduled with larger jitter than required*. For instance, for jobs of a_2 from Figure 2 with $e_2 = 1$, $t_2^b = 4$, $t_2^a = 0$ and $p_2 = 9$, there are only 14 instants t, where it can be scheduled, i.e. $t \in \{4, \cdots, 17\}$. If $jit_2 \geq 13$, the jitter constraint can be omitted since the activity can be scheduled only at 14 instants due to the third improvement and it is not possible to have jitter bigger than 13 time units and still respect the periodicity of the activity. We denote by I_i the worst-case slack of the activity, i.e. the lower bound on the number of time instants where activity a_i can be scheduled and we compute it according to Equation (13). Hence, the jitter constraints are only kept in the model if Inequality (14) holds, i.e. the activity has space to be scheduled with larger jitter than required. We refer to an activity satisfying Equation (14) *jitter-critical*. Otherwise, it is a *non-jitter-critical* activity.

$$I_i = p_i - (t^b + t^a + e_i) \tag{13}$$

$$jit_i \leq I_i - 2,\ a_i \in A. \tag{14}$$

Experimental results have shown that even on smaller problem instances with 40-55 activities, the proposed improvements reduce computation time by up to 30 times for ILP model and 12 times for SMT model. Moreover, the first and the third improvements result in the most significant reduction of the computation time. However, when experimentally comparing these two improvements, we see that the behavior is rather dependent on the problem instance characteristics, as both the first and the third improvements can be the most effective on different problem instances.

V. Heuristic Algorithm

Although the proposed optimal models solve the problem optimally, this section introduces a heuristic approach to solve the problem in reasonable time for *larger instances*, possibly sacrificing the optimality of the solution within acceptable limits.

A. Overview

The proposed heuristic algorithm, called *3-Level Scheduling (3-LS)* heuristic, creates the schedule constructively. It assigns the start time to every job of an activity in a given HP. Moreover, it implements 3 levels of scheduling, as shown in Figure 3. The first level inserts activity by activity into the schedule, while removing some of the previously scheduled activities, a_u, if the currently scheduled activity a_c cannot be scheduled. However, in case the activity a_u to be removed had problems being scheduled in previous iterations, the algorithm goes to the second level, where two activities that were problematic to schedule, a_c and a_u are *scheduled simultaneously*. By scheduling these two activities together, we try to avoid problems with a sensitive activity further in the scheduling process. Simultaneous scheduling of two activities means that two sets of start times s_c and s_u are decided for activities a_c and a_u concurrently. The third scheduling level is initiated when

even co-scheduling two activities a_c and a_u simultaneously does not work. Then, the third level starts by removing all activities except the ones that were already scheduled by this level previously and its predecessors. Next, it co-schedules the two problematic activities again. Note that although there may be more than two problematic activities, the heuristic always considers maximally two at once.

Having three levels of scheduling provides a good balance between solution quality and computation time, since the effort to schedule problematic activities is reasonable to not prolong the computation time of the approach and to get good quality solutions. Experimental results show that 94% of the time is spent in the first scheduling level, where the fastest scheduling takes place. However, in case the first level does not work, the heuristic algorithm continues with the more time demanding second scheduling level and according to the experimental results it spends 3% of time in this level. The final 3% of the total computation time is spent in the third scheduling level that prolongs the computation time the most since it unschedules nearly all the activities scheduled before. Thus, three levels of scheduling is a key feature to make the heuristic algorithm cost efficient and yet still able to find a solution most of the time. As seen experimentally in Section VI, it suffices to find a good solutions for large instances within minutes.

Note that the advantage of scheduling all jobs of one activity at a time compare to scheduling by individual jobs lies in the significantly reduced number of entities we need to schedule. Hence, unlike the exact model that focus on scheduling jobs for all of the activities at a time, the 3-LS heuristic approach decomposes the problem to smaller sub-problems for one activity. This implies that the 3-LS heuristic is not optimal and is also a reason why it takes significantly less time to solve the problem.

B. Sub-model

The schedule for a single activity or two activities at the same time, respecting the previously scheduled activities is found by a so called *sub-model*. The sub-model for one activity a_i that is non-jitter-critical (i.e. a_i for which Inequality (14) does not hold) is formulated as follows. The minimization criterion is the sum of the start times of all a_i jobs (Equation (15)). Note that the activity index i is always fixed in the sub-model since it only schedules a single activity at a time.

$$\textit{Minimize:} \sum_{j \in 1..n_i} s_i^j \tag{15}$$

The reasons for scheduling activities as soon as possible are twofold. Firstly, it is done for dependent activities to extend the time interval in which the successors of the activity can be scheduled, thereby increasing the chances for this DAG component to be scheduled. Secondly, scheduling at the earliest instant helps to reduce the fragmentation of the schedule, i.e. how much free space in the schedule is left between any two consecutively scheduled jobs, resulting in better schedulability in case the periods are almost harmonic, i.e. being multiples of each other, which is common in the automotive domain [28].

The start time of each job j can take values from the set

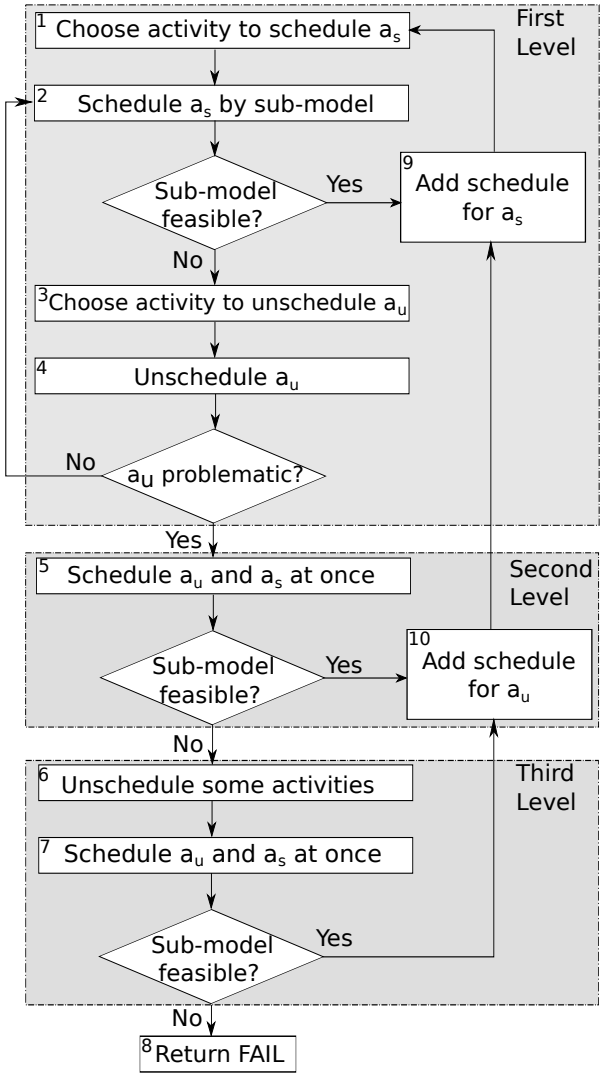


Fig. 3. Outline of 3-Level Scheduling heuristic.

D_i^j (Equation (17)), which is the union of intervals, i.e.

$$D_i^j = \{[l_1^{i,j}, r_1^j] \cup [l_2^{i,j}, r_2^{i,j}] \cup \cdots \cup [l_w^{i,j}, r_w^{i,j}]\},$$
$$r_o^{i,j} < l_{o+1}^{i,j},\ l_o^{i,j} \leq r_o^{i,j},\ j = 1, \cdots, n_i,\ o = 1, \cdots, w-1 \tag{16}$$

and $\{l_1^{i,j}, r_1^{i,j}, \cdots, l_w^{i,j}, r_w^{i,j}\} \in \mathbb{Z}^{2 \cdot w}$, where w is the number of intervals in D_i^j and $l_o^{i,j}, r_o^{i,j}$ are the start and end of the corresponding interval o. This set of candidate start times is obtained by applying periodicity constraints (6) and precedence constraints (9) to already scheduled activities and changing the resulting intervals so that the activity can be executed fully with respect to its execution time. Note that since we insert only activities whose predecessors are already scheduled, all constraints are satisfied if the start time of the job s_i^j belongs to D_i^j.

For the example in Figure 2 with all the execution times equal to 1, with a single core, and with no activities scheduled, $D_2^1 = \{[0 \cdot p_2 + t_2^b; 2 \cdot p_2 - t_2^a - e_2 - 1]\} = \{[0+4; 18-0-1-1]\} = \{[4; 16]\}$, which is basically the application of the third

improvement from Section IV. Now, suppose in the previous iterations a_8^1 is scheduled at time 4 and a_{10}^1 is scheduled at time 6. Then, the resulting $D_2^1 = \{[5;5] \cup [7,16]\}$, since a_2^1 must be scheduled after a_8^1 and it cannot collide with any other activity on the core.

$$s_i^j \in D_i^j,\ j = 1, 2, \cdots, n_i. \tag{17}$$

Furthermore, similarly to the ILP model in Section IV, the precedence constraints for consecutive jobs of the same activity must also be added.

$$\begin{aligned} s_i^j + e_i &\leq s_i^{j+1}, \\ s_i^{n_i} + e_i &\leq s_i^0 + H, \\ j &= 1, 2, \cdots, n_i - 1 \end{aligned} \tag{18}$$

The pseudocode of the sub-model is presented in Algorithm 1. As an input it takes the first activity to schedule a_1, and the optional second activity to schedule a_2 together with their requirements, and the set of intervals D. If a_2 is set to an empty object, the sub-model must schedule only activity a_1. In case a_1 is non-jitter-critical, this scheduling problem can be trivially solved by assigning $s_i^j = l_1^{i,j}$ and checking that it does not collide with the job in the previous period. If it does, we schedule this job at the finish time of the previous job if possible, otherwise at the end of the resource interval it belongs to. If the start time is more than the refined deadline of this job from Section IV-C, the activity cannot be scheduled.

It is clear that this rule will always result in a solution, minimizing (15) if one exists. Moreover, if for some job s_i^j, the interval D_i^j is empty, then there is no feasible assignment of this activity to time with the current set of already scheduled activities. On the other hand, when a_1 is jitter-critical, the

Input: a_1, a_2, D
if a_2 = *NULL* **then**
  if a_1 *is non-jitter-critical* **then**
    $S = \min_{x \in D_i} x$: *Constraint* (18) *holds*;
  else
    $S = ILP(a_1, D)$;
  end
else
  $S = ILP(a_1, a_2, D)$;
end
Output: S

Algorithm 1: Sub-model used by 3-LS heuristic

sub-model is enriched by the set of jitter constraints (12) and the strategy to solve it has to be more sophisticated. The sub-model in this case is solved as an ILP model, which has significantly shorter computation times on easier problem instances in comparison to SMT, as shown experimentally in Section VI. This is important for larger problem instances where sub-model is launched thousands of times, since the heuristic decomposes a large problem to many small problems by scheduling jobs activity by activity. Although this problem seems to be NP-hard in the general case because of the non-convex search space, the computation time of the sub-model is still reasonable due to the relatively small number of jobs of one activity (up to 1000) and the absence of resource constraints.

We formulate Constraint (17) as an ILP in the following way. First, we set $l_1^j \leq s_i^j \leq r_w^j$ defined earlier in this section, and for each $r_t^{i,j}$ and $l_{t+1}^{i,j}$ two new Constraints (19) and a variable $y_{i,j,t} \in \{0, 1\}$ are introduced, which handles the "$\vee$" relation of the two constraints similarly to the variable $x_{i,l}^{j,k}$ from the ILP model in Section IV.

$$\begin{aligned} s_i^j + (1 - y_{i,j,t}) \cdot H &\geq l_{t+1}^{i,j} \\ s_i^j &\leq r_t^{i,j} + y_{i,j,t} \cdot H \end{aligned} \tag{19}$$

Finally, when the sub-model is used to schedule two activities at once, i.e. a_2 is not an empty object, Criterion (15) is changed to contain both activities, and the resource constraints (11) for a_1 and a_2 are added. The resulting problem is also solved as an ILP model, but similarly to the previous case takes rather short time to compute due to small size of the problem. Note that the 3-LS heuristic also utilizes the proposed computation time improvements for the ILP model from Section IV and always first checks non-emptiness of D_j for each job j before creating and running the ILP model.

C. Algorithm

The proposed 3-LS heuristic is presented in Algorithm 2. The inputs are the *set of activities* A, the *priority rule* Pr that states the order in which the activities are to be scheduled and the *rule to choose the activity to unschedule* Un if some activity is not schedulable with the current set of previously scheduled activities. The algorithm begins by initializing the interval set D for each a_i^j as $D_i^j = \{[(j-1) \cdot p_i + t_i^b; j \cdot p_i - t_i^a - e_i - 1]\}$ (line 2). Then it sorts the activities according to the priority rule Pr (line 3), described in detail Section V-D. The rule always states that higher priority must be assigned to a predecessor over a successor, so that no activity is scheduled before its predecessors. Note that the first part of the first level of scheduling is similar to the list scheduling approach [44].

In each iteration, the activity with the highest priority a_c in the *priority queue of activities to be scheduled* Q, is chosen and scheduled by the sub-model (line 7). If a feasible solution S is found, the interval set D is updated so that all precedence and resource constraints are satisfied. Firstly, for each a_l that is mapped to the same resource with a_c, i.e. $map_l = map_c$, the intervals in which a_c is scheduled are taken out of D_l. Secondly, for each successor a_l of activity a_c the intervals are changed as $D_l^j = D_l^j \setminus \{[0, S_j + e_c - 1]\}$, since a successor can never start before a predecessor is completed. Next, the feasible solution is added to the *set of already scheduled activities, represented by their schedules* Sch, and Q is updated to contain previously unscheduled activities, if there is any. If the current activity a_c is not schedulable, at least one activity has to be unscheduled. The activity to be unscheduled a_u is found according to the rule Un and this activity with all its successors are taken out of Sch (line 15). Next, the set of intervals D is updated in the inverse manner compared to the previously described new activity insertion. To prevent cyclic scheduling and unscheduling of the same set of activities, a set R of *activities that were problematic to schedule* is maintained. Therefore, the activity to schedule a_c has to be added to R (line 17) if it is not there yet. In case the activity to be unscheduled is not problematic, i.e. $a_u \notin R$, the algorithm schedules a_c without a_u scheduled in the next iteration. Otherwise, the second level of scheduling takes place, as shown in Figure 3. In this case, the sub-model is called to schedule a_c and a_u simultaneously (line 20) and the set of two schedules S are added to Sch.

```
Input: A
1  Sch = ∅, R = ∅, Scratch = ∅;
2  D.initialize();
3  Q = sort(A, Pr);
4  while |Sch| < |A| do
5      a_c = Q.pop();
6      // Schedule a_c alone
7      S = SubModel(a_c, NULL, D);
8      if SubModel found feasible solution then
9          // Add previously unscheduled activities to Q
10         Q.update();
11         Sch.add(S) ;          // First scheduling level
12         D.update();
13     else
14         a_u = getActivityToUnschedule(Sch, Un);
15         Sch = Sch \ {a_u ∪ a_u.suc};
16         D.update();
17         R.add(a_c);
18         if R.contains(a_u) then
19             // Schedule a_c and a_u simultaneously
20             S = SubModel(a_c, a_u, D) ;
21             if SubModel found feasible solution then
22                 Sch.add(S) ;          // Second level
23             else
24                 // Leave in Sch only activities from
                       Scratch and predecessors of a_c and a_u
25                 Sch = Scratch ∪ a_c.pr ∪ a_u.pr;
26                 D.update();
27                 S = SubModel(a_c, a_u, D);
28                 if SubModel found feasible solution then
29                     Sch.add(S) ;          // Third level
30                     D.update();
31                     Scratch.add(a_c, a_u, a_c.pr, a_u.pr);
32                 else
                       Output: FAIL
33                 end
34             end
35         end
36     end
37 end
Output: Sch
```

Algorithm 2: 3-Level Scheduling Heuristic

Sometimes, even simultaneous scheduling of two problematic activities does not help and a feasible solution does not exist with the given set of previously scheduled activities Sch. If this is the case, we go to the third level of scheduling and try to schedule these two activities almost from scratch, leaving in the set of scheduled activities Sch only the set $Scratch$ of activities *that were previously scheduled in level 3* and the predecessors of a_c and a_u (line 29). The set $Scratch$ is introduced to avoid the situation where the same pair of activities is scheduled almost from scratch more than once, which is essential to guarantee termination of the algorithm. At the third scheduling level, the algorithm runs the sub-model to schedule a_c and a_u with a smaller set of scheduled activities Sch. In case of success, the obtained schedules S are added to Sch (line 29) and a_c together with a_u and their predecessors $a_c.pr$ and $a_u.pr$ are added to the set of activities $Scratch$, scheduled almost from scratch. If the solution is not found at this stage, the heuristics fails to solve the problem. Thus, the 3-LS heuristic proceeds iteration by iteration until either all activities from A are scheduled or the heuristic algorithm fails. Note that the same structure of the algorithm holds for both ZJ and JC cases.

D. Priority and Unscheduling Rules

There are two rules in the 3-LS heuristic: Pr *to set the priority* of insertion and Un *to select the activity to unschedule*. The rule to set the priorities considers information about activity periods P, activity execution times E, the critical lengths of the predecessors execution before t^b and after t^a and the jitter requirements jit. However, not only the jitter requirements of the activity need to be considered, but also the jitter requirements of its successors. The reason is that if some non-jitter-critical activity would precede an activity with a critical jitter requirement in the dependency graph, the non-jitter-critical activity postpones the scheduling of the jitter-critical activity, resulting in the jitter-critical activity not being schedulable. We call this parameter *inherited jitter* of an activity, computed as $jit_i^{inher} = \min_{a_j \in Pred_i} jit_j$. Using the inherited jitter for setting the priority is similar to the concept of priority inheritance [38] in event-triggered scheduling.

Thus, the priority assignment scheme Pr sets the priority of each activity a_i to be a vector of two components $priority_{sched} = (\min(I_i, jit^{inher}), \max(I_i, jit^{inher}))$, where I_i is the worst-case slack of the corresponding DAG, defined in Equation (13). The priority is defined according to lexicographical order, i.e. by comparing the first value in the vector and breaking the ties by the second. We compare first by the most critical parameter, either jitter jit_i^{inher} or the worst-case slack I_i, since those two parameters reflect how much freedom the activity has to be scheduled and the activity with less freedom should be scheduled earlier. This priority assignment strategy considers all of the aforementioned parameters, by definition outperforming the strategies that compare based on only subsets of these parameters.

The rule Un to choose the activity to unschedule is a multi-level decision process. The general rules are that *only activities that are mapped to the resource where activity a_c is mapped are considered* and we do not unschedule the predecessors of a_c. Moreover, the intuition behind the Un rule is that unscheduling activities with very critical jitter requirements or with already scheduled successors should be done only if no other options exist. The exact threshold for being very jitter-critical depends on the size of the problem, but based on experimental results we set the threshold of a high jitter-criticality level to the minimum value among all periods. Thus, whether or not an activity is very jitter-critical is decided by comparing its jitter to the threshold value $thresh = \min_{a_i \in A} p_i$.

The rule Un can hence be described by three steps that are executed in the given order:

1) If there are activities without already scheduled successors and with $jit_i \geq thresh$, choose the one with the highest I_i.
2) If all activities have successors already scheduled, but activities with $jit_i \geq thresh$ exist, we choose the one according to the vector $priority_{unsched}$ = (number of successors scheduled, I_i) comparing lexicographically.
3) Finally, if all activities have $jit < thresh$, the step chooses the activity to unschedule according to the priority vector $priority_{unsched} = (jit^{inher}, I_i)$ comparing lexicographically.

Step 1 is based on the observation that activities with very critical jitter requirements are typically hard to schedule,

unlike those with no jitter requirements or less critical ones. Besides, unscheduling many activities instead of one may cause prolongation of the scheduling process and possibly more complications with further scheduling of successors. Moreover, since only activities of cause-effect chains are a part of precedence relations, there are many activities with no predecessors and successors that can be unscheduled. This is typical for the automotive domain [28]. Step 2 allows unscheduling of activities with already scheduled predecessors, preferring to keep in the schedule activities with critical jitter requirements. Step 3 states that if all of the activities have very critical jitter requirements, the activity with the highest value of inherited jitter should be unscheduled. In all three steps, ties are broken by choosing the activity with higher worst-case slack I value by the same intuition as in the Pr rule.

We have experimentally determined that comparing to the unscheduling rule with only worst-case slack (I_i) considered, the gain of the presented unscheduling rule is 5% more utilization achieved on average.

VI. Experiments

This section experimentally evaluates and compares the proposed optimal models and 3-LS heuristic on synthetic problems with jitter requirements set differently to show the benefits of JC scheduling in terms of utilization. Furthermore, we quantify the trade-off of additional cost in terms of memory to store the schedule and increase in computation time versus this gained utilization. Note that the goal of this section is to show the advantages and disadvantages of the JC approach. The experimental setup is presented first, followed by experiments that evaluates the proposed exact and heuristic approaches for different jitter and period requirements. We conclude by demonstrating our approach on a case study of an Engine Management System with more than 10000 activities to be scheduled.

A. Experimental Setup

Experiments are performed on problem instances that are generated by a tool developed by Bosch [28]. There are five sets of 100 problem instances, each set containing 20, 30, 50, 100 and 500 tasks, respectively. The same problem instance is presented with different jitter requirements. The generation parameters for each dataset are presented in Table I, and the granularity of the timer is set to be 1 μs. Message communication times are computed for the considered platform with the following parameters: bandwidth $bnd = 400$ MB/s and latency $lat = 50$ clock cycles.

The mapping is found as described in Section III-C so that load is balanced across the cores, i.e. the resulting mapping utilizes all cores approximately equally. The resulting problem instances contain 30-45, 50-65, 90-130, 180-250 and 1500-2000 activities (tasks and messages) for sets with 20, 30, 50, 100 and 500 tasks, respectively.

While we initially assume a system with 3 cores connected over a crossbar (resulting in 6 resources), inspired by the Infineon Aurix Tricore Family TC27xT, the approach can scale to a higher number of cores, as shown in Section VI-B4.

The metric for the experiments on the synthetic datasets is the maximum utilization for which the problem instance is still schedulable. The utilization is defined as $r_y = \sum_{a_i \in A: map_i = y} \frac{e_i}{p_i}$ on each resource $y = 1, \cdots, 6$. To achieve the desired utilization on each resource, the execution times of activities are scaled appropriately. The experiments always start from a utilization of 10%, increasing in steps of 1%, solving until the approach is not able to find a feasible solution. The last utilization value for which the solution was found is set as the *maximum utilization* of the approach on the problem instance. Although this approach to set the maximum schedulable utilization may not be completely fair, the utilization is monotonic in most cases. Therefore, we have chosen to approximate the results by setting this rule to get results that are easier to interpret.

TABLE I
Generator parameters for the sets of problem instances

Set	$\|T\|$	*P [ms]*	*Variable accesses per task*	*Chains per task*
1	20	$1, 2, 5, 10$	4	4
2	30	$1, 2, 5, 10$	4	6
3	50	$1, 2, 5, 10, 20, 50, 100$	4	8
4	100	$1, 2, 5, 10, 20, 50, 100$	4	15
5	500	$1, 2, 5, 10, 20, 50, 100$	8	50

Experiments were executed on a local machine equipped with Intel Core i7 (1.7 GHz) and 8 GB memory. The ILP model and ILP part of the 3-LS heuristic were implemented in IBM ILOG CPLEX Optimization Studio 12.5.1 and solved with the CPLEX solver using concert technology, while the SMT model was implemented in Z3 4.5.0. The ILP, SMT and heuristic approaches were implemented in the JAVA programming language.

B. Results

First, the experiments compare the computation time of the two optimal ILP and SMT approaches to show for which problem instances it is advantageous to use each of the approaches. Secondly, we evaluate trade-off between the maximum achievable utilization and computation time of the 3-LS heuristic and the optimal approaches for differently relaxed jitter requirements. Thirdly, since memory consumption to store the final schedule is also a concern, the trade-off between solution quality and required memory is evaluated for systems of different sizes. Finally, a comparison of different period settings is presented to show the applicability of the approach to different application domains and to evaluate the behavior of both ZJ and JC approaches for periods set differently. A time limit of 3 000 seconds per problem instance was set for the optimal approaches to obtain the results in reasonable time. Note that the best solution found so far is used if the time limit is hit.

1) Comparison of the ILP and SMT models with different jitter requirements: First of all, we compare the computation time distribution for Set 1 and Set 2 (of smaller instance sizes with 30-45 activities and 50-65 activities, respectively) for SMT and ILP approaches with jitter requirements of each activity $a_i \in A$ set to $jit_i = \frac{p_i}{2}$, $jit_i = \frac{p_i}{5}$, $jit_i = \frac{p_i}{10}$ and $jit_i = 0$. Since the first problem instance from Set 3 was computing for two days before it was stopped with no optimal solution found for both SMT and ILP models, the experiments with optimal approaches only use the first two sets. We will return

TABLE II
NUMBER OF PROBLEM INSTANCES OPTIMAL APPROACHES FAILED TO SOLVE BEFORE TO TIME LIMIT OF 3 000 SECONDS

jit_i	$p_i/2$		$p_i/5$		$p_i/10$		0	
	Set 1	Set 2	Set1	Set 2	Set 1	Set 2	Set 1	Set 2
ILP	14	76	9	53	6	45	4	27
SMT	2	51	3	13	2	9	2	7

to the larger sets in Section VI-B3 when evaluating the 3-LS heuristic.

The distribution is shown in the form of box plots [24], where the quartile, median and three quartiles together with outliers (plus signs) are shown. Outliers are numbers that lie outside 1.5×the interquartile range away from the top or bottom of the box that are represented by the top and the bottom whiskers, respectively. Note that outliers were also successfully solved within the time limit.

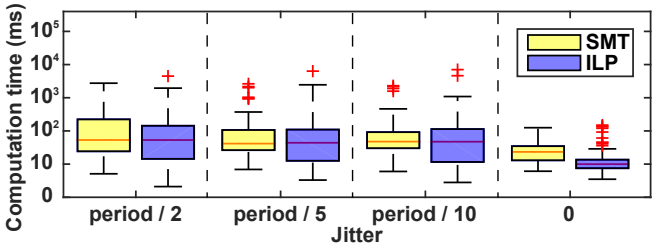


Fig. 4. Computation time distribution for the SMT and ILP models with different jitter requirements for Set 1.

The number of problem instances from Set 1 and Set 2, that the optimal approaches failed to solve within the given time limit is shown in Table II. Moreover, Figure 4 displays the computation time distribution on Set 1, where only problem instances, that both the ILP and SMT solvers were able to optimally solve all jitter requirements within the timeout period are included. For Set 1, it is 82 (out of 100), and for Set 2, it is 21 (out of 100) problem instances. The computation time distribution for Set 2 in show a similar trend, but since the sample is too small to be representative, we do not display them.

The results in Table II show that for more difficult problem instances the SMT model is significantly better than the ILP model in terms of computation time, since it is able to solve more problem instances within the given time limit. On the other hand, the comparison on the problem instances that both approaches were able to solve in Figures 4 indicates that the ILP runs faster on simpler problem instances that can be found at the bottom of the boxplots in Figure VI-B1. As one can see, more relaxed jitter requirements result in longer computation time, which is a logical consequence of having larger solution space.

Thus, *the SMT model is more efficient than the ILP model for the considered problem on more difficult problem instances, while the ILP model shows better results for simpler instances, which justifies the usage of the ILP model in the 3-LS heuristic. Besides, more relaxed jitter requirements cause longer computation time for the optimal approaches*. Therefore, the SMT approach results are used for further comparison with the 3-LS heuristic.

2) Comparison of the optimal and heuristic solutions with different jitter requirements: Figure 5 shows the distribution of the maximum utilization on Set 1 for SMT and 3-LS heuristic with different jitter requirements. For comparison, we use the solution with the highest utilization, while the low value of initial utilization guarantees that at least some solution is found. The time limit caused 3 problem instances in Set 1 not to finish when using the SMT approach and these instances are not included in the results. The results for Set 2 are similar to that of Set 1, but due to small number of solvable instances we do not show them. The results for the optimal approach

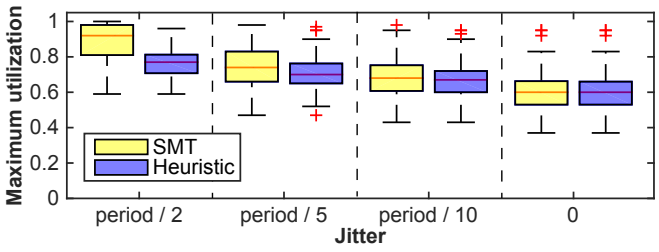


Fig. 5. Maximum utilization distribution for the optimal SMT and 3-LS heuristic approaches with different jitter requirements for Set 1.

shows that stricter jitter requirements cause lower maximum achievable utilization. Namely, the average maximum utilization is 89%, 75%, 69%, 61% for Set 1 and 95%, 81%, 74%, 67% for Set 2 for the instances with jitter requirements equal to half, fifth, tenth of a period and zero, respectively. Meanwhile, the comparison of the 3-LS heuristic to the optimal solution reveals that the average difference goes from 17% and 23% for Set 1 and Set 2, respectively, with the most relaxed $jit_i = \frac{p_i}{2}$ to 0.1% for both sets with ZJ scheduling. This difference for problem instances with more relaxed jitter requirements is caused by very large complexity of the problem solved. Furthermore, while the heuristic solves all problem instances in hundreds of milliseconds, the SMT model fails on 62 problem instances out of 200 within a time limit of 3 000 seconds. This reduction of the computation time by the heuristic is particularly important during design-space exploration, where many different mappings or platform instances have to be considered. In that case, it is not possible to spend too much time per solution.

Hence, we conclude that *heuristic performs better with tighter jitter requirements and hence particularly well for ZJ scheduling, resulting in an average degradation of 7% for all instances. Moreover, unlike the SMT model, the 3-LS heuristic always finds feasible solutions in hundreds of milliseconds, hence providing a reasonable trade-off between computation time and solution quality*.

3) Comparison of the heuristic with ZJ and JC scheduling: While the previous experiment focused on comparing the optimal approach and the heuristic, therefore using only smaller problem instances, this experiment evaluates the 3-LS heuristic on all sets. Due to time restrictions, only two jitter requirements were considered, $jit_i = \frac{p_i}{5}$ and $jit_i = 0$. Figure 6 shows the distribution of the maximum utilization for the 3-LS heuristic on Sets 1 to 5. In all sets, 100 problem instances were used for this graph. The results show that with growing size of the problem instance, the maximum utilization generally increases. The

average difference in maximum utilization of the 3-LS heuristic on the problem instances with JC and ZJ requirements is 15.3%, 9.7%, 8.6%, 4.2% and 7.5% for Sets 1 to 5, respectively, with JC achieving higher utilization. The decreasing difference with growing sizes of the problem is caused by the growing average utilization. For instance, the average maximum utilization for Set 5 is 89.1% for the problem instances with JC requirements and 82.6% for the problem instances with ZJ requirements, pushing how far the maximum utilization for the JC scheduling can go. This tendency of increasing maximum utilization for the ZJ scheduling can be intuitively supported by the fact that more and more activities are harmonic with each other, which results in easier scheduling. In reality, harmonization costs a significant amount of over-utilization, especially when activities with smaller periods are concerned. On problem instances without harmonized activity periods the JC scheduling can show notably better results for larger instances compared to ZJ scheduling, as shown in Section VI-B5.

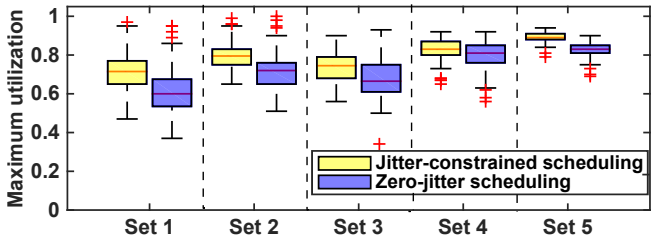


Fig. 6. Maximum utilization distribution for the 3-LS heuristic with jitter-constrained and zero-jitter requirements in sets with 20, 30, 50, 100 and 500 tasks.

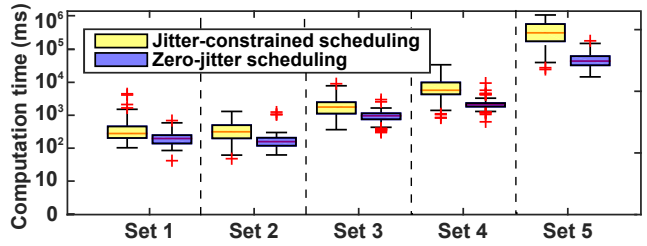


Fig. 7. Computation time distribution for the 3-LS heuristic with jitter-constrained and zero-jitter requirements in sets with 20, 30, 50, 100 and 500 tasks.

Figure 7 shows the computation time of ZJ and JC using the 3-LS heuristic. Similarly to the optimal approach, the 3-LS heuristic takes longer to solve problem instances with JC requirements due to the larger solution space. Specifically, the average computation time for JC heuristic for Sets 1 to 5 are 0.3, 0.6, 3.6, 14.5 and 1003.6 seconds, respectively, while for ZJ scheduling it is 0.15, 0.28, 1.6, 4 and 109 seconds. Thus, solving a problem instance with JC requirements with 1500-2000 activities takes less than 17 minutes on average, which is still reasonable. Hence, *the 3-LS heuristic with JC scheduling provides better results, but requires more time than the 3-LS heuristic with ZJ scheduling*.

To summarize this experiment, JC scheduling *is promising in terms of maximum utilization, as it schedules with up to 55% higher resource utilization*. Besides, the computation time of the proposed heuristic is affordable even for larger problem instances, while the optimal models fail to finish in reasonable time already for much smaller instances. Moreover, *the proposed heuristic solves the problem instances with ZJ requirements near-optimally with a difference of 0.1% in schedulable utilization on average*. Generally, *the JC heuristic provides more efficient solutions than the ZJ heuristic, while requiring longer computation time*.

4) Evaluation of required memory and maximum utilization trade-off with different number of cores: The trade-off between maximum achievable utilization and the amount of memory required to store the schedule is evaluated by this experiment. Figure 8 shows the average maximum utilization achieved on systems with different number of cores and with gradually increasing percentage of JC jobs on 50 problem instances from Set 2 (due to time restrictions). The jitter constraint is set to $jit_i = \frac{p_i}{5}$ and the instances are solved to optimality. Furthermore, the problem instances with different numbers of cores are solved in steps of 5% of jobs with zero-jitter requirement, which reflects how much memory is necessary to store the schedule for such solutions. Note that the execution times of the activities are scaled proportionally to the number of cores so that each resource has a required utilization.

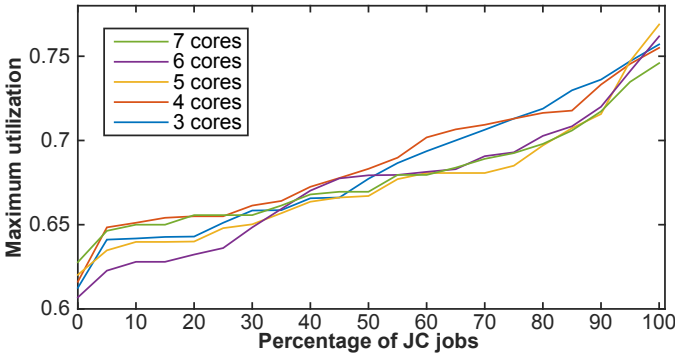


Fig. 8. Utilization distribution for different percents of JC activities for different architectures.

The results show that introducing more JC jobs and thus increasing memory requirements for storing the final schedules can significantly improve the average maximum utilization. Namely, for the architecture with 4 cores, the maximum utilization with all ZJ jobs is 61%, while relaxing jitter requirements of half the jobs results in 69% utilized resources, and relaxation all of the jobs increases the maximum utilization to 76%. Concerning the required memory to store the schedule, the problem instances with 4 cores on average contain 80 jobs with JC scheduling and 49 jobs while scheduling in zero-jitter manner. Thus, assuming we need 8 bytes to store the schedule of one job, the memory overhead of relaxing jitter is 31 * 8 = 248 bytes, which is a reasonable price to pay for utilization gain of 15% on average on each resource.

Concerning the increasing number of cores, the results demonstrate that on average there is no significant dependency on how much cores we have in the system. Hence, *JC scheduling can result in high utilization gain, although at the cost of increased memory requirements to store the resulting schedule*.

5) Comparison of the different period settings: To show that the approach is applicable to other domains, an experiment with different period settings is performed. All problem instances from Set 2 are solved monoperiodically ($p_i = 10$ ms for each activity $a_i \in A$), or with harmonic periods (activities with $p_i = 2$ ms are changed to $p_i = 5$ ms), or with initial periods (i.e. with periods 1, 2, 5, 10 ms), or with non-harmonic periods (with periods 2, 5, 7, 12 ms). Figure 9 displays the average maximum utilization achieved by the 3-LS heuristic with ZJ and JC scheduling ($jit_i = \frac{p_i}{5}$) on 100 problem instances from Set 2. Since the optimal approach was not able to solve 7 out of 10 first instances with non-harmonic periods within the given time limit, due to its complexity and extended hyper-period, the optimal approach results are not included in the figure.

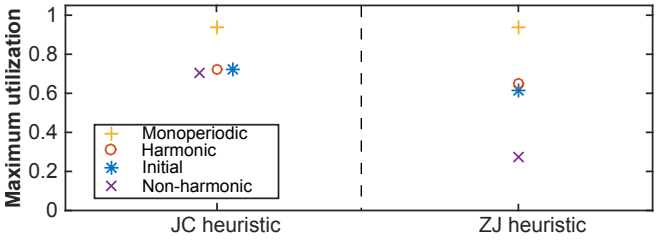


Fig. 9. Utilization distribution for problem instances with different periods.

The results show that the maximum utilization for both ZJ and JC is achieved when scheduling monoperiodically, which is explained by having less possible collisions in the resulting schedule. An interesting observation is that for JC scheduling all other period settings on average resulted in very similar maximum utilization, while the ZJ approach shows the variation of 27% with non-harmonic periods, 62% with initial periods and 65% with harmonic period set. Relative insensitivity of JC scheduling to period variations can be caused by significantly larger solution space due to relaxation of strict jitter constraints. This allows to find solutions with high utilization even with the non-harmonic period setting.

Besides, the same order of computation time distribution is shown by different period settings, i.e. monoperiodic scheduling is the fastest, while the problem instances in the non-harmonic period set result in the longest computation time.

Thus, *the proposed approach is applicable to other domains, where the application periods have different degree of harmonicity. Furthermore, increasing harmonicity of the period set results in higher maximum utilization, lower computation time and lower gain of JC scheduling in comparison with ZJ scheduling in terms of maximum utilization.*

C. Engine Management System Case Study

We demonstrate the applicability of the proposed 3-LS heuristic on an Engine Management System (EMS). This system is responsible for controlling the time and amount of air and fuel injected by the engine by considering the values read by numerous sensors in the car (throttle position, mass air flow, temperature, crankshaft position, etc). By design, it is one of the most sophisticated engine control units in a car consisting of 1000-2000 tightly coupled tasks that interact over 20000 to 30000 variables, depending on the features in that particular variant. A detailed characterization of such an application is presented by Bosch in [28], along with a problem instance generator that creates input EMS models in conformance with the characterization.

We consider such a generated EMS problem instance, comprising 2000 tasks with periods 1, 2, 5, 10, 20, 50, 100, 200 and 1000 ms and with 30000 variables in total, where each task accesses up to 12 variables. There are 60 cause-effect chains in the problem instance with up to 11 tasks in each chain. We consider the target platform to be similar to an Infineon AURIX Family TC27xT with a processor frequency of 125 MHz and an on-chip crossbar switch with a 16 bit data bus running at 200 MHz, thus having a bandwidth of 16-bit x 200 MHz / 8 = 400 MB/s. The time granularity is 1 μs, and the resulting hyper-period is 1000 ms. However, setting the hyper-period to be 100 ms results in a utilization loss of less than 0.5%, arising from shortening the scheduling periods of tasks with periods 200 ms and 1000 ms and over-sampling, which is a reasonable sacrifice to decrease the memory requirements of the schedule. The tool in [28] provides the number of instructions necessary to execute each task, which is used to compute the worst-case execution time with the assumption that each instruction takes 3 clock cycles on average (including memory accesses that hit/miss in local caches).

The mapping of tasks to cores by the simple ILP described in Section III requires minimally 3 cores with the utilization approximately 89.6% on each core and approximately 30% on each input port of the crossbar. Moreover, the resulting scheduling problem has 10614 activities with 104721 jobs for the JC assumptions in total. Neither SMT nor ILP can solve this problem in reasonable time, but the JC heuristic with $jit_i = \frac{p_i}{5}$ for all a_i solves the problem in 43 minutes. By gradually introducing more activities a_i with $jit_i = 0$, we have found a maximum value of 85% ZJ activities for which the 3-LS heuristic is still able to find a solution, which takes approximately 12 hours. Note that the computation time has increased with introducing more ZJ activities due to more restricted solution space. However, to store the schedule in the memory for 0% ZJ jobs, 104721 * 8 = 818 Kbytes of memory is required assuming that one job start time needs 8 bytes, while with 85% ZJ jobs it is only 19394 * 8 = 152 Kbytes. *Thus, for realistic applications the optimal approaches take too long, while the 3-LS heuristic approach is able to solve the problem in reasonable time. Moreover, increasing the percent of ZJ activities has shown to provide a trade-off between computation time and required memory to store the obtained schedule.*

VII. Conclusions

This article introduces a co-scheduling approach to find a time-triggered schedule of periodic tasks with hard real-time requirements that are executed on multiple cores and communicate over an interconnect. Moreover, precedence and jitter requirements are put on these tasks due to the nature of such applications in the automotive domain. To optimally solve the considered problem, we propose both an Integer Linear Programming (ILP) model and Satisfiability Modulo Theory (SMT) model with computation time improvements that exploit problem-specific information to reduce the computation time. Furthermore, a three-step heuristic scheduling approach, called 3-LS heuristic, where the schedule is found constructively

is presented. The heuristic works in three levels, where the scheduling complexity and the time consumption grow for each level, providing a good balance between solution quality and computation time.

We experimentally evaluate the efficiency of the proposed optimal and heuristic approaches with jitter-constrained (JC) requirements, comparing to the widely used zero-jitter (ZJ) approach and quantify the gain in terms of maximum utilization of the resulting systems for the optimal and heuristic approaches. The results show that JC scheduling by the optimal approaches achieves higher utilization with an average difference of 28% compared to optimal ZJ scheduling. Moreover, the experimental evaluations indicate that SMT model is able to solve more problem instances optimally within a given time limit than the ILP model, while the ILP model shows better computation time on simpler problem instances. We also show that the 3-LS heuristic solves the problem instances with ZJ requirements near-optimally. The computation time of the proposed heuristic is acceptable even for larger problem instances, while the optimal models fail to finish in reasonable time already for smaller problem instances. Furthermore, the approach is demonstrated on a case study of an Engine Management System, where 2000 tasks are executed on cores, sending around 8000 messages over the interconnect. Here, we show that for realistic applications, the proposed SMT solution takes too long and the 3-LS heuristic is able to find the solution in reasonable time, providing a trade-off between required memory to store the schedule and computation time depending on percent of activities with zero-jitter requirements.

ACKNOWLEDGMENTS

This work was supported by the European Unions Horizon 2020 research and innovation programme under grant agreement No. 688860 (HERCULES), and Eaton European Innovation Centre. It was also partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); also by FCT/MEC and the EU ARTEMIS JU within project(s) ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2).

REFERENCES

[1] TriCore pipeline behaviour and instruction execution timing, June 2016.

[2] T. F. Abdelzaher and K. G. Shin. Combined task and message scheduling in distributed real-time systems. *IEEE Transactions on parallel and distributed systems*, 10(11):1179–1191, 1999.

[3] A. Al Sheikh, O. Brun, P.-E. Hladik, and B. J. Prabhu. Strictly periodic scheduling in IMA-based architectures. *Real-Time Systems*, 48(4):359–386, 2012.

[4] A. Albert. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. *Embedded World 2004*, pages 235–252, 2004.

[5] A. Bar-Noy, R. Bhatia, J. S. Naor, and B. Schieber. Minimizing service and operation costs of periodic scheduling. *Mathematics of Operations Research*, 27(3):518–544, 2002.

[6] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[7] M. Becker, D. Dasari, B. Nikolic, B. Akesson, V. Nelis, and T. Nolte. Contention-free execution of automotive applications on a clustered many-core platform. *Euromicro Conference on Real-Time Systems*, 2016.

[8] R. Bosch. CAN specification version 2.0, 1991.

[9] Y. Cai and M. Kong. Nonpreemptive scheduling of periodic tasks in uni-and multiprocessor systems. *Algorithmica*, 1996.

[10] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, pages 181–194, 1990.

[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] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):1–44, 2011.

[13] M. Di Natale and J. A. Stankovic. Scheduling distributed real-time tasks with minimum jitter. *Computers, IEEE Transactions on*, 2000.

[14] J. Dvořák and Z. Hanzálek. Using two independent channels with gateway for FlexRay static segment scheduling. *IEEE Transactions on Industrial Informatics*, 12(5):1887–1895, 2016.

[15] M. Forget, E. Grolleau, C. Pagetti, and P. Richard. Dynamic priority scheduling of periodic tasks with extended precedences. *ETFA, 2011 IEEE 16th Conference on*, pages 1–8, 2011.

[16] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. AUTOSAR–a worldwide standard is on the road. *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, 62, 2009.

[17] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. de Dinechin. Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources. *Real-Time Systems*, 52(4):1–51, 2015.

[18] Z. Hanzalek, P. Burget, and P. Sucha. Profinet IO IRT message scheduling with temporal constraints. *Industrial Informatics, IEEE Transactions on*, 6(3):369–380, Aug 2010.

[19] R. Holte, L. Rosier, I. Tulchinsky, and D. Varvel. The pinwheel: A realtime scheduling problem. *Proceedings of the 22 Hawaii International Conference on System Sciences*, pages 693–702, 1989.

[20] Honeywell Aerospace. Application specific integrated circuits based on TTEthernet ready for first orion test flight., May 05/2014.

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

[22] P. Jayachandran and T. Abdelzaher. End-to-end delay analysis of distributed systems with cycles in the task graph. In *ECRTS'09. 21st Euromicro Conference on*, pages 13–22. IEEE, 2009.

[23] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of period and sporadic tasks. *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139, Dec 1991.

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

[25] D.-I. Kang, R. Gerber, and M. Saksena. Parametric design synthesis of distributed embedded systems. *Computers, IEEE Transactions on*, 49(11):1155–1169, 2000.

[26] O. Koné *et al.* Event-based MILP models for resource-constrained project scheduling problems. *Computers & Operations Research*, 38(1), 2011.

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

[28] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmark for free. *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems*, 2015.

[29] M. Lukasiewycz and S. Chakraborty. Concurrent architecture and schedule optimization of time-triggered automotive systems. In *Proceedings IEEE/ACM/IFIP conference*, pages 383–392. ACM, 2012.

[30] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty. Modular scheduling of distributed heterogeneous time-triggered automotive systems. *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 665–670, January 2012.

[31] A. Minaeva, P. Šůcha, B. Akesson, and Z. Hanzálek. Scalable and efficient configuration of time-division multiplexed resources. *Journal of Systems and Software*, 113:44 – 58, 2016.

[32] Y. Monnier, J.-P. Beauvais, and A.-M. Déplanche. A genetic algorithm for scheduling tasks in a real-time distributed system. *Euromicro Conference, 1998. Proceedings. 24th*, pages 708–714, 1998.

[33] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion. Multisource software on multicore automotive ECUs—combining runnable sequencing with task scheduling. *Industrial Electronics, IEEE Transactions on*, 59(10):3934–3942, 2012.

[34] M. Panić, S. Kehr, E. Quiñones, B. Boddecker, J. Abella, and F. J. Cazorla. Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores. *International Conference on Hardware/Software Codesign and System Synthesis*, page 29, 2014.

[35] D.-T. Peng and K. G. Shin. Static allocation of periodic tasks with precedence constraints in distributed real-time systems. *Distributed Computing Systems, 1989.*, pages 190–198, 1989.

[36] W. Puffitsch, E. Noulard, and C. Pagetti. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems*, 51(5):526–565, 2015.

[37] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 6(4):412–420, April 1995.

[38] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175–1185, 1990.

[39] W. Steiner. TTEthernet specification. *TTTech Computertechnik AG, Nov*, 39:40, 2008.

[40] W. Steiner. An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks. *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, 31:375–384, November 2010.

[41] D. Tămaş-Selicean, P. Pop, and W. Steiner. Design optimization of TTEthernet-based distributed real-time systems. *Real-Time Systems*, 51(1), 2015.

[42] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and microprogramming*, 40(2-3):117–134, 1994.

[43] S. Tuohy, M. Glavin, C. Hughes, E. Jones, M. Trivedi, and L. Kilmartin. Intra-vehicle networks: A review. *Intelligent Transportation Systems, IEEE Transactions on*, 16(2):534–535, 2015.

[44] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.