Mapping and Scheduling Automotive Applications on ADAS Platforms using Metaheuristics

Shane D. McLean*, Silviu S. Craciunas†, Emil Alexander Juul Hansen*, Paul Pop*
*Technical University of Denmark Kongens Lyngby, Denmark; paupo@dtu.dk
†TTTech Computertechnik AG, Vienna, Austria; silviu.craciunas@tttech.com

***Abstract*—Modern Advanced Driver-Assistance Systems (ADAS) merge critical and non-critical software functions with complex timing requirements and inter-dependencies onto the same integrated hardware platform. Real-time safety-critical automotive applications feature complex dependency chains between tasks (e.g., performing sensing, processing and actuation) which have to satisfy worst-case end-to-end latency constraints. The resulting scheduling problem requires both the assignment of tasks to the available cores of the platform and the computation static schedule tables for the real-time tasks, such that task deadlines, as well as end-to-end task chain constraints, are satisfied. We propose a heuristic approach based on Simulated Annealing (SA) which creates static schedule tables by simulating Earliest Deadline First (EDF) scheduling parameterized by task offsets and local deadlines decided by SA. We evaluate the proposed solution with real-world and synthetic test cases scaled to fit the future requirements of ADAS systems.**

***Index Terms*—Automotive applications, task scheduling, task preemption, Simulation**

I. Introduction

Modern vehicles integrate a growing number of complex functions, commonly known as Advanced Driver Assistance Systems (ADAS), which provide driver assistance, e.g. automated or assisted parking, lane changing and emergency brake assistance and even fully autonomous driving. The growing trend to migrate more and more of ADAS functions from hardware to software allows modularization within an integrated hardware platform that can be cooperatively used and centrally managed [1].

The fusion of multiple software functions into the same hardware platform has multiple advantages, like reusability and portability, but also several challenges, especially in terms of real-time, testing and safety [2]. This paradigm enables the integration of functions of different criticality levels in a composable manner with guaranteed temporal and spatial isolation such that they can coexist on the same platform without sacrificing real-time capabilities. However, this mixed-criticality paradigm applied to the automotive domain requires new concepts in terms of safety-critical temporal and spatial isolation, new scheduling results and configurations tools, as well as analysis methods for SIL certification (c.f. [1], [3]).

Generally, ADAS platforms are composed of heterogeneous multi-core CPUs and Systems-on-chip (SoCs) of different performance and safety levels, that are interconnected by a (real-time) communication backbone [4], [5]. In such integrated platforms, the ADAS functions have not only complex timing requirements but also feature a complex interdependence between sensors, control software and actuators. For example, one function for driver assistance collects sensor data from both cameras and distance sensors (ultrasonic, LIDAR) into a sensor fusion layer which transmits the data to control algorithms that activate e.g. the emergency brake system. This succession of function execution creates a temporal dependency chain, which has to comply to a set of timing requirements in terms of latency. In order to guarantee both the interdependence and real-time timing behavior, the ADAS functions have to be scheduled in an appropriate manner. Moreover, other less critical systems, like infotainment, are also integrated into the same platform, and must not interfere with the real-time behavior of critical functions.

In this paper[1], we present a heuristic-based scheduling algorithm for ADAS platforms, that takes into account the different dimensions of timing and dependency requirements and is designed with scalability in mind. The optimization algorithm is based on a Simulated Annealing metaheuristic, which takes into account not only the timing constraints but also design goals, such as function allocation on computing units. For the function allocation we use a regular task model in which the WCET changes based on the core speed and the communication happens at the end of task instance execution. Future work may also include the LET model [8] which is becoming increasingly popular in the automotive domain since it can provide deterministic communication behavior.

The scheduling of task sets with dependencies has been a well-studied topic within the real-time community (e.g. [9]–[11]). In the context of ADAS, the complex task chain requirements have been addressed in terms of computing the worst-case end-to-end latency of a chain based on a given schedule (c.f. [12], [13]). In such approaches, the schedule is fixed by the scheduling policy (e.g., fixed priority) and the analysis focuses on determining the worst-case latency, when there is variance in the execution of said tasks. Our approach is different in that it generates schedules, that already adhere to the task chain requirements, which does not necessitate a further analysis, since the real-time requirements are

†The research presented throughout this paper has partially received funding from the European Community's Horizon 2020 programme under the UP2DATE project (grant agreement 871465).

[1]The paper is based on original technical material contained in a thesis of the author [6] which is archived as a technical report [7].

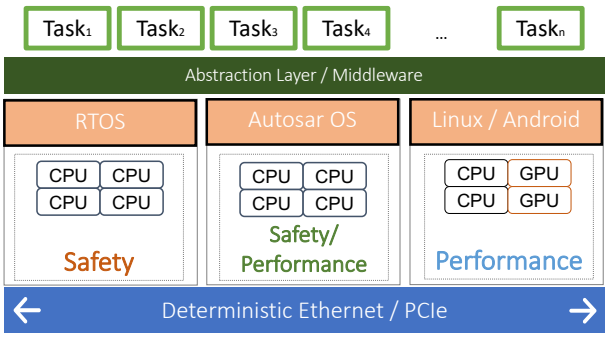


Fig. 1. High-level platform model.

guaranteed by the schedule construction. In [14], the authors present a model-checking based method to compute worst-case response times and end-to-end latencies of tasks, that have chain dependency and communication constraints. In the work presented in [15], the authors introduce a task chain latency analysis, that does not require information about the concrete scheduling algorithm.

To the best of our knowledge, this is the first work to propose a heuristic-based solution to the combined task-to-core assignment and scheduling problem in ADAS platforms that generates schedule tables which respect both task and complex task chain timing constraints.

We start by introducing the platform and application models in Sections II followed by a description of the scheduling problem (Section III). We introduce the algorithm in Section IV followed by an experimental evaluation in Section V and conclude the paper in Section VI.

II. Platform and Application Models

A. System Model

The ADAS hardware platform is a multi-core multi-SoC embedded ECU featuring a variety of CPUs and Graphics Processing Units (GPUs) running at different speeds, which are interconnected through either a deterministic Ethernet backbone (TSN [16], TTEthernet [17]) or through PCIe. RazorMotion [18], for example, features a Renesas RH850P/1H-C ASIL D MCU with lockstep cores running at 240 MHz and two Renesas R-Car H3 ASIL B SoCs with four Cortex A57, four Cortex A53, one Cortex R7, one IMP-X5, and one IMG PowerVR GX6650 GPU.

Figure 1 presents a high-level view of the ADAS platform. Each host can run a different operating system depending on the safety and performance requirements. Each such OS can have a different scheduling policy, ranging from fixed-priority (AUTOSAR [19]) to table-driven or dynamic priority scheduling (typically in safety RTOSes). However, there is a growing tendency to use a table-driven static schedule execution due to the compositionality and isolation properties [20]–[23], i.e. tasks that are already scheduled are not influenced by new tasks being added to the system. In order to provide a common execution environment and hardware abstraction, a middleware layer, e.g. the MotionWise [18] layer, is running on top of each operating system. The middleware layer also ensures portability of software functions to be located according to their execution and safety requirements [1]. Moreover, the middleware layer provides the capability to execute tasks according to a table-driven pre-computed schedule independent of the underlying OS dispatching mechanisms which ensures temporal isolation [22]. Hence, in this paper we focus on creating static schedules for the table-driven dispatching mechanism of such ADAS systems.

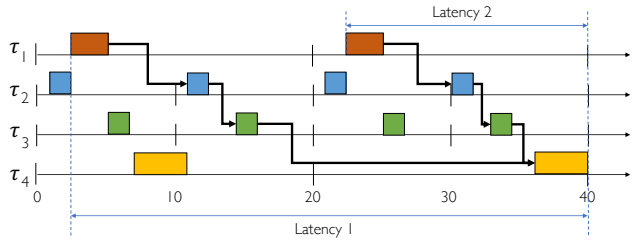


Fig. 2. Task chain example.

We model an ADAS platform as a graph $\mathcal{A}(\mathcal{V},\mathcal{E})$, where each vertex $v_i \in \mathcal{V}$ is a processor and the edges $\mathcal{E}$ are the communication links between the processors. Each processor $v_i \in \mathcal{V}$ has a list of cores $\mathcal{C}_i$.

B. Application Model

On top of this platform a large number of different software functions, implemented by different vendors, must be integrated and deployed. It is crucial that software functions (which may be tested independently) can be integrated with other software functions compositionally. The system is composed of applications (called tasks or runnables), that are either pre-assigned to cores or must be assigned by the scheduling algorithm. Tasks have real-time requirements, both in terms of execution (offset, deadline, jitter) as well as temporal dependencies arising from task chains (defined below). We model the applications as a set of n periodic tasks, $\Gamma = \{\tau_i \mid 1 \leq i \leq n\}$, similar to the model in [24]. A task τ_i is defined by the tuple $(\sigma_i, r_i, \phi_i, C_i, T_i, D_i)$ with σ_i representing the core, C_i denoting the computation time, T_i the period, r_i the earliest release time, ϕ_i the initial offset/displacement of task arrival times and D_i the relative deadline of the task under the assumption that $D_i \leq T_i$. Each real-time task τ_i yields an infinite set of instances (jobs) $\tau_{i,k}, k = 1,2,\ldots$ [25, p. 80]. Tasks can be preempted at any time instant on a timeline with macrotick granularity given by the underlying OS capabilities. If a task τ_i is pre-assigned to a core, then its core σ_i will be given. Otherwise, we decide their assignment to a specific core, in that the σ_i of a task τ_i can take any value from a finite set of core values $\mathcal{C}_i$. The assignment of tasks to cores is captured by the mapping function $M : \Gamma \rightarrow \mathcal{C}_i$.

Currently, tasks cannot migrate at run time, after they have been assigned to a core, but in the future we envision that task migration, when done properly with respect to the deterministic timing behavior, will allow even better resource utilization. The scheduling allows preemption, i.e., a table can

be constructed such that a task is interrupted by another task, and then resumes its execution.

For a given mapping M, we denote the schedule table with $\mathcal{S}$. In this table, each task τ_i has a list of offsets Φ_i on its core σ_i. The first offset in Φ_i, denoted with ϕ_i, captures the initial offset of τ_i's arrival time within the schedule, and the rest of the offsets in Φ_i are the times when task τ_i resumes its execution, if preempted. Figures 2 and 3 show examples of such schedule tables.

C. Timing Constraints

Each task may have implicit timing constraints arising out of the task definition and explicit design parameters in terms of arrival offsets and/or deadline requirements. Hence, a task must execute periodically with the given period T_i, and in each period it must finish its worst-case execution C_i within the defined deadline D_i, starting after the earliest release time r_i. In addition, tasks may also have jitter requirements, i.e. constraints on the variance of execution of consecutive period instances [25, p. 81-82], due to control loop considerations [26]. We denote the jitter requirements of a task τ_i with J_i and the observed jitter j_i, i.e., for any two consecutive task instances, the maximal deviation of both starting and finishing times are bounded by J_i.

Other timing requirements are related to message passing between tasks, where the communication latency have to be considered. The most complex set of timing requirements come from the so-called task (or event) chains (c.f. [15]). A task chain specifies, that at least one instance of every task in the given task chain list has to be executed in the specified order within a given maximum end-to-end *reaction latency*. These chains also have a priority, p_i, which can be used for optimization criteria. Since the tasks in the chain can be on different hosts/cores, the communication latencies must be also taken into account. Please note that tasks in the chains may have different activation patterns and periodicity.

We give a simple example of a task chain in Figure 2 composed of 4 tasks, a source (τ_1), two processing tasks (τ_2, τ_3) and a sink (τ_4) with periods of 20 *ms*, 10 *ms*, 10 *ms* and 20 *ms*, respectively. From each instance of the source there needs to be a succession of instances of the other tasks in the right order such that the latency is not exceeded. It is allowed, that an instance of the processing or sink tasks merges multiple signals. In the example, the sink merges the signal from two execution instances of the processing task τ_3.

Let $\mathcal{L}_\aleph$ denote the set of task chains, where a task chain is given by the tuple $\aleph_i = (\{\tau_1 \prec ... \prec \tau_k\}, L_i, p_i)$ with L_i being the allowed end-to-end latency and $p_i \in [0,1]$ is the priority.

For a chain $\aleph_i = (\{\tau_1 \prec ... \prec \tau_k\}, L_i, p_i) \in \mathcal{L}_\aleph$, we formalize the correctness condition for the in-order execution and end-to-end latency requirement as follows:

$$\begin{aligned} &\forall \tau_{1,x}, x \in \left\{0, \ldots, \frac{hp_i}{T_1}\right\} : \exists \{y_2, \ldots, y_k\} \text{ such that} \\ &start(\tau_{2,y_2}) \geq end(\tau_{1,x}) \wedge end(\tau_{k,y_k}) - start(\tau_{1,x}) \leq L_i \wedge \\ &\Big(\forall j \in \{2, k-1\} : start(\tau_{j+1,y_{j+1}}) \geq end(\tau_{j,y_j})\Big), \end{aligned} \quad (1)$$

where hp_i is the hyperperiod of the chain $\aleph_i$ calculated as the least common multiple of the periods of the tasks in the respective chain and interfering tasks[2] and $start(\tau_{i,j})$ and $end(\tau_{i,j})$ denote the start and end of the execution of the job $\tau_{i,j}$, respectively.

III. Problem Formulation

The scheduling algorithm needs to find an assignment of unassigned tasks to cores such that the tasks are schedulable on each assigned core with respect to their timing constraints (offsets, deadlines, and jitter) as well as with respect to the task chain requirements. Moreover, since there is communication either between individual tasks or between tasks in a task chain, the scheduling algorithm also needs to find a schedule for the deterministic communication backbone that respects the required maximum latencies. For this paper we focus on the scheduling problem on the task level and only consider the communication cost as a constant delay for a PCIe communication backbone, as part of a task's WCET. Extending the scheduling strategy for the communication layer for TTEthernet and TSN backbones is subject for future work.

As an input to our problem we have (1) the ADAS platform $\mathcal{A}$ and (2) the applications, denoted by the set of tasks Γ, including the task chains $\mathcal{L}_\aleph$ and all the mapping and timing constraints. We are interested to determine (i) a mapping M of tasks to the cores of the platform and (ii) a static schedule $\mathcal{S}$ of tasks on each core, such that the task deadlines and their jitter, as well as end-to-end constraints on task chains are satisfied.

IV. Mapping and Scheduling Strategy

As can be seen, the scheduling problem resulting out of the platform and task models is very complex and necessitates new scheduling approaches. While other approaches use optimal algorithms (e.g., based on Optimization Modulo Theories or ILP [27]), the size and growing complexity of ADAS platforms make such approaches infeasible in practice. It is expected that ADAS platforms, which already have the complexity of an entire in-vehicle electronics system [18], will grow to a scale of thousands of functions with hundreds of complex event chain requirements. We therefore aim to find a heuristic algorithm that can scale with the growing ADAS trend and can solve complex scheduling problems in a realistic time frame.

We describe a Simulated Annealing (SA)-based metaheuristic approach, first introduced in [6], [7] and extended in [28], [29], which uses an EDF-based heuristic to solve the task scheduling problem. The scheduling heuristic allows task preemption by simulating an Earliest Deadline First (EDF) scheduling policy parameterized by task offsets and local deadlines decided by SA, see Section IV-C. For a mapping of task to cores we assign, according to our SA strategy, an offset and a deadline to each task and simulate EDF to obtain the static schedule, if one can be found. We then check if the schedule adheres to the timing requirements imposed by the jitter and task chain constraints. The algorithm, explained

[2] I.e., tasks that execute on the same core as the tasks in the chain

in detail in Section IV-A, tries to modify the mapping of tasks M, the task offsets ($\phi_i \geq 0$) and deadlines $\mathcal{D}$, in order to find an optimal solution with respect to the end-to-end latency of chains. The novelty in our approach is that our proposed Simulated Annealing method makes use of the different dimensions, that influence task execution, i.e., task mapping, task offsets and task deadlines in order to converge to a near-optimal solution faster than traditional approaches.

A. Simulated Annealing

Simulated Annealing is a heuristic method that aims to optimize solutions by randomly selecting a candidate solution in the neighbourhood of the current one [30]. The SA algorithm is a variant of the neighborhood search technique, where the local search space is explored by moving from the current solution to a neighbor solution [31, p.285]. A new solution is accepted if it is an improvement, however, a worse solution can also be accepted with a certain probability that depends on the deterioration of the cost function *Cost* and on a *cooling scheme* captured by the initial temperature, t_s and a cooling rate cr, which is the rate at which the temperature drops with time [31, p.285].

An essential component of the algorithm is the generation of a new candidate solution s' (also called neighbor) starting from s. The neighbor solutions s' are generated through performing design transformations (also called *moves*) on s.

SA is presented in Alg. 1, and it takes as input: The platform model $\mathcal{A}$, the applications Γ, the initial solution s_0 which acts as the starting point of the search (s_0 is generated by a Greedy mapping algorithm, where tasks are iteratively assigned to those cores that have the most available utilization), the initial temperature t_s, the cooling rate cr which controls the temperature decay, and the number of iterations to maintain a static temperature i.

Algorithm 1 SimulatedAnnealing($\mathcal{A}, \Gamma, s_0, t_s, cr, i$)

```
1:  t ← t_s
2:  s ← ScheduleSynthesis(A, Γ, s_0)
3:  s* ← s
4:  while timeleft do
5:      while t > 1.0 do
6:          for k ← 1 to i do
7:              s' ← GenerateNeighbor(A, Γ, s)
8:              if Cost(s') <Cost(s) then
9:                  s ← s'
10:                 if Cost(s') <Cost(s*) then
11:                     s* ← s'
12:                 end if
13:             else if exp((Cost(s)−Cost(s'))/t) > random[0,1] then
14:                 s ← s'
15:             end if
16:             t ← t · (1 − cr)
17:         end for
18:     end while
19: end while
20: return s*
```

We denote with s' the neighboring candidate solution generated from the currently accepted solution s, line 7. Also, s' is accepted if it improves on s, line 9. The cost function *Cost* used to evaluate a solution is detailed in Section IV-B. We record the best so far solution s^* at line 11. SA returns s^* (at line 20) when it terminates, that is when the allotted time has expired (line 4). As mentioned, the main feature of a SA is that we also accept worse solutions, with a certain probability, see line 13 in Alg. 1.

The GenerateNeighbor procedure functions as a simple state machine, allowing different moves to be chosen randomly. We use three moves, described in the following, SwapTask, AdjustOffset and AdjustDeadline. Various probability assignments for these moves were tried, and, based on observations from performed experiments a uniform distribution has been chosen for all actions.

AdjustDeadline adjusts the deadline of a single randomly selected task. Only tasks that failed at complying to the jitter constraints, are potential candidates for this move. Note that the deadlines in $\mathcal{D}$ are used to control the resulting EDF schedule. We do not change the relative deadline D_i of the task, which is one of its timing constraints. For a task τ_i, AdjustDeadline will modify the deadline used by EDF to schedule τ_i, such that it is lower or equal to D_i. We check for each resulted schedule that all timing constraints are satisfied.

SwapTasks swaps the core mapping of two randomly selected tasks, considering the imposed mapping constraints. For example, if the task has a processor affinity, the swapping is done within the cores of the particular processor. Only tasks that are allowed to swap, are considered, meaning only tasks without a predefined core assignment. Offset and Deadline adjustments are reset to zero for both tasks when performing this action. Finally, the utilization/core load is not considered, and as such this action might overload one of the cores.

AdjustOffset changes the offset of a randomly selected task. This action has two modes. (1) It will select tasks from a specific processor, if deadline/jitter constraints are violated. The target is determined by the processor with the highest number of accumulated violations. Initially, the specific core were also included as part of the selected, however it decreased the performance. (2) If no deadline/jitter violations occur, then the task is chosen randomly from the complete task set.

B. Cost Function

The cost function (*Cost*), defined in Eq. 2, captures both a minimization objective with respect to the end-to-end latency of task chains and penalties representing constraint violations given by the application. We introduce $\rho_{\aleph}$ as a weighted average cost of computation chain violations as well as ρ_D and ρ_J representing deadline and jitter costs (defined below).

The function itself has two cases, (1) a value if the solution configuration meets all the timing constraints and (2) a combination of static and dynamic penalties, if one or more

timing constraints are violated, i.e.,

$$Cost(s) = \begin{cases} \frac{\sum_{\aleph_i \in \mathcal{L}_\aleph} \frac{l_i}{L_i} \cdot p_i}{|\mathcal{L}_\aleph|} \cdot w_1 & \text{if } \chi(s) = \text{true} \\ w_1 + \rho_\aleph + \rho_D + \rho_J & \text{if } \chi(s) = \text{false} \end{cases} \quad (2)$$

where $\chi(s)$ is a test defined by $\chi(s) = \rho_\aleph + \rho_D + \rho_J \not> 0$.

The minimization objective is the average weighted distance of the measured end-to-end latency l_i over the imposed constraint L_i, of all task chains. The static penalty w_1 in Eq. 2 for $\chi(s) = false$ ensures, that any invalid solution will be rated worse relative to that of any valid solution. That is, it adjusts the score such that the minimal penalty value is higher than that of any feasible solution, thereby preventing the annealing process from accepting any invalid candidate over a valid one, as the currently best. Furthermore, the penalty function incurs an increased cost, if any end-to-end, deadline or jitter constraints are violated, as also evident from Eq. 2. Here $\rho_\aleph$, listed in Eq. 3, measures the weighted average of end-to-end violation. The violation of a chain $\aleph_i$ is defined as the difference between its highest observed chain latency l_i and its end-to-end constraint L_i.

$$\rho_\aleph = \frac{\sum_{\aleph_i \in \mathcal{L}_\aleph} \frac{min(L_i, max(0, l_i - L_i))}{L_i}}{|\mathcal{L}_\aleph|} \cdot w_2 \quad (3)$$

Also, from Eq. 3 we see that the observable violation range is clamped in the interval $[0, L_i]$, whereas the penalty itself is bounded by $[0, w_2]$ and grows proportionally with the number of violations.

Likewise, the additional deadline and jitter costs (ρ_D and ρ_J) is listed by Eq. 4 and Eq. 5, respectively. Here ρ_D measures the weighted average of deadline violations with a violation range clamped in the interval $[0, D_i]$. The deadline violation of a task τ_i is denoted as the difference between the maximal relative finishing time of all τ_i's instances f_i and the relative deadline D_i.

$$\rho_D = \frac{\sum_{\tau_i \in \Gamma} \frac{min(D_i, max(0, f_i - D_i))}{D_i}}{|\Gamma|} \cdot w_3. \quad (4)$$

Finally, ρ_J measures the weighted average of jitter violations. We define the jitter violation of a task τ_i as the difference between the maximal observed jitter j_i and the threshold J_i. The violation range is then clamped in the interval $[0, J_i]$.

$$\rho_J = \frac{\sum_{\tau_i \in \Gamma} \frac{min(J_i, max(0, j_i - J_i))}{J_i}}{|\Gamma|} \cdot w_4, \quad (5)$$

In Eq.2-Eq.5, we list w_1, w_2, w_3 and w_4 as static weights designed to capture the importance of the respective violation with the following constraints: $w_2 \geq w_1$, $w_3 \geq w_1$, $w_4 \geq w_1$. The constants were determined based on manual experimentation and observations, with w_1 through w_4 set to 10,000, 40,000, 10,000 and 60,000, respectively. Please note that there are no optimal values for the weights, since they have to be adapted to the application domain, criticality definitions and design goals of the respective use-case.

Algorithm 2 ScheduleSynthesis($\mathcal{A}, \Gamma, s$)

```
1:  Initialize(s)
2:  l ← 2 · hyperperiod + MaxOffset
3:  for each v_i ∈ V ∈ A do
4:      for each σ_k ∈ v_i do
5:          Enqueue(Q_σ, σ_k)
6:      end for
7:  end for
8:  while Empty(Q_σ) is false do
9:      σ ← Dequeue(Q_σ)
10:     cycle ←NextCycle(σ)
11:     if cycle < l then
12:         next ← EDFSimulation(σ, cycle)
13:         SetNext(σ, next)
14:         Enqueue(Q_σ, σ)
15:     end if
16: end while
17: return s
```

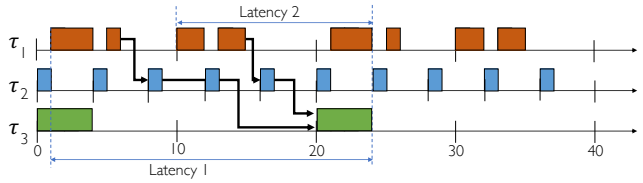
The penalty function has been devised such that small improvements, with respect to violations, will reduce the weighted penalties, ideally leading to gradual improvements. If a course-grained penalty function is used, it would result in a lot of equal values between candidates with respect to their fitness. Using the fine grained approach, the penalty is thus a tighter approximation of the true distance from achieving a valid solution.

C. EDF Simulation for Schedule Synthesis

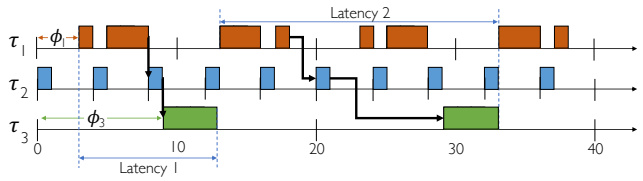
Scheduling of tasks to processors is a well researched topic [32]. In our case, it is intractable to generate optimal schedules, hence we use a heuristic algorithm. In order to include task preemption in a simplified manner, we propose a schedule synthesis heuristic (Alg. 2) based on *simulating* Earliest Deadline First (EDF) scheduling, similar to [11], [28], [29]. EDF is a scheduling algorithm [25] which prioritizes tasks at each time instant depending on their deadlines, i.e., the one with the earliest deadline will get control of the CPU. Given the task WCETs, offsets and deadlines, the schedule table $\mathcal{S}$ is generated by simulating how EDF would execute tasks until the hyperperiod. For a given mapping, offsets and deadlines, EDF will always produce the same schedule. We vary the output schedule produced by the EDF simulation by allowing the Simulated Annealing to change the mapping M, offsets ϕ_i of each task τ_i and deadlines $\mathcal{D}$ via the moves presented earlier.

ScheduleSynthesis receives as input the architecture $\mathcal{A}$, applications Γ and the the solution s to be simulated by EDF (containing the mapping M, offsets ϕ and deadlines $\mathcal{D}$). Due to the nature of task chains and their temporal dependencies, the simulation of each core cannot be executed separately. That is, each schedule produced by the individual EDF simulations must take every other task execution into account.

We start by assigning all tasks to their respective cores (line 1 in Alg.2). All tasks without a task mapping will



(a) End-to-end task chain latencies **not** satisfied



(b) End-to-end task chains latencies satisfied

Fig. 3. Unoptimized (a) vs. optimized (b) schedules.

be mapped according to a best-fit strategy with respect to utilization, i.e., balancing the processor and core utilization. We run the simulation for a length l (set in line 2), after which the schedule will repeat itself. l is defined by $2 \cdot hyperperiod + MaxOffset$, where hyperperiod is determined as the Least Common Multiple of all tasks in Γ and the *MaxOffset* is the maximum over all offsets ϕ_i [33].

The iteration over the simulation length l is done in the while-loop in ScheduleSynthesis (lines 8–16). The current time is captured by *cycle*, and we advance the time to the *next* event, that needs to be simulated.

The EDF simulation is performed per core σ (line 12) and we use a queue Q_σ, containing all cores from all processors, ordered by the earliest event that needs to be simulated. To start the simulation, lines 3–7 add cores to the queue Q_σ, by visiting all the cores in the architecture $\mathcal{A}$.

The next event to be simulated is determined by taking the head of the queue Q_σ (Dequeue) and calling NextCycle. We add cores to be simulated in Q_σ only if we are still within the simulation length l. The while loop stops, when there are no cores to be simulated (Q_σ is empty). The EDF simulation logic is taking place in the EDFSimulation function , called at line 12, which simulates up to the *next* event, which is returned. The product of ScheduleSythesis is then a recording of all occurred events, from which we can derive the schedule table $\mathcal{S}$ of the current solution s.

Our EDFSimulation implementation is able to skip unnecessary cycles. It does so by progressing towards the nearest event defined by either releasing a tasks from the waiting queue, choosing the task with the earliest deadline first from the ready queue, completing a task or allowing preemption to occur on certain break point defined by a parameter called *macrotick* for each core. The macrotick defines the preemption granularity. The macrotick is set such that it allows preemption, under the constraint that the overhead due to context switches, on each processor, should be low, see [11], [34], [35] for a discussion.

We illustrate the EDF approach via the example in Figure 3, consisting of an architecture with a single processor with two cores, σ_1 and σ_2, both of which having a macro tick of 1 *ms*. The depicted application is modelled by the task set $\Gamma = \{\tau_1, \tau_2, \tau_3\}$, which is constrained by zero jitter for all tasks. The tasks are defined as $\tau_1 = (\sigma_0, 0, \phi_1, 4, 10, 10)$, $\tau_2 = (\sigma_0, 0, \phi_2, 1, 4, 4)$ and $\tau_3 = (\sigma_1, 0, \phi_3, 4, 20, 20)$. Furthermore, the set of task chains is defined by $\mathcal{L}_\aleph = \{\aleph_1\}$, with $\aleph_1 = (\{\tau_1 \prec \tau_2 \prec \tau_3\}, 20, 1.0)$. Please note, that given Eq. 1, only two chain instances are necessary to validate as the hyperperiod of $\aleph_1$ is 20 *ms*, and the period of τ_1 is 10 *ms*.

Figure 3a depicts a solution, where the jitter and the task chain end-to-end constraints are violated, whereas Figure 3b shows a valid solution. As seen from Figure 3a, τ_1 violates its jitter constraints, as the start (and end) of execution within its periods varies. This is detected, when TriggerTaskEvent raises the events for the respective tasks instances. For example, the event triggering the start of execution with respect to $\tau_{1,1}$ and $\tau_{1,2}$ differs by 1 *ms*. While the initial offset ϕ_i for all tasks is 0, resulting in $\tau_{2,1}$ and $\tau_{3,1}$ starting their execution first, neither are source tasks with respect to $\aleph_1$. Moving forward, $\tau_{1,1}$ is started at cycle 1, causing the event to trigger the registration of a task chain instance $\aleph_{1,1}$. At cycle 4, $\tau_{1,1}$ is preempted by $\tau_{2,2}$ while $\tau_{3,1}$ completes is execution. Although $\tau_{3,1}$ is a sink task, and a chain instance has been registered, the instance has yet to receive the completion of $\tau_{1,1}$ and $\tau_{2,k}$ before it is accepted. That is, the presence of an event from $\tau_{2,k}$ that happens after $\tau_{1,1}$ must be registered. Subsequently, $\tau_{1,1}$ completes at cycle 5, allowing the $\aleph_{1,1}$ to advance its state, waiting for $\tau_{2,3}$. Lastly, $\tau_{3,2}$ finalizes its execution at cycle 23, thus completing $\aleph_{1,1}$ with a resulting latency of 23, which incidentally violates the given constraints. The chain instance $\aleph_{1,2}$ is registered at cycle 10, and also finalizes at cycle 23, yielding a latency of 14. Given that both latency and jitter constraints have been violated, the product of the ScheduleSynthesis is not feasible.

However, in an optimized solution, solving the associated violations can be achieved by manipulating the initial offsets ϕ_i for the tasks, as depicted in Figure 3b. Here, the schedule has been altered such that all executions of τ_1 and τ_3 have been deferred by ϕ_1 and ϕ_3. For τ_1, the displacement ϕ_1 solves the jitters, because all jobs $\tau_{1,i}$ now start (and end) at the same cycle relative to its period. Finally, τ_3 has been displaced by 9 cycles, such that its initial execution allows $\tau_{3,1}$ to catch the events from $\tau_{2,3}$ (and by extension $\tau_{1,1}$), thus reducing the latency of $\aleph_{1,1}$. Likewise, by introducing ϕ_1 for τ_1 the latency of $\aleph_{1,1}$ was reduced even further. The combined effect of ϕ_1 and ϕ_3 is full compliance of all constraints with the resulting latency's of 10 *ms* and 20 *ms* for $\aleph_{1,1}$ and $\aleph_{1,2}$, respectively.

V. Experimental Results

As a first experiment, we were interested to determine the ability of our proposed SA to find near-optimal solutions. We have implemented an exhaustive search that finds the optimal solution; however, we were able to do that only for small task sets of less than 10 tasks considering an architecture with two cores. Our SA was able to find the same optimal solution in less than 10 s. In the next sets of experiments, determining the efficacy of SA was achieved through a combination of synthetic and realistic test scenarios, benchmarked against two

TABLE I
EVALUATION RESULTS ON SYNTHETIC TEST CASES

Test case	Time	Greedy							SA							GA						
		Chains			Jitter			Sched.	Chains			Jitter			Sched.	Chains			Jitter			Sched.
		Min	*Avg*	*Max*	*Min*	*Avg*	*Max*		*Min*	*Avg*	*Max*	*Min*	*Avg*	*Max*		*Min*	*Avg*	*Max*	*Min*	*Avg*	*Max*	
ADAS1x100%	1 hour	0.97	0.98	1.00	0.58	0.61	0.68	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
ADAS1x200%	2 hours	0.97	0.99	1.00	0.55	0.67	0.75	1.00	0.98	1.00	1.00	0.94	1.00	1.00	1.00	0.98	1.00	1.00	0.71	0.95	1.00	1.00
ADAS1x300%	3 hours	0.97	0.99	1.00	0.52	0.64	0.72	1.00	0.97	0.99	1.00	0.70	0.87	1.00	1.00	0.97	0.99	1.00	0.70	0.88	1.00	1.00
ADAS1x400%	4 hours	0.97	0.97	0.98	0.52	0.64	0.73	1.00	0.97	0.99	1.00	0.69	0.80	0.88	1.00	0.94	0.99	1.00	0.70	0.81	0.92	1.00
ADAS1x500%	5 hours	0.97	0.98	0.98	0.51	0.62	0.70	1.00	0.95	0.98	0.99	0.63	0.78	0.86	1.00	0.95	0.98	1.00	0.64	0.79	0.87	1.00

other heuristics: Greedy and Genetic Algorithm (GA). Greedy decides the mapping of tasks to cores aiming to distribute the workload such that no core overloads. Thus, iterating through tasks, Greedy allocates a task to the core with the least utilization available. For the Greedy-based heuristic, we use the same EDF simulation technique (with preemption), considering zero offsets and the deadlines of the tasks (i.e., these parameters are not optimized).

GA is a multi-objective optimization heuristic inspired from evolutionary theory [36]. We (i) encode each solution (*chromosome*) as an array where each entry (*gene*) contains information on the mapping, offset and deadline of a task and (ii) randomly initialize *N individuals*. We then (iii) evolve some selected candidates by using (iv) recombination and (v) mutation. Finally, (vi) the evolved candidates with better *fitness* will replace the parent population. The fitness of a solution is evaluated using the same cost function as for SA, however the cost for each task deadline, task jitter, and chain is its own objective. The cost for the GA is the vector of each of these objectives. Steps (iii) to (vi) are repeated until the allotted time is exhausted. We employ a standard uniform crossover, and for mutation we do as follows: for each gene in the chromosome, we compare a randomly generated number with a "probability of mutation" and if this number is smaller, then this position is mutated. To select parents we sort the "population" using the "non-dominated" sorting method from [36]. Half the population are kept as parents, and to create new individuals, two random parents are picked until all individuals have been created.

All experiments were conducted on a High Performance Computing (HPC) cluster, with each node configured with 2xIntel Xeon Processor 2660v3 (10 cores, 2.60GHz) and 128 GB memory. Both SA and GA run on one node at a time.

Synthetic Test Cases. The synthetic task sets were generated using a tool developed for this purpose [7], which derives the desired task properties from the realistic task set presented in the next subsection. We are interested to determine if using an SA meta-heuristic combined with EDF-simulation is a viable solution for finding feasible schedules, when confronted with very large task sets.

Thus, we have used five test cases, ranging from 100% to 500% in scale, i.e., for *ADAS1x100%* the application contains 151 tasks and 31 chains using a model of the architecture discussed in section II, whereas with *ADAS1x200%* the architecture would double the number of processors, tasks and task chains. The results are presented in Table I, with each row representing the results of a task case. A *test case* is a scenario consisting of 30 synthetically generated task sets, with each undergoing 30 trials (runs of SA and GA on the same test case). Thus a single test case, e.g. *ADAS1x100%*, would conduct 900 trials for each algorithm. As the experiment progresses through each case, the algorithms were given additional time due to an inherent increased complexity of the problem (see the *Time* column).

TABLE II
EVALUATION RESULTS ON REALISTIC TEST CASES

Test case	Time	Greedy			SA						
		Chains	Jitter	Sched.	Chains			Jitter			Sched.
					Min	Avg	Max	Min	Avg	Max	
ADAS1	3.20	0.81	0.37	1.00	0.97	0.99	1.00	0.95	0.99	1.00	1.00
ADAS2	6.40	0.65	0.21	1.00	0.94	0.99	1.00	0.84	0.99	1.0	1.00
ADAS3	13.20	0.48	0.21	1.00	0.84	0.99	1.00	0.74	0.97	1.0	1.00

For each algorithm (Greedy, SA and GA), we show in the table, under the *Sched.* columns, the percentage of cases (out of the 30 trials) for which the algorithms determine schedulable solutions (all *deadline* constraints are satisfied; 1 means 100%). The columns labelled *Chains* have the percentage of chains out of the total chains, for which the respective algorithm was able to satisfy the end-to-end constraints. Similarly, *Jitter* denotes the percentage of jitter constraints satisfied. These values are presented for in terms of minimum, average and maximum considering the 30 runs. Note that the Greedy algorithm is not stochastic and always outputs the same result.

As we can see from Table I, the Greedy approach has comparatively the worst performance in terms of complying with the constraints. We also see that SA is able to find schedulable solutions (in terms of deadlines, chains and jitter constraints) within the allotted time, even when the problem size increases. We see that SA has a drop in finding feasible schedules (from 100% in column *Chains.* for ADAS1x100%, to 63% for ADAS1x500%, and cannot meet the jitter constraints for some of the two largest test cases). We estimate that this is caused by a combination of increased difficulty of the task sets and their constraints, as well the crude method for estimating the time allotted. We observed that both SA and GA obtain similar quality results, with SA being slightly better for smaller test cases and GA doing slightly better for larger test cases. Reflecting on the results, we expect that SA would be able to find an increased percentage of feasible solutions given more time. However, both metaheuristics (SA and GA) are clearly superior to the mapping heuristic such as Greedy, when presented with very large task sets.

Realistic Test Cases. For the following evaluation, we were interested in the ability of SA to handle realistic test cases. Thus, we have used three test cases, ADAS1 to ADAS3, which are variants of an anonymized realistic task set, currently in use in a series-production vehicle. All test cases have 151 tasks and 31 task chains, but with varying jitter, earliest activation and macrotick constraints. The experiment was setup such that 30 trials were conducted by SA for each test case; the time limit used is in minutes. As we can see from Table II, SA can find feasible solutions for all test cases. As the test cases get progressively more difficult from ADAS1 to ADAS3, in terms of timing constraints that need to be satisfied, SA retains its ability of finding solutions within the allotted time, albeit at a slightly lower rate. By comparison we see that the percentage of resolved constraints for the Greedy algorithm decreases similarly, and fails on all accounts to find feasible schedules that meet all the constraints.

VI. Conclusions and Future Work

In this paper we have considered safety-critical ADAS applications mapped on modern multi-processor platforms. The applications are modeled as a set of communicating software tasks with complex timing requirements, e.g. jitter, deadlines and end-to-end latency bounds on task chains. We have proposed an optimization strategy that, given the application and platform models, determines a mapping of tasks to the cores of the platform and a static schedule of tasks on each core, such that the timing constraints are satisfied. Our optimization strategy uses a Simulated Annealing metaheuristic to explore the solution space, combined with a scheduling heuristic based on an EDF simulation to solve the preemptive task scheduling problem. The experimental evaluation on several realistic and synthetic test cases has demonstrated that our proposed strategy is able to find solutions, that meet the timing constraints at a higher rate than traditional approaches, and scales with the growing trend of ADAS platforms.

As future work we want to extend both SA and GA to include the communication scheduling (considering TSN) as well as potential virtualization layers. We envision adding extra optimization objectives, such as reducing the number of task preemptions in order to reduce context switch overhead.

References

[1] G. Niedrist, "Deterministic architecture and middleware for domain control units and simplified integration process applied to ADAS," https://www.tttech.com/technologies/adas/, 2016.

[2] O. Gietelink, J. Ploeg, B. D. Schutter, and M. Verhaegen, "Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations," *Vehicle System Dynamics*, vol. 44, no. 7, 2006.

[3] M. Hammond, G. Qu, and O. A. Rawashdeh, "Deploying and scheduling vision based advanced driver assistance systems (ADAS) on heterogeneous multicore embedded platform," in *Proc. FCST*, 2015.

[4] S. Sommer, A. Camek, K. Becker, C. Buckl, A. Zirkler, L. Fiege, M. Armbruster, G. Spiegelberg, and A. Knoll, "Race: A centralized platform computer based architecture for automotive applications," in *Proc. IEVC*, 2013.

[5] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nelis, and T. Nolte, "Contention-free execution of automotive applications on a clustered many-core platform," in *Proc ECRTS*, 2016.

[6] S. D. McLean, "Mapping and scheduling of real-time tasks on multi-core autonomous driving platforms," Master's thesis, Technical University of Denmark, 2019.

[7] S. D. McLean, P. Pop, and S. S. Craciunas, "Mapping and scheduling of real-time tasks on multi-core autonomous driving platforms," Technical University of Denmark, Tech. Rep., 2019.

[8] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the let paradigm," in *RTAS*, 2018.

[9] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Syst.*, vol. 2, 1990.

[10] T. F. Abdelzaher and K. G. Shin, "Combined task and message scheduling in distributed real-time systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 11, pp. 1179–1191, 1999.

[11] S. S. Craciunas, R. Serna Oliver, and V. Ecker, "Optimal static scheduling of real-time tasks on distributed time-triggered networked systems," in *Proc. ETFA*. IEEE Computer Society, 2014.

[12] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *J. Syst. Archit.*, vol. 80, no. C, Oct. 2017.

[13] J. Schlatow and R. Ernst, "Response-time analysis for task chains in communicating threads," in *Proc. RTAS*, 2016.

[14] A. C. Rajeev, S. Mohalik, M. G. Dixit, D. B. Chokshi, and S. Ramesh, "Schedulability and end-to-end latency in distributed ecu networks: Formal modeling and precise estimation," in *Proc. EMSOFT*, 2010.

[15] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *Proc. RTCSA*, 2016.

[16] IEEE, "TSN Task Group," https://1.ieee802.org/tsn/.

[17] W. Steiner, G. Bauer, B. Hall, and M. Paulitsch, "TTEthernet: Time-Triggered Ethernet," in *Time-Triggered Communication*. CRC, 2011.

[18] TTTech Computertechnik AG, "Automated Driving Offering," https://www.tttech-auto.com/products/automated-driving/, 2018.

[19] S. Bunzel, "AUTOSAR – the standardized software architecture," *Informatik-Spektrum*, vol. 34, no. 1, pp. 79–83, Feb 2011. [Online]. Available: https://doi.org/10.1007/s00287-010-0506-7

[20] F. Sagstetter, S. Andalam, P. Waszecki, M. Lukasiewycz, H. Stähle, S. Chakraborty, and A. Knoll, "Schedule integration framework for time-triggered automotive architectures," in *Proc. DAC*. ACM, 2014.

[21] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty, "Modular scheduling of distributed heterogeneous time-triggered automotive systems," in *Proc ASPDAC*, 2012.

[22] A. Mehmed, W. Steiner, and M. Rosenblattl, "A time-triggered middleware for safety-critical automotive applications," in *Ada-Europe*, 2017.

[23] R. Ernst, S. Kuntz, S. Quinton, and M. Simons, "The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092)," *Dagstuhl Reports*, vol. 8, no. 2, pp. 122–149, 2018.

[24] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. of the ACM*, vol. 20, 1973.

[25] G. C. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag, 2011.

[26] M. Di Natale and J. A. Stankovic, "Scheduling distributed real-time tasks with minimum jitter," *IEEE Transactions on Computers*, vol. 49, no. 4, pp. 303–316, 2000.

[27] S. S. Craciunas and R. Serna Oliver, "Combined task- and network-level scheduling for distributed time-triggered systems," *Real-Time Systems*, vol. 52, no. 2, pp. 161–200, 2016.

[28] M. Barzegaran, A. Cervin, and P. Pop, "Towards quality-of-control-aware scheduling of industrial applications on fog computing platforms," in *Proc. IoT-Fog*. ACM, 2019.

[29] M. Barzegaran, A. Cervin, and P. Pop, "Performance optimization of control applications on fog computing platforms using scheduling and isolation," *IEEE Access*, vol. 8, 2020.

[30] E. K. Burke, G. Kendall *et al.*, *Search methodologies*. Springer, 2005.

[31] P. Pop, P. Eles, and Z. Peng, *Analysis and Synthesis of Distributed Real-Time Embedded Systems*, 22nd ed. Kluwer Academic Publishers, 2004.

[32] O. Sinnen, *Task scheduling for parallel systems*. Wiley&Sons, 2007.

[33] J. Y.-T. Leung and M. Merrill, "A note on preemptive scheduling of periodic, real-time tasks," *Inf. Process. Lett.*, vol. 11, no. 3, 1980.

[34] A. Zuepke, M. Bommert, and D. Lohmann, "AUTOBEST: a united AUTOSAR-OS and ARINC 653 kernel," in *Proc. RTAS*, 2015.

[35] M. Aichouch, J. Prévotet, and F. Nouvel, "Evaluation of the overheads and latencies of a virtualized RTOS," in *Proc. SIES*, 2013.

[36] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II," in *Proc. PPSN*, 2000.