frontiers in Robotics and AI

ORIGINAL RESEARCH
published: 04 January 2022
doi: 10.3389/frobt.2021.762227

Configuring ADAS Platforms for Automotive Applications Using Metaheuristics

Shane D. McLean[1], Emil Alexander Juul Hansen[1], Paul Pop[1] and Silviu S. Craciunas[2]*

[1]Technical University of Denmark Kongens Lyngby, Kongens Lyngby, Denmark, [2]TTTech Computertechnik AG, Vienna, Austria

Modern Advanced Driver-Assistance Systems (ADAS) combine critical real-time and non-critical best-effort tasks and messages onto an integrated multi-core multi-SoC hardware platform. The real-time safety-critical software tasks have complex interdependencies in the form of end-to-end latency chains featuring, e.g., sensing, processing/sensor fusion, and actuating. The underlying real-time operating systems running on top of the multi-core platform use static cyclic scheduling for the software tasks, while the communication backbone is either realized through PCIe or Time-Sensitive Networking (TSN). In this paper, we address the problem of configuring ADAS platforms for automotive applications, which means deciding the mapping of tasks to processing cores and the scheduling of tasks and messages. Time-critical messages are transmitted in a scheduled manner via the timed-gate mechanism described in IEEE 802.1Qbv according to the pre-computed Gate Control List (GCL) schedule. We study the computation of the assignment of tasks to the available platform CPUs/cores, the static schedule tables for the real-time tasks, as well as the GCLs, such that task and message deadlines, as well as end-to-end task chain latencies, are satisfied. This is an intractable combinatorial optimization problem. As the ADAS platforms and applications become increasingly complex, such problems cannot be optimally solved and require problem-specific heuristics or metaheuristics to determine good quality feasible solutions in a reasonable time. We propose two metaheuristic solutions, a Genetic Algorithm (GA) and one based on Simulated Annealing (SA), both creating static schedule tables for tasks by simulating Earliest Deadline First (EDF) dispatching with different task deadlines and offsets. Furthermore, we use a List Scheduling-based heuristic to create the GCLs in platforms featuring a TSN backbone. We evaluate the proposed solution with real-world and synthetic test cases scaled to fit the future requirements of ADAS systems. The results show that our heuristic strategy can find correct solutions that meet the complex timing and dependency constraints at a higher rate than the related work approaches, i.e., the jitter constraints are satisfied in over 6 times more cases, and the task chain constraints are satisfied in 41% more cases on average. Our method scales well with the growing trend of ADAS platforms.

Keywords: automotive applications, task scheduling, task preemption, time-sensitive networking, TSN, IEEE 802.1Qbv

OPEN ACCESS

Edited by:
Dakshina Dasari,
Robert Bosch, Germany

Reviewed by:
Risat Pathan,
University of Gothenburg, Sweden
Navid Razmjooy,
Independent researcher, Belgium

****Correspondence:***
Paul Pop
paupo@dtu.dk

Specialty section:
This article was submitted to Robotic Control Systems, a section of the journal Frontiers in Robotics and AI

Received: *21 August 2021*
Accepted: *14 October 2021*
Published: *04 January 2022*

Citation:
McLean SD, Juul Hansen EA, Pop P and Craciunas SS (2022) Configuring ADAS Platforms for Automotive Applications Using Metaheuristics. Front. Robot. AI 8:762227. doi: 10.3389/frobt.2021.762227

1 INTRODUCTION

Advanced Driver Assistance Systems (ADAS), present in more and more modern consumer vehicles, perform complex functions that range from driver assistance, e.g., automated or assisted parking, lane changing, etc., to fully autonomous driving. In modern ADAS systems, there is a drive towards moving functions from hardware to software and the architecture from distributed to centralized, allowing modularization within an integrated hardware platform that can be cooperatively used and centrally managed (Niedrist, 2018). This drive has multiple advantages, like reusability and portability, but presents several challenges, especially in terms of real-time, testing, and safety (Gietelink et al., 2006). The fusion of multiple software functions of different criticality levels onto the same hardware platform has to be done in a composable manner with guaranteed temporal and spatial isolation without sacrificing real-time capabilities. 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. (Hammond et al., 2015; Niedrist, 2018)).

Generally, integrated ADAS platforms are composed of heterogeneous multi-core CPUs and Systems-on-chip (SoCs) of different performance and safety levels that are interlinked by a (real-time) communication network (Sommer et al., 2013; Becker et al., 2016b). In such integrated platforms, the ADAS functions have complex timing requirements and 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 via the time-aware communication backbone to control algorithms that activate, e.g., the emergency brake system. This succession of function execution and message transmission creates a temporal dependency chain, which has to comply with a set of timing requirements in terms of latency. In order to guarantee both the interdependence and real-time behavior of tasks and their messages, the safety-critical ADAS functions and their communication frames have to be scheduled appropriately. 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.

1.1 Related Work

The scheduling of task sets with dependencies has been a well-studied topic within the real-time community. Task schedules with inter-task dependencies are computed in (Chetto et al., 1990) by modifying the offsets and deadlines of the individual tasks and then using EDF to schedule the new task set (Buttazzo, 2011). In (Choi and Agrawala, 2000) the notion of absolute and relative timing constraints (i.e., events are temporally dependent on each other) for source and sink task requirements are introduced. Furthermore, the authors present a scheduling approach for uniprocessor systems with complex timing constraints such as jitter requirements. In (Fohler, 1994) the authors compute static schedules for tasks that communicate through bounded delay protocols like TDMA or TTP using dependency graphs. The work in (Tindell and Clark, 1994) presents an analysis of the schedulability of tasks that communicate using the TDMA protocol. In (Abdelzaher and Shin, 1999) an optimal task schedule for communicating tasks is generated using a branch-and-bound approach. A similar approach is introduced in (Peng et al., 1997) with modified optimization criteria. (Craciunas et al., 2014) presents a similar heuristic scheduling approach to ours which uses EDF simulation to create static schedules for tasks with communication and precedence dependencies but in contrast to our work, the results only apply for dependencies between tasks with equal periods. The temporal dependencies between tasks presented in the prior work described above are not as complex as the ones arising from the ADAS task chains where not only task periods can be different, but the correctness of the chain dependency is related to individual task jobs. Multi-rate tasks and complex precedence constraints have been analyzed in (Forget et al., 2011, 2017; Mubeen and Nolte, 2015). Additionally, in (Isović and Fohler, 2000), a two-step approach for distributed systems is introduced, which is based on an offline computation and an online EDF mechanism for scheduling tasks with complex constraints like jitter and job-level precedence requirements.

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 multi-rate chains, c.f. (Becker et al., 2017), depending on the available system information, e.g., scheduling algorithm or task offsets. 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 schedule construction guarantees the real-time requirements. In (Rajeev et al., 2010), 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 (Becker et al., 2016a), the authors introduce a task chain latency analysis that does not require information about the concrete scheduling algorithm. (Verucchi et al., 2020) use an existing list-scheduling algorithm but apply it on a directed acyclic graph (DAG), which is constructed from multi-rate task sets such that complex precedence and timing constraints are captured and satisfied.

In (Lukasiewycz et al., 2012) a modular framework for ILP-based scheduling of time-triggered distributed automotive systems is presented, where both bus access and operating system schedules are created. The end-to-end latency of chains only applies single-rate dependency chains, and the method suffers from an exponential increase in runtime with increasing the number of tasks and messages. An extension that adds an incremental step in order to reduce the runtime complexity of the schedule generation has been proposed in (Sagstetter et al., 2014), where the focus is on integrating locally optimized schedules into a globally non-optimal solution. In terms of the communication backbone, the scheduling problem for TSN networks has been addressed in, e.g., (Craciunas et al., 2016; Serna Oliver et al., 2018) for fully deterministic communication needs, including latency and jitter requirements without taking into account the schedule of the communicating tasks. Furthermore, the combined task and message scheduling problem has also been thoroughly studied for other types of networks, e.g., TTEthernet (Craciunas and Serna Oliver, 2016), shared registers (Becker et al., 2016a), or the Universal Communication Model (Pop et al., 2003).

1.2 Contributions

In our earlier work (McLean et al., 2020) we considered that the communication backbone is done via Peripheral Component Interconnect Express (PCIe), and we have used a periodic real-time task model in which the worst-case execution time (WCET) of a task changes based on the core speed and the communication is modeled as overhead at the end of task instance execution. This paper extends our work to consider the IEEE 802.1 Time-Sensitive Networking (TSN) deterministic Ethernet standard for the communication. TSN is becoming a de-facto standard in several areas, e.g., industrial, automotive, avionics, space, with a broad industry adoption and several vendors developing TSN switches. This paper presents a heuristic-based scheduling algorithm for ADAS platforms that considers the different dimensions of timing and dependency requirements and is designed with scalability in mind. The optimization approaches are based on metaheuristics (Simulated Annealing and Genetic Algorithm), which take into account not only the timing constraints but also design goals, such as function allocation on computing units. We consider both PCIe and TSN for the communication. Future work may also include the LET model (Biondi and Di Natale, 2018) which is becoming increasingly popular in the automotive domain since it can provide deterministic communication behavior.

To the best of our knowledge, this is the first work to propose a heuristic-based solution to the combined scheduling problem in ADAS platforms that requires a solution for the task-to-core assignment, static task schedule generation, and the scheduling of TSN messages sent by the tasks, which respects both task and complex task chain timing constraints.

We start by introducing the platform and application models in **Section 2** followed by a description of the scheduling problem in **Section 3**. We introduce the algorithm in **Section 4** followed by an experimental evaluation in **Section 5** and conclude the paper in **Section 6**.

2 PLATFORM AND APPLICATION MODELS

2.1 System Model

The modern integrated ADAS hardware platform features a multi-core multi-SoC embedded ECU with a variety of CPUs and Graphics Processing Units (GPUs) running at different speeds, which are interconnected through either a deterministic Ethernet backbone, such as TSN (IEEE, 2016b) or TTEthernet (Steiner et al., 2011), or through PCIe. RazorMotion (TTTech Computertechnik AG, 2018), for example, features a Renesas RH850P/1H-C ASIL D MCU with lockstep cores running at 240 MHz and two Renesas R-Car H3

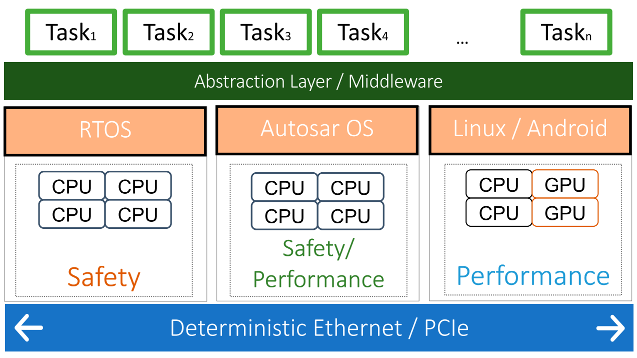

FIGURE 1 | High-level platform model.

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, which is similar to the platform described in (Marija Sokcevic, 2020). 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 (Bunzel, 2011)) 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 (Lukasiewycz et al., 2012; Sagstetter et al., 2014; Mehmed et al., 2017; Ernst et al., 2018), 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 (TTTech Computertechnik AG, 2018) 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 (Niedrist, 2018). 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 (Mehmed et al., 2017). Hence, in this paper, we focus on creating static schedules for the table-driven dispatching mechanism of such ADAS systems.

Tasks performing software functions of different criticality levels communicate with each other both on- and off-chip through different means. On-chip communication is usually done through buffers, message passing, or shared memory, while off-chip communication is achieved either through PCIe or TSN. The safety-critical communication also has to adhere to stringent timing requirements and has to be aligned to the execution schedule of the real-time tasks. For example, when PCIe is used, a message sending cost has to be taken into account when scheduling the respective communicating tasks. When using time-aware switched Ethernet technologies like TSN, the schedule of the messages has to be aligned to the execution of the tasks, and the end-to-end latency requirements comprising both task execution and message transmission have to be met.

We model an ADAS platform as a graph $\mathcal{A}(\mathcal{V}, \mathcal{E})$. A vertex $v_i \in \mathcal{V}$ is either an end system (ES) that performs computations, the set of all ESes denoted with $\mathcal{ES}$, or a TSN switch (SW), their set being denoted with $\mathcal{SW}$. The edges $\mathcal{E}$ are the communication links. Each ES $v_i \in \mathcal{ES}$ has a set of computation cores $\mathcal{C}_i$.

2.2 TSN

Time-Sensitive Networking (IEEE, 2016b) addresses the need to have more determinism and real-time capabilities over standardized Ethernet networks. To achieve this, TSN defines a series of amendments to the IEEE 802.1Q standard, as well as stand-alone mechanisms and protocols (e.g., 802.1ASrev). TSN has already seen adoption in the industrial domain and is becoming increasingly relevant in the automotive domain. The main mechanisms out of the TSN ecosystem that we consider in this paper are the clock synchronization protocol IEEE 802.1ASrev (IEEE, 2016a), which provides a synchronized clock reference, and the timed-gate functionality of IEEE 802.1Qbv (IEEE, 2015) bringing scheduled communication capabilities on the egress ports of devices. The timed-gate mechanism is essentially a shaping gate that forwards selected message streams from each egress queue according to the transmission schedule encoded in so-called Gate-Control Lists (GCL). A TSN stream is defined by a payload size, a talker (sender), one or more listeners (receivers), and optional timing requirements in terms of jitter and latency. The global schedule synthesis has been studied in (Craciunas et al., 2016; Dürr and Nayak, 2016; Pop et al., 2016; Serna Oliver et al., 2018) focusing on enforcing deterministic transmission, temporal isolation, and compositional system design for critical streams with end-to-end latency requirements.

A communication link is modeled as a directed edge is represented by two vertices $[v_a, v_b] \in \mathcal{E}$. All physical links in the system are bidirectional and so for each $[v_a, v_b] \in \mathcal{E}$, there exists a $[v_b, v_a] \in \mathcal{E}$ with the same properties except that source and destination are swapped. In TSN-based systems, a directional virtual edge is created for each core, i.e., the edge $[v_a, v_a]$ is added to $\mathcal{E}$. Each link $[v_a, v_b] \in \mathcal{E}$ has a set of attributes. We denote with $[v_a, v_b].s$ the transmission bandwidth of the link, $[v_a, v_b].c$ denotes the number of queues in associated egress port.

2.3 Application Model

On top of this platform, many 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, $\Gamma = \{\tau_i | 1 \leq i \leq n\}$, similar to the model in (Liu and Layland, 1973). 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 worst-case execution time (WCET), 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) $\tau_{i,k}$, $k = 1, 2, \ldots$ (Buttazzo, 2011, 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 $M\colon \Gamma \rightarrow \mathcal{C}_i$.

The scheduling allows preemption, i.e., a schedule table can be constructed such that a task is interrupted by another task and then resumes its execution. 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 correctly with respect to the deterministic timing behavior, will allow even better resource utilization.

Tasks may exchange messages. A message is modeled as a flow (stream) $\Phi_i \in \mathcal{L}_\Phi$, which has a period $\Phi_i.T$ which is the same as its sender and receiver tasks, a deadline $\Phi_i.D$, which can be arbitrary but smaller than or equal to the period $\Phi_i.T$, a message size $\Phi_i.P$, and a fixed route $\Phi_i.r$. Messages are transmitted as frames. If a message exceeds the maximum transmission unit (MTU) of 1,500 bytes defined for standard Ethernet, then the message is split into $k = \lceil \frac{\Phi_i.P}{1500} \rceil$ fragments. The message is split such that each resultant frame is its maximum size until the last, which has the last bytes. A frame $\theta_{i,m}^{[v_a,v_b]}$ is an instance of the frame on the link $[v_a, v_b]$. It is associated with the ith flow and has sequence number m. A frame has a transmission duration of $\theta^{[v_a,v_b]}.L$ microseconds on the link $[v_a, v_b]$ where it is transmitted. A frame can have a maximum payload of MTU 1,500 bytes plus the 42 bytes Ethernet header.

2.4 Timing Constraints

Each task may have implicit timing constraints arising from the task definition and explicit design parameters related to 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 (Buttazzo, 2011, p. 81-82), due to control loop considerations (Di Natale and Stankovic, 2000). We denote the jitter requirements of a task τ_i with J_i and the observed jitter j_i, i.e., the maximal deviation of both starting and finishing times for any two consecutive task instances are bounded by J_i.

Other timing requirements are related to message passing between tasks, where the communication latency has to be considered. The most complex set of timing requirements come from the so-called task (or event) chains (c.f. (Becker et al., 2016a)). 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 needs have to be included in the end-to-end latency considerations. In a PCIe backbone, the latencies between communicating tasks are modeled and enforced as an additional delay after executing

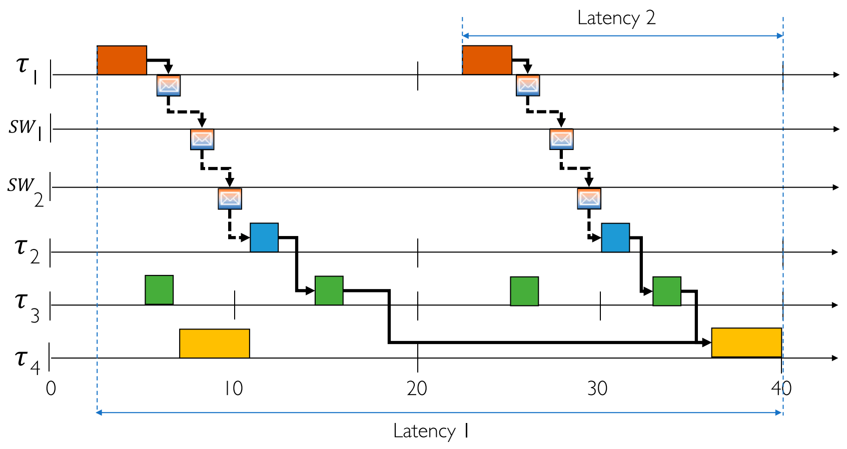

FIGURE 2 | Task chain example with TSN communication.

the sending tasks. In the case of TSN, the frame schedule must be aligned to the task execution to ensure that the messages are sent after the sending task has been executed and before the receiving task starts. Please note that tasks in the chains may have different activation patterns and periodicity, i.e., we are considering multi-rate chains (c.f. (Becker et al., 2017)).

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*, 20 *ms*, 10 *ms* and 20 *ms*, respectively. The critical communication between τ_1 and τ_2 is done off-chip through the TSN network over two switches (SW_1 and SW_2) since τ_1 and τ_2 are running on different SoCs. For the purposes of this illustration, tasks τ_2, τ_3, and τ_4 are located on the same core, and the communication is assumed to be in 0-time. 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. The communication frames from τ_1 to τ_2 have to be scheduled in such a way that the message is forwarded through the switches and arrives at τ_2 before the respective instance of τ_2 executes. The latency of the communication needs to be also included in the total end-to-end latency.

Let $\mathcal{L}_{\aleph}$ denote the set of task chains, where a task chain is given by the tuple $\aleph_i = (\{\tau_1 \prec \ldots \prec \tau_k\}, L_i, p_i)$ with L_i being the allowed end-to-end latency and $p_i \in [0, 1]$ is the priority. For a chain $\aleph_i = (\{\tau_1 \prec \ldots \prec \tau_k\}, L_i, p_i) \in \mathcal{L}_{\aleph}$, we formalize the correctness condition for the in-order execution and end-to-end latency requirement as follows:

$$\forall \tau_{1,x}, x \in \left\{0, \ldots, \frac{hp_i}{T_1}\right\}: \exists \{y_2, \ldots, y_k\}, y_j \in \left\{0, \ldots, \frac{hp_i}{T_j}\right\}, \forall j \in \{2, k\}$$

such that

$$start(\tau_{2,y_2}) \geq end(\tau_{1,x}) \wedge end(\tau_{k,y_k}) - start(\tau_{1,x}) \leq L_i \wedge \left(\forall j \in \{2, k-1\}: start(\tau_{j+1,y_{j+1}}) \geq end(\tau_{j,y_j})\right), \quad (1)$$

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.[1] 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.

If there is communication over the TSN network between any two tasks in the chain, the TSN network schedule needs to reflect several correctness conditions. Firstly, the correctness conditions from (Craciunas et al., 2016) for generating GCL schedules need to be fulfilled in order to have correct and deterministic frame transmission over IEEE 802.1Qbv TSN devices. In addition to the technological constraints of standard full-duplex Ethernet networks, a deterministic timing of frames is enforced in (Craciunas et al., 2016) through so-called frame/flow isolation constraints. In the timed-gate mechanism of IEEE 802.1Qbv, the transmission schedule applies to the entire traffic class (as opposed to individual frames like in, e.g., TTEthernet). Therefore, the queue state has to be known and deterministic in order to ensure that the right frames are sent at the right time. Hence, the isolation conditions in (Craciunas et al., 2016) enforce that a correct GCL schedule isolates frames of different flows either in the space domain by placing them in different egress queues or in the time domain, preventing frames of different flows from being in the same queue at the same time. Secondly, implementation or network-specific correctness conditions need to be fulfilled. Here we mention the synchronization error and the microtick of the timeline. The synchronization protocol defined in IEEE 802.1As-rev ensures a common clock reference; however, individual clocks may still have a bounded time differential towards the clock reference. The maximum of all the individual bounded clock errors is called the network precision (δ). Furthermore, the (hardware) realization of the required state machines defined in, e.g., IEEE 802.1Qbv implementing the TAS mechanism also has a certain overhead, resulting in a minimum mandatory spacing of scheduled events (called microtick or link granularity). The

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

microtick or link granularity defines the fastest rate at which schedule events can be processed by the TSN hardware and hence, the granularity of the TSN scheduling timeline. Thirdly, the GCL schedule and, more specifically, the schedule offsets of the frame transmission need to be aligned to the task schedule. For example, a message transmitted between two tasks within a chain has to be scheduled for sending after the execution of the sending task and has to arrive before the receiving task executes. Additionally, the communication latency, which adds to the chain's overall end-to-end latency, has to be within the given latency requirements.

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 O_i on its core σ_i. The first offset in O_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 O_i are the times when task τ_i resumes its execution if preempted. The schedule table $\mathcal{S}$ also contains the GCLs, which are captured via an offset $\theta^{[v_a,v_b]_{i,m}}.\phi$, where i and m are the flow and the frame instance, respectively, and $[v_a, v_b]$ is the link on which the frame is transmitted. **Figure 2** shows a Gantt chart, which is a visual representation of a schedule table.

3 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 concerning their timing constraints (offsets, deadlines, and jitter) as well as concerning 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.

As an input to our problem we have 1) the ADAS platform $\mathcal{A}$ and 2) the applications, denoted by the set of applications Γ, including the task chains $\mathcal{L}_{\aleph}$ and all the mapping and timing constraints. We are interested in determining 1) a mapping M of tasks to the cores of the platform and 2) 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. We consider a constant delay for the PCIe communication backbone as part of a task's WCET. For the TSN backbone, we assume that the flows and their routing are given (e.g., using the shortest path), and the schedule $\mathcal{S}$ also contains the offsets of frames, i.e., we also 3) determine the TSN GCLs.

4 MAPPING AND SCHEDULING STRATEGY

The presented in the previous section is a combination of the problems in (Pop et al., 2016) and (McLean et al., 2020). Both problems are complex scheduling problems, and the decision problem associated with them have been thoroughly investigated in the literature. (Sinnen, 2007) and (Garey and Johnson, 1979) prove it to be in the NP-complete class by reducing it to the known 3-PARTITION and PARTITION problems. With the assumption that $P \neq NP$, the scheduling problem cannot be solved efficiently by a polynomial-time algorithm. In our initial investigation of the problem, we have implemented a solution using Optimization Modulo Theories (OMT), which is an extension of Satisfiability Modulo Theories (SMT), based on the work in (Craciunas and Serna Oliver, 2016). However, the OMT/SMT approach has not been able to find solutions due to the complexity of the problem. The increasing complexity of ADAS platforms renders such mathematical programming approaches, including Integer Linear Programming (ILP) (Craciunas and Serna Oliver, 2016), infeasible in practice. It is expected that ADAS platforms, which already have the complexity of an entire in-vehicle electronics system (TTTech Computertechnik AG, 2018), will grow to a scale of thousands of functions with hundreds of complex event chain requirements. For such intractable problems, researchers have proposed the use of problem-specific heuristics and metaheuristics (Burke and Kendall, 2005), as an alternative to exact optimization methods which have exponential running times. Several metaheuristic approaches have been presented in the literature (Burke and Kendall, 2005), and the challenge is to identify the right metaheuristic for our problem. Metaheuristics aim to find a good quality solution in a reasonable time but do not guarantee that an (optimal) solution will always be found. Based on the review of the related work, we have decided to implement a combination of heuristics for scheduling, based on List Scheduling (Sinnen, 2007) and a *simulation* of the Earliest Deadline First (EDF) (Craciunas and Serna Oliver, 2016) scheduling algorithm. For the mapping, we have decided to compare the Simulated Annealing (SA) and Genetic Algorithm (GA) metaheuristics, which have been shown in the literature to be a promising approach for task mapping problems.

4.1 Solution Overview and Cost Function

An overview of our optimization strategy is illustrated in **Figure 3**. The metaheuristics (SA or GA) decide the mapping M via an iterative search which generates neighboring solutions from the current solution, see "SA or GA search procedure" box in the figure, see the details in **Section 4.2**. The schedule $\mathcal{S}$ of flows and messages is decided by a combination of scheduling heuristics, i.e., List Scheduling (LS) and Earliest Deadline First (EDF), see the box with LS and EDF in the figure, and the details in **Section 4.3** and **Section 4.4**, respectively. Finally, the quality of a solution is evaluated using a *cost function*, see "Cost Function" box in the figure and the discussion later in this section. We present our proposed solution for TSN-based systems. The same solution is used for PCIe-based systems, with the observation that the flows are not considered, i.e., they are modeled as an overhead added to the sending task's WCET.

We use an LS-based heuristic to jointly schedule the flows and the tasks involved in communication, presented in **Section 4.3**. Once the communicating tasks and flows are scheduled, we use an Earliest Deadline First (EDF)-based scheduling heuristic to add the tasks not involved in communication across cores and optimize the schedule, also by introducing design-time "preemption", i.e., task splitting, see **Section 4.4**. We then check if the schedule adheres to the timing requirements imposed by the jitter and task chain constraints. The EDF scheduling heuristic introduces design-time task preemption

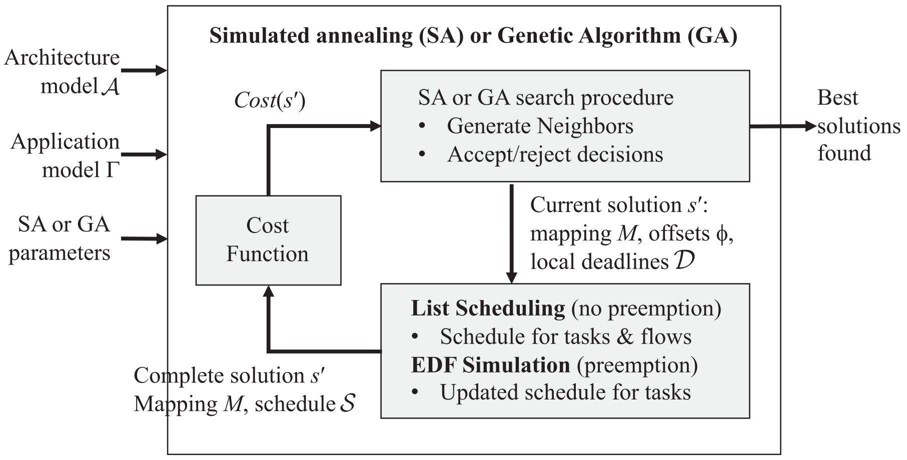
Simulated annealing (SA) or Genetic Algorithm (GA)
Architecture model $\mathcal{A}$
Application model Γ
SA or GA parameters
$Cost(s')$
SA or GA search procedure
• Generate Neighbors
• Accept/reject decisions
Best solutions found
Cost Function
Current solution s': mapping M, offsets ϕ, local deadlines $\mathcal{D}$
List Scheduling (no preemption)
• Schedule for tasks & flows
EDF Simulation (preemption)
• Updated schedule for tasks
Complete solution s' Mapping M, schedule $\mathcal{S}$

FIGURE 3 | Solution overview.

by *simulating* at design time an EDF scheduling policy parameterized by task offsets and local deadlines. The heuristics receive as an input the mapping of tasks to cores. The LS heuristic is controlled by the tasks and flow offsets ($\phi_i \geq 0$), which are the earliest times a task can be started, or a flow can be sent. The EDF heuristic is controlled by both the offsets and local deadlines $\mathcal{D}$, see the arrow in **Figure 3** from the "SA or GA search procedure" box and the LS and EDF box below.

The mapping and the controlling parameters (offsets, deadlines) for the scheduling heuristics are determined by the metaheuristics, as part of their search procedure. We have developed two metaheuristics, one based on Simulated Annealing (SA), see **Section 4.2.1** and one based on a Genetic Algorithm (GA), see **Section 4.2.2**. Both metaheuristics modify the mapping of tasks M, the task and flow offsets ($\phi_i \geq 0$), and deadlines $\mathcal{D}$, to find an optimal solution with respect to the cost function. The novelty in our approach is that our metaheuristics make use of the different dimensions that influence task execution, i.e., task mapping, task offsets, and task deadlines, to converge to a near-optimal solution faster than traditional approaches.

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. The function 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.

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

During the search, the metaheuristics do not reject the invalid solutions. Instead, we "penalize" an invalid solution by increasing its cost function value to be larger than the values for valid solutions in the hope of driving the search towards valid solutions. A solution is *invalid* if one of the three constraints is violated: 1) There is a task or flow which does not meet its deadline, i.e., the worst-case response time f_i of a task τ_i is larger than its deadline D_i (or the worst-case delay of a flow is larger than its deadline). 2) There is a chain $\aleph_i$ which has an end-to-end latency l_i that is greater than its allowed latency L_i. 3) There is a task τ_i which has a jitter j_i greater than maximum allowed jitter J_i, see the notations in **Section 2.4**. We capture each of these constraint violations using a separate penalty term, which is zero if the constraint is not violated: 1) ρ_D for deadline violations, 2) $\rho_\aleph$ for chain latency violations and 3) ρ_J for jitter violations. Hence, if the sum of these penalty terms is zero then the solution is valid, i.e., all constraints are satisfied and thus all penalty terms are zero. We capture this situation with a test defined by $\chi(s) = \rho_\aleph + \rho_D + \rho_J \ngtr 0$. $\chi(s)$ is true if the solution is valid, i.e., case (1) in **Eq. 2**, and false if the solution is invalid, corresponding to case (2).

Let us first discuss case (1) when the solution is valid. In this case, the value of the cost function which has to be minimized is defined as the average weighted distance of the measured end-to-end latency l_i over the imposed constraint L_i, of all task chains. Basically, the smaller the chain latencies, the smaller the term. When we sum up $\frac{l_i}{L_i}$ we also multiply with the chain's priority p_i to capture the relative importance of chains. The resulted summation is divided with the number of chains $|\mathcal{L}_\aleph|$. Note that when a solution is valid, $l_i \leq L_i$. The divisions of l_i by L_i and of the summation by $|\mathcal{L}_\aleph|$ are intended to normalize the cost function term. In addition, we multiply the thus resulted term with a static penalty weight w_1. We will discuss the use of weights shortly when we cover case (2).

Let us consider the example in **Figure 4** where we have three tasks and one chain on a single core. The details are given next to the figure, with the note that the allowed end-to-end latency of the chain L_1 is 20, and its priority p_i is 1.0. **Figure 4A** shows a valid solution, whereas **Figure 4B** an invalid one. For **Figure 4A** we have $l_1 = 20$, hence the cost function is $\frac{20}{20} \cdot 1.0 \cdot w1 = 1 \cdot w1$. Let us now discuss the cost function for the case 2) when a solution is invalid. In that case, $\rho_\aleph + \rho_D + \rho_J$ will be greater than zero. To this term, we add w_1 to ensure that any invalid solution will be rated

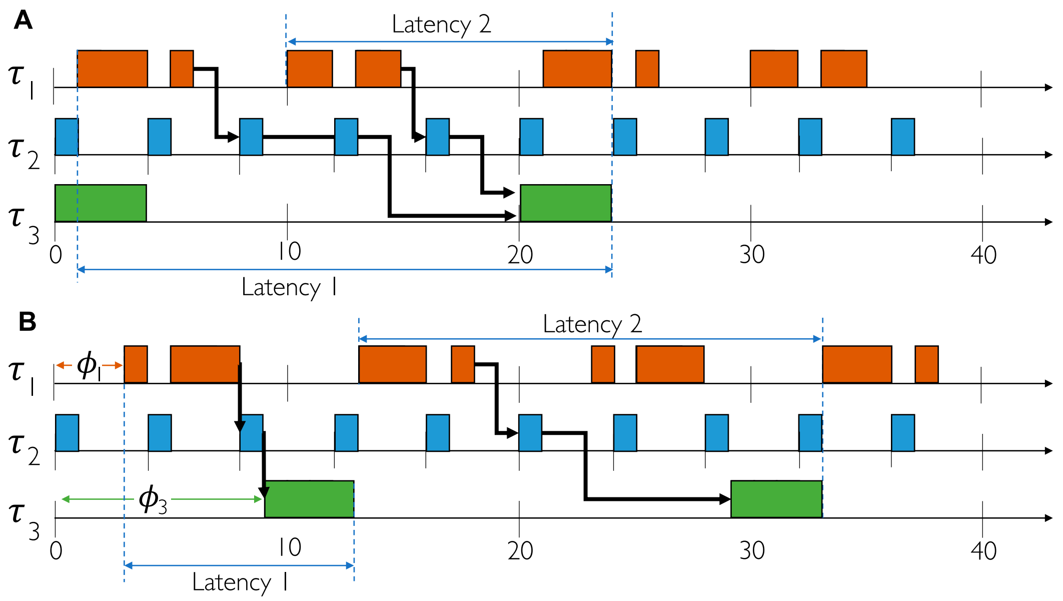

FIGURE 4 | Schedule optimization. **(A)** End-to-end task chain latencies not satisfied. **(B)** End-to-end task chains latencies satisfied.

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 search process from accepting any invalid candidate over a valid one. As discussed, we use ρ_D to penalize deadline violations, $\rho_\aleph$ to penalize chain violations and ρ_J to penalize jitter violations.

The values of these penalty functions are dynamic, i.e., a larger value is used bigger constraint violations. We first define the $\rho_\aleph$ from **Eq. 3** in detail and then discuss the other two penalties, which are similarly constructed. The relative importance of these penalties are determined by the weights w_2, w_3 and w_3. $\rho_\aleph$ 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.

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

If l_i is smaller or equal to L_i then the chain constraint is satisfied and the term is zero. We discuss here the case when the constraint is not satisfied, i.e., $l_i > L_i$. When $l_i > L_i$ the *max* operator will return $l_i - L_i$. To normalize the penalty value, we clamp any observed violation $l_i - L_i$ to the interval $[0, L_i]$ using the *min* operator and divide by L_i, hence the term in the summation will be in the interval [0, 1]. We divide the sum with the number of chains $|\mathcal{L}_\aleph|$ and then multiply with the static weight w_2. Let us illustrate this with the example in **Figure 4B**, where the latency $L_i = 20$ for chain $\aleph_1$ is violated, i.e., $l_i = 23$. Hence, $\rho_\aleph = \frac{\frac{min(20, max(0, 23-20))}{20}}{1} \cdot w_2 = \frac{3}{20} \cdot w_2$.

Likewise, the additional deadline and jitter costs (ρ_D and ρ_J) is listed by **Eqs. 4**, **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 or flow i is denoted as the difference between the maximal relative finishing time of all of i's instances f_i and the relative deadline D_i.

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

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 $[0, J_i]$.

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

In **Eqs. 2–5**, 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.

4.2 Metaheuristics

The SA and GA metaheuristics aim to iteratively optimize solutions by randomly changing existing solutions s to create new solutions s' and evaluate them by using the cost function. They take as input the platform model $\mathcal{A}$ and the applications Γ and return the best solution s^* found according to the cost function. Both start from an initial solution s_0. Metaheuristics can start from any initial solution, even a random one. However, in our case, we have developed a *Greedy* algorithm to generate the initial solution. With SA and GA the mapping is optimized during the search. With Greedy, the mapping is decided constructively as follows. For each task, considering the processor affinity constraints (that restrict the mapping to specific processors), we iterate through all cores and identify the core with the lowest utilization. The utilization of a task is its WCET divided by its period. The utilization of core is the sum of all task utilizations on that core. We then map the task to the respective core. This ensures a balanced utilization of cores in the initial solution. Once the mapping is decided, we use the same LS and EDF scheduling heuristics to schedule the tasks and flows, see **Section 4.3** and **Section 4.4**, respectively. However, with Greedy, the input parameters to the scheduling heuristics, such as offsets for both LS and EDF and local deadlines for EDF, are not optimized. Thus, we consider that the offsets are all set to zero and the local deadlines are set to the absolute deadlines.

4.2.1 Simulated Annealing

We first describe a Simulated Annealing (SA)-based metaheuristic approach, which uses an EDF-based heuristic to solve the task scheduling problem. Simulated Annealing is a heuristic method that aims to optimize solutions by randomly selecting a candidate solution in the neighborhood of the current one (Burke and Kendall, 2005). The SA algorithm accepts a new neighbor solution if it is better than the current one. Moreover, a worse solution can be accepted with a certain probability given by the cost function *Cost* and the *cooling scheme* defined by an initial temperature, t_s and a cooling rate cr, specifying the rate at which the temperature drops with each iteration.

A new candidate solution s' (also called neighbor) is generated starting from the current solution s by performing design transformations (also called *moves*) on s. We use three moves, described in the following. *AdjustDeadline* adjusts the deadline of a single randomly selected task. Only tasks that failed at complying with 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 or flow.

The function that generates neighbor solutions is implemented as a simple state machine, allowing moves mentioned earlier to be chosen randomly. Various probability assignments for these moves were tried, and, based on observations from performed experiments, a uniform distribution has been chosen for all actions.

4.2.2 Genetic Algorithm

GA is a multi-objective optimization heuristic inspired by evolutionary biology (Deb et al., 2000). We 1) encode each solution (*chromosome*) as an array where each entry (*gene*) contains information on the mapping, offset and deadline of a task/flow and 2) randomly initialize *N individuals*. We then 3) evolve some selected candidates by using 4) recombination and 5) mutation. Finally, 6) the evolved candidates with better *fitness* will replace the parent population. As mentioned, GA is a multi-objective metaheuristic. This means that the *fitness* is captured with several cost function values, i.e., $\rho_{\aleph}$ for chains, ρ_D for deadlines and ρ_J for jitter constraints, see the discussion in **Section 4.1**. This is in contrast to SA, which collapses all these terms into a single cost function value, as defined in **Eq. (2)**. Steps 3) to 6) are repeated until the allotted time is exhausted.

Several crossover operators have been proposed in the literature, and we have implemented: uniform order-based crossover (OX), (OX) using 2-point and using 1-point approaches (Syswerda, 1991), Partially-Mapped Crossover (Goldberg and Lingle, 1985), Cycle-Based Crossover (Goldberg and Lingle, 1987) and Alternating-Position Crossover (Larranaga et al., 1997). Based on our experiments, we have decided to employ a standard uniform crossover. Regarding mutation, 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. The probability of mutation has been determined using ParamILS (Hutter et al., 2009) as discussed in **Section 5**. To select parents, we sort the "population" using the "non-dominated" sorting method from (Deb et al., 2000). Half the population is kept as parents, and to create new individuals, two random parents are picked until all individuals have been created.

4.3 Joint Flow and Task Scheduling

One approach to the task and flow scheduling problem is to solve the problems separately and then fit them together. This is a reasonable approach when the two sub-problems do not form a circular dependency. In our case, however, the two scheduling problems are closely linked together. Let us consider the example in **Figure 5**, where we have a topology of three end systems (ES_1 to ES_3) and one switch (SW_1). We have four tasks τ_A, τ_B, τ_C, and τ_D with precedence constraints [$\tau_A \prec \tau_B \prec \tau_C \prec \tau_D$] that form a chain. τ_A needs to be executed on ES_1, τ_B on ES_2 and τ_C and τ_D on ES_3. Task τ_A sends a message to τ_B, and τ_B sends a message to τ_C. For illustration purposes, the task WCETs and the transmission times of message frames on links are a single time unit, and network precision and macrotick are ignored. The period of all tasks is 8 time units, and the chain latency is also 8 time units.

If the messages are scheduled first, then the solution of the flow would look as shown in **Figure 5A**. This schedule minimizes the flow latencies, but since task τ_B must receive message 1 and send message 2, the schedule contains a lot of idle time; hence, the chain latency becomes 13 time units. Note that messages 1 and 2 scheduled at the beginning of the schedule are sent by tasks τ_A

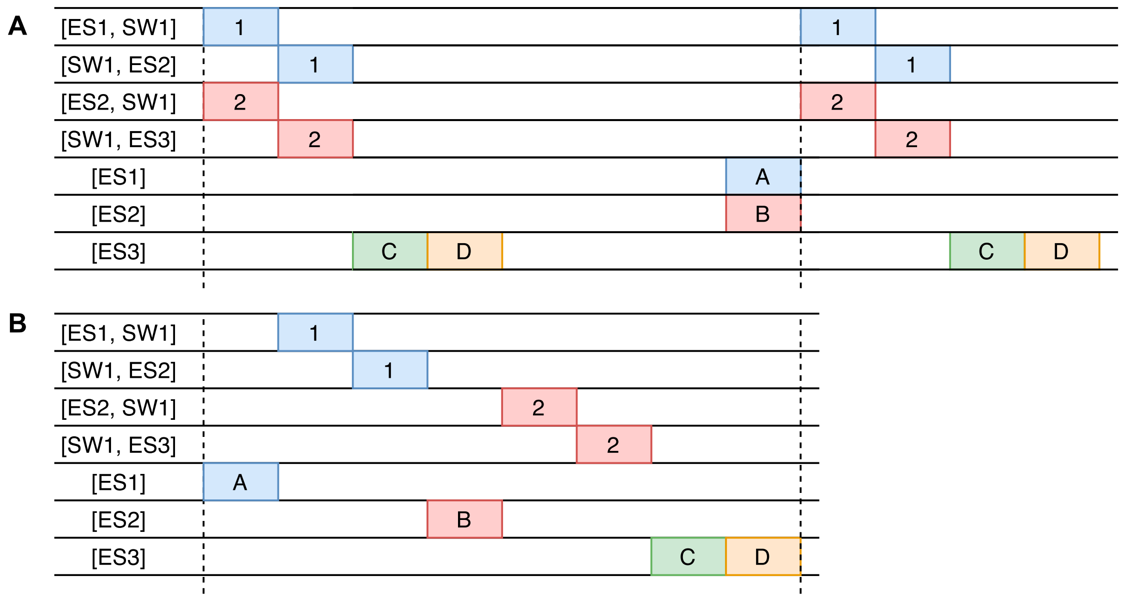

FIGURE 5 | Scheduling approaches, **(A)** is a schedule where flows 1 and 2 are scheduled before taking tasks into account, **(B)** is the optimized scenario where task and flows are scheduled concurrently.

and τ_B from the previous period. The same issue exists if the tasks are scheduled without any knowledge of the network. However, if both task scheduling and message scheduling are optimized concurrently, then an optimized solution, shown in **Figure 5B**, can be produced. This reduces the chain latency substantially to 8 time units from 13 in **Figure 5A**, meeting thus the task chain latency constraint.

In this section, we propose a joint flow and task scheduling heuristic based on List Scheduling (LS) (Sinnen, 2007). LS is a widely used task scheduling heuristic that is known to obtain good quality solutions when determining static schedules for tasks on multiprocessors. We have re-purposed LS for jointly scheduling flows and tasks. Our LS is inspired by the individual flow scheduling heuristic from (Raagaard and Pop, 2017), which uses variants of ASAP (As Soon As Possible) and ALAP (As Late As Possible) scheduling. Both of these are a special case of the List Scheduling heuristic (Sinnen, 2007). Our LS is more general, scheduling both flows and tasks. This LS can use offset and ordering parameters to control the placement of frames, which is not considered in (Raagaard and Pop, 2017).

LS receives as input the architecture $\mathcal{A}$, applications Γ and the solution s generated by the outer metaheuristic, containing the mapping M, offsets ϕ and deadlines $\mathcal{D}$. LS returns a partial schedule table $\mathcal{S}'$ covering the hyperperiod of Γ and including all flows and those communicating tasks involved in sending and receiving the flows. The joint scheduling is achieved by adding those tasks involved in communication as "virtual flows" and the cores where they are mapped for execution as "virtual links". Then, precedence constraints are added to ensure that a frame cannot be sent before its sending task has finished executing, and a receiving task cannot start before its input frames have arrived. Hence, in the following, flows and links also denote tasks and cores, respectively. Note that not all tasks are involved in the communication over TSN. Tasks that are not sending/receiving flows are added to the schedule in a subsequent step using the EDF-based heuristic from **Section 4.4**.

Algorithm 1. ScheduleFlow (θ, ϕ). Schedules a flow θ as soon as possible (ASAP), considering its offset ϕ given by the metaheuristic. All frames in the flow have initialized lower and upper bounds to $-\infty$, ∞, respectively.

```
Schedules a flow $\theta$ As Soon As Possible (ASAP), considering its offset $\phi$ given by the metaheuristic. All
frames in the flow have initialized lower and upper bounds to $-\infty, \infty$, respectively.
1. For every frame of flow $\theta$, do the following:
2.     Get lower bound by Eq. 6
3.     Schedule the frame as ASAP between lower/upper bounds: set $\theta.\phi$ to the earliest transmission time
4.     If the frame can be scheduled, continue, otherwise backtrack.
5.     Set the upper bound of the next frame to the latest available time for its queue
6. If all frames are scheduled, terminate, otherwise go to 2
7. Return $\theta$ ▷ The offsets of all the frames of flow $\theta$ (part of the schedule $\mathcal{S}'$) have now been set
```

Similar to (Raagaard and Pop, 2017), LS starts with an empty timeline and iteratively schedules one flow at a time. The metaheuristic specifies the order in which flows are chosen. Similar to (Raagaard and Pop, 2017), the flows are chosen according to their deadlines since flows with tight deadlines are the hardest to schedule and therefore should be picked first. The tie-breaker for the ordering is given by the flow period. Each flow is scheduled using the ScheduleFlow procedure in **Algorithm 1** such that the end-to-end latency is minimized. The termination condition for the LS is that either a schedule has been found for all flows or the current iteration does not produce a feasible schedule with respect to the flow deadline.

We now examine the steps of **Algorithm 1** in more detail. In step 1 frames are retrieved in the order given by **Eq. 7**, see **Section**

4.3.3 for an explanation of how the next frame is determined. In step 3, the frame offset ϕ is set to the earliest time in a *feasible region*, greater than the *lower bound*, determined in step 2. **Section 4.3.2** and **Section 4.3.1** present how we define and determine the feasible regions and the lower bound, respectively. If the algorithm reaches a state where a frame cannot be scheduled, e.g., there is not enough space, then it needs to find another solution. This search is done by backtracking: In step 4 backtracking is done by increasing the lower bound to the latest available time which is less than the frame offset found in step 3, then rescheduling the previous frame, see **Section 4.3.4**.

4.3.1 Lower Bound

The lower bound for the LS algorithm, inspired by (Raagaard and Pop, 2017), is calculated using **Eq. 6**. $\theta_{i,m}^{[v_x,v_a]}.(\phi + L)$ is used as shorthand for $\theta_{i,m}^{[v_x,v_a]}.\phi + \theta_{i,m}^{[v_x,v_a]}.L$, and where ϕ is the frame offset and L is the frame transmission duration. This equation ensures that the assigned offset fulfills the link congestion and flow transmission constraints, i.e., that frames must be fully received before being transmitted. The *link congestion* constraint does not allow two or more frames to be sent on the same link at the same time. The *flow transmission* constraint restricts the sending of a frame to be after the reception and buffering of that frame in the switch. The link congestion and flow transmission constraints result in a minimum possible end-to-end latency for flows. This lower bound is influenced by the route of the flow as well as the flow characteristics. The following equation, introduced in (Raagaard and Pop, 2017), captures the lower bound on $\theta_{i,m}^{[v_a,v_b]}$, when considering the previous frame on the same link, $\theta_{i,m-1}^{[v_a,v_b]}$, and the same frame on the previous link, $\theta_{i,m}^{[v_x,v_a]}$.

$$\underline{\phi}\left(\theta_{i,m}^{[v_a,v_b]}\right) = \begin{cases} \theta_{i,m}^{[v_a,v_b]}.\phi & \text{if} m = 1 \wedge [v_a, v_b] = \Phi_{i,s} \\ \theta_{i,m-1}^{[v_a,v_b]}.(\phi + L) & \text{if} m > 1 \wedge [v_a, v_b] = \Phi_{i,s} \\ \theta_{i,m}^{[v_x,v_a]}.(\phi + L) + \delta & \text{if} m = 1 \wedge [v_a, v_b] \neq \Phi_{i,s} \\ \max\Big(\theta_{i,m-1}^{[v_a,v_b]}.(\phi + L), & \\ \qquad \theta_{i,m}^{[v_x,v_a]}.(\phi + L) + \delta\Big) & \text{Otherwise.} \end{cases} \tag{6}$$

4.3.2 Feasible Regions

When a flow is scheduled, each frame will *block* those queues and the network links where it is scheduled. The *feasible region* for a frame, similar to (Raagaard and Pop, 2017), is a set of intervals where the frame can be scheduled without violating the feasibility of the existing partial schedule. The algorithm relies on feasible regions to find out where the frames can be scheduled without interfering with other frames. Since frames can have different periods, this complicates the search for space where the frame can be scheduled.

We introduce two operations that the feasible region implement, i.e., queue *blocking*, and *searching* for the feasible region of a frame. *Blocking* is used when a frame is scheduled in a known feasible region, and *searching* is used when the algorithm is searching for an appropriate place for a frame. *Blocking* happens at most once for every flow scheduled, while *searching* can happen several times, depending on how hard it is to schedule a frame. In order to minimize the time in search, the following method of constructing the feasible regions is used.

If a frame $\theta_{i,m}^{[v_a,v_b]}$ is scheduled on the network, then it will block the egress queue $\Phi_i^{[v_a,v_b]}.\rho$ from $\theta_{i,m}^{[v_a,v_b]}.\phi$ to $\theta_{i,m}^{[v_a,v_b]}.\phi + \theta_{i,m}^{[v_a,v_b]}.L$. When we need to determine if a frame of another period interferes with the other frame, we have to check that it does not interfere in the whole hyperperiod. Instead of determining this each time we have to check for a new frame, we create a feasible region for each different period in the network. The feasible regions of other periods are then blocked using the BlockQueues procedure from **Algorithm 2**.

Algorithm 2. BlockQueues ($\theta_{i,m}^{[v_a,v_b]}$). Procedure for blocking time slots inside the feasible regions for a frame instance $\theta_{i,m}$ on a link $[v_a, v_b]$

```
Procedure for blocking time slots inside the feasible regions for a frame instance $\theta_{i,m}$ on a link $[v_a, v_b]$.
1. For each period $T$ in the set of periods used in the TSN network
2.     For $i \in \left\{0, 1, ..., \frac{hyperperiod}{\theta_{i,m}.T}\right\}$
3.         Calculate the times $start = \theta_{i,m}.\phi + (i \cdot \theta_{i,m}.T)$ and $end = \theta_{i,m}.\phi + (i \cdot \theta_{i,m}.T) + \theta_{i,m}.L$
4.         Block link $[v_a, v_b]$ and $\theta_{i,m}$'s queue $T$ from $start \mod T$ to $e \mod T$
```

Step 4 of Alg. 2 does the queue blocking. It takes a *start*, *end*, and a frame, and blocks the frame's queue and link in that interval, where mod is the modulo operator. If *start* > *end*, then the queue is blocked in the intervals $[0, end[$ and $[start, Q_T]$, where Q_T is the period of the queue. An example of the blocking is illustrated in **Figure 6**, where we show on a link $[v_a, v_b]$ how the feasible region of a frame instance $\theta_{i,m}$ is blocked. Let us assume that an earlier frame in the same queue and link had a period of 10 *ms* and our frame's period is $\theta_{i,m}.T = 15$ *ms*. Both frames have an offset of 2 *ms* and their transmission times are 1 *ms*. Frame instance $\theta_{i,m}$ cannot use the time slots where the earlier frame has been scheduled, at every 10 *ms*, the first row in **Figure 6**. In addition, we also need to block those times where, if $\theta_{i,m}$ is scheduled periodically with a period of 15, runs the risk of conflicting with the other frame with a period of 10. For example, $\theta_{i,m}$ cannot be scheduled at time 7, because its next occurrence at 7 + 15 = 22 would conflict with the other frame that periodically is scheduled every 10 *ms* with an offset of 2, i.e., 2 + 10 + 10 = 22. The second row in **Figure 6** shows the times blocked by **Algorithm 2** for our example with two frames.

4.3.3 Getting the Next Frame

The LS heuristic schedules the frames in the order specified by **Eq. 7**:

$$\text{NextFrame}\left(\theta_{i,m}^{[v_a,v_b]}\right) = \begin{cases} \theta_{i,1}^{\text{LCL}} & \text{if bLNL} \wedge \text{bLS} \\ \theta_{i,m+1}^{\text{FNL}} & \text{if bLNL} \\ \theta_{i,m}^{\text{NL}} & \text{otherwise,} \end{cases} \tag{7}$$

where *LCL* is the last "virtual link" (core), *FNL* is the first network link, *NL* is the next link, *bLNL* is true when frame is on the last network link, and *bLS* is true if the frame is the last frame in the flow. The NextFrame function is valid for all frames except the last "virtual frame", where *NextLink* is not defined.

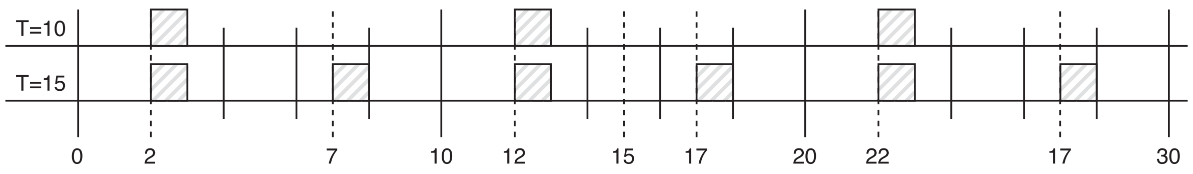

FIGURE 6 | Blocking times of a frame with a period of 15 *ms* considering another frame with a period of 10 *ms* over their hyperperiod.

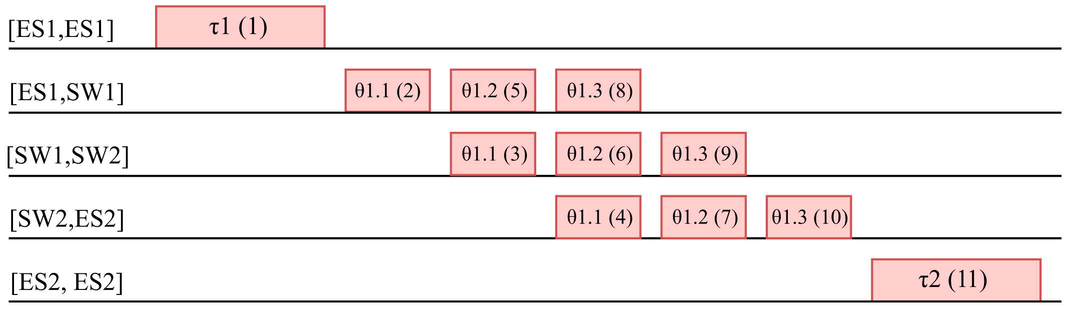

FIGURE 7 | How tasks and flow frames are scheduled together. The order is indicated by the number in parenthesis.

LS starts from the first "virtual frame" (sender task), and goes through each frame and ends with the last "virtual frame" (receiving task). The idea is to allow backtracking only to change the last scheduled frame. We illustrate an ordering in **Figure 7**, where the order is indicated in parenthesis inside the rectangles representing tasks and frames. In **Figure 7** we have a setup where a task τ_1 on ES_1 modeled as a "virtual frame" on the "virtual link" $[ES_1, ES_1]$ sends a flow Φ_1 of size 3xMTU, which hence has to be split in three frames $\theta_1.1$, $\theta_1.2$ and $\theta_1.3$, to a task τ_2 on ES_2.

Figure 7 shows the order in which **Eq. 7** will visit the frames. Note that by using this order and converting tasks to "virtual frames" on "virtual links" we can treat the tasks and frames together and schedule them jointly. Thus, the frames τ_1 and τ_2 on "virtual links" $[ES_1, ES_1]$ and $[ES_2, ES_2]$, respectively, are "virtual frames" (tasks), hence they are scheduled as tasks without concern for MTU-size limits. However, the flow Φ_1 has to be split into frames $\theta_1.1$ to $\theta_1.3$, which are then scheduled as frames on the physical links. The idle times in the schedule in **Figure 7** between each frame are due to the link granularity and synchronization, which have been considered for this example.

4.3.4 Backtracking

When the LS heuristic schedules a frame instance $\theta_{i,m}$, it sets the upper bound of the frame instance $\theta_i^{NL}.\bar{\phi}$ on the next link NL to the latest time available in the queue of the next link NL. When we schedule the frame instance $\theta_{i,m}^{NL}$ there can be two situations, visualized in **Figures 8A,B**. In **Figure 8** we show the previous *link* where the frame instance $\theta_{i,m}$ was transmitted, and the next link NL where the frame instance $\theta_{i,m}^{NL}$ has to be transmitted next. On the link NL we show with hatched rectangles the times where $\theta_{i,m}^{NL}$ cannot be transmitted, e.g., because other frames are being transmitted.

Case (1) is when we have enough space to send $\theta_{i,m}^{NL}$ on NL, depicted in **Figure 8A**. The frame can then be placed into the queue as early as the constraints allow. Conversely, case (2), is when there is not enough space for the frame on NL, as shown in **Figure 8B**. In this case, we use *backtracking*. We first need to check later times on NL when there is space to transmit $\theta_{i,m}^{NL}$. This is achieved by increasing the lower bound of $\theta_{i,m}$ on the previous link such that $\theta_{i,m}^{NL}$ will have to be delayed, i.e., scheduled at a later time. If we are unsuccessful in pushing $\theta_{i,m}^{NL}$ later, we need to push also $\theta_{i,m}$ later, which is achieved by going back on the links where $\theta_{i,m}$ was transmitted, and delaying the frames. This is the backtracking process, which continues going back on the previous links and delaying the frames until we are able to find space for all of them. If such a space cannot be found, it means that the frame cannot be scheduled, hence this solution is infeasible. The metaheuristics will hopefully then guide the search in their outer loop to other solutions where the frame can be scheduled.

4.4 EDF Simulation for Schedule Synthesis

The List Scheduling heuristic from **Section 4.3** has scheduled all the flows and the corresponding communicating tasks, resulting in a partial schedule $\mathcal{S}'$. We propose the EDF-based heuristic in **Algorithm 3** for scheduling the rest of the non-communicating tasks and optimizing the schedule. *EDFScheduleSynthesis* receives as input the architecture $\mathcal{A}$, applications Γ the partial solution s containing the mapping M, offsets ϕ, deadlines $\mathcal{D}$ and partial schedule $\mathcal{S}'$. The schedule synthesis heuristic from **Algorithm 3** is

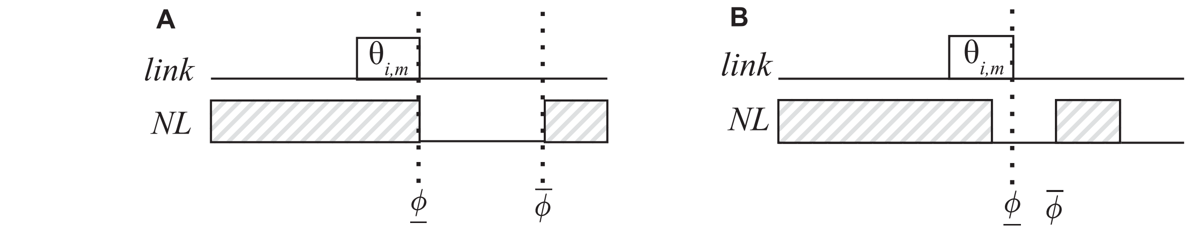

FIGURE 8 | Visualization of lower bound and upper bounds. The hatched areas are already filled by other frames, such that the non-hatched areas form the feasible region. **(A)** Case where the next frame can be scheduled in the feasible region. **(B)** Case where there is not enough space is available to schedule the frame and backtracking will be used to move the frame forward.

based on *simulating* Earliest Deadline First (EDF) scheduling, similar to (Craciunas et al., 2014; Barzegaran et al., 2020). EDF is a scheduling algorithm (Buttazzo, 2011) that 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 partial schedule $\mathcal{S}'$, the complete 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 optimize the schedule produced by the EDF simulation by allowing the metaheuristics in an outer loop to change the mapping M, offsets ϕ_i of each task τ_i and deadlines $\mathcal{D}$, and by allowing preemption for the tasks. The EDF simulation is implemented using the *discrete-event simulation* (DES) paradigm, where the operation of a system is seen as a discrete sequence of events in time, and an *event* captures the change of state in the system at a particular moment in time.

Algorithm 3. EDFScheduleSynthesis ($\mathcal{A}$, Γ, *s*). Schedules the tasks that are not involved in the communication on top of the partial schedule $\mathcal{S}'$ produced by **Algorithm 1**, creating the final schedule $\mathcal{S}$

```
Schedules the tasks that are not involved in the communication on top of the partial schedule $\mathcal{S}'$ produced
by Alg. 1, creating the final schedule $\mathcal{S}$.
 1. Assign unscheduled tasks to cores such that the utilization is balanced
 2. Set the simulation length $l \leftarrow 2 \cdot hyperperiod + MaxOffset$
 3. For each $v_i \in \mathcal{V} \in \mathcal{A}$ ▷ Add cores to the queue $Q_\sigma$, by visiting all the cores in the architecture $\mathcal{A}$
 4.     For each $\sigma_k \in v_i$
 5.         Enqueue($Q_\sigma, \sigma_k$)
 6. For each event in the $Q_\sigma$, as long as the queue is not empty▷ Simulate how EDF works
 7.     Get the next event $\sigma \leftarrow$ Dequeue($Q_\sigma$)
 8.     Get the next $cycle$ ←NextCycle($\sigma$)
 9.     If the cycle is within the simulation length, i.e., $cycle < l$
10.         Determine the next event $next \leftarrow$ EDFSimulation($\sigma, cycle$)
11.         Set the $next$ event on the core $\sigma$
12.         Add the event to the event queue, Enqueue($Q_\sigma, \sigma$)
13. return $\mathcal{S}$
```

We start by assigning all tasks to their respective cores (step 1 in **Algorithm 3**). All tasks without a task mapping will 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 step 2), after which the schedule will repeat itself. *l* is defined by 2 ·*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 (Leung and Merrill, 1980). The iteration over the simulation length *l* is done in the steps 6–12. 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 (step 10), 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, steps 3–5 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. NextCycle is called for a core σ and returns the time *cycle* when the next event of interest for scheduling occurs on that core. Our NextCycle implementation skips unnecessary cycles, i.e. when no events of interest to the scheduling occur. It does so by progressing towards the nearest event defined by either releasing a task 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 a 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 (Aichouch et al., 2013; Craciunas et al., 2014; Zuepke et al., 2015) for a discussion.

We add cores to be simulated in Q_σ only if we are still within the simulation length *l*. The algorithm 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 10, which simulates up to the *next* event, which is returned. The product of EDFScheduleSythesis is then a recording of all occurred events, from which we can derive the schedule table $\mathcal{S}$ of the current solution *s*. As mentioned, LS schedules jointly the flows and those tasks involved in the communication. EDF schedules the tasks not scheduled so far, i.e., those which are not involved in inter-core communication. The EDF algorithm allows task preemption, which in our case means that tasks in the schedule can be split at design-time. This optimizes the schedule table, as the scheduling search space becomes larger and latencies and jitters can be further optimized compared to the case such task splitting is not allowed. In addition, EDF will be allowed to split the communicating tasks already added to the partial schedule $\mathcal{S}'$ by the LS heuristic if such task splitting will not result in constraint violations.

We illustrate the EDF approach via the example in **Figure 4**, 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 $\Gamma = \{\tau_1, \tau_2, \tau_3\}$. In this example all tasks have a jitter constraint of 0, meaning that they have to start in each period instance at the same time in relation to the start of the period. 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 4A depicts a solution, where the jitter and the task chain end-to-end constraints are violated, whereas **Figure 4B** shows a valid solution. As seen from **Figure 4A**, τ_1 violates its jitter constraints, as the start (and end) of execution within its periods varies. This is detected when the events for the respective tasks instances are raised. 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 finalizes at cycle 23, yielding a latency of 14. Given that both latency and jitter constraints have been violated, the product of the EDFScheduleSynthesis 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 4B**. 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 jitter constraint violation, 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 latencies of 10 and 20 *ms* for $\aleph_{1,1}$ and $\aleph_{1,2}$, respectively.

5 EXPERIMENTAL RESULTS

We have evaluated the proposed solutions in for both network setups, PCIe in **Section 5.1** and TSN in **Section 5.2**. We have used both realistic test cases and synthetic test cases. The synthetic test cases were generated using a tool developed for this purpose (McLean et al., 2019), which derives the desired task properties from the realistic test cases. The test case generation tool was extended to add TSN flows based on the work from (Craciunas and Serna Oliver, 2016). All experiments were conducted on a High Performance Computing (HPC) cluster, with each node configured with 2xIntel Xeon Processor 2660v3 (10 cores, 2.60 GHz) and 128 GB memory. Both SA and GA run on one node at a time.

The choice of parameters for the metaheuristics has been done using ParamILS (Hutter et al., 2009), which performs a stochastic search in the parameter space. ParamILS works by giving it a list of possible values for each parameter to be tuned. The list of values was initially chosen on a broad scale, and then if runs seemed to converge, the range was narrowed. Because the different test case sizes have varying difficulty, parameter tuning was done separately for the different test case sizes. For each type of test case, 10 parallel runs were launched with differing seeds so that more solutions could be discovered (Hutter et al., 2012).

5.1 Experimental Results for PCIe-Based Systems

As a first experiment, we were interested in determining our proposed SA's ability to find near-optimal solutions. We have implemented an exhaustive search that finds the optimal solution; however, we could only do that 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 following sets of experiments, determining the efficacy of SA was achieved through a combination of synthetic and realistic test scenarios, benchmarked against two other heuristics: Greedy, presented as the initial solution for the metaheuristics in **Section 4.2** and Genetic Algorithm (GA).

5.1.1 Synthetic Test Cases

We were then interested in determining 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 2**, whereas with *ADAS1x200%* the architecture would double the number of processors, tasks and task chains. The results are presented in **Table 1**, 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

TABLE 1 | 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 h	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 h	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 h	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 h	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 h	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

were given additional time due to an inherent increased complexity of the problem (see the *Time* column).

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 labeled *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 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 1**, the Greedy approach has comparatively the worst performance in terms of complying with the constraints. We also see that SA can 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. Both metaheuristics (SA and GA) are clearly superior to the mapping heuristic, such as Greedy, when presented with very large task sets.

5.1.2 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 set up such that 30 trials were conducted with SA for each test case; the time limit used is in minutes. As we can see from **Table 2**, 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 to find 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. We have also implemented an approach from the related work (Verucchi et al., 2020), called *DAG*, which constructs a Directed Acyclic Graph (DAG) from the input task set. The constructed DAG can handle the multiple periods of tasks in the task set and encodes the chain constraints. Such a DAG is built on the fly by our approach when constructing a solution. The DAG approach does not consider "preemption", i.e., the tasks will not be split when scheduled, and uses List Scheduling instead of EDF for scheduling the DAG. As we can see, the DAG approach from related work is similar to our Greedy approach and significantly under-performs

TABLE 2 | Evaluation results on realistic test cases.

Test case	Time	DAG			Greedy			SA						
		Chains	Jitter	Sched.	Chains	Jitter	Sched.		Chains			Jitter		Sched.
						Min	Avg	Max	Min	Avg	Max			
ADAS1	3.20	0.90	0.31	1.00	0.81	0.37	1.00	0.97	0.99	1.00	0.95	0.99	1.00	1.00
ADAS2	6.40	0.55	0.10	1.00	0.65	0.21	1.00	0.94	0.99	1.00	0.84	0.99	1.00	1.00
ADAS3	13.20	0.74	0.10	1.00	0.48	0.21	1.00	0.84	0.99	1.00	0.74	0.97	1.00	1.00

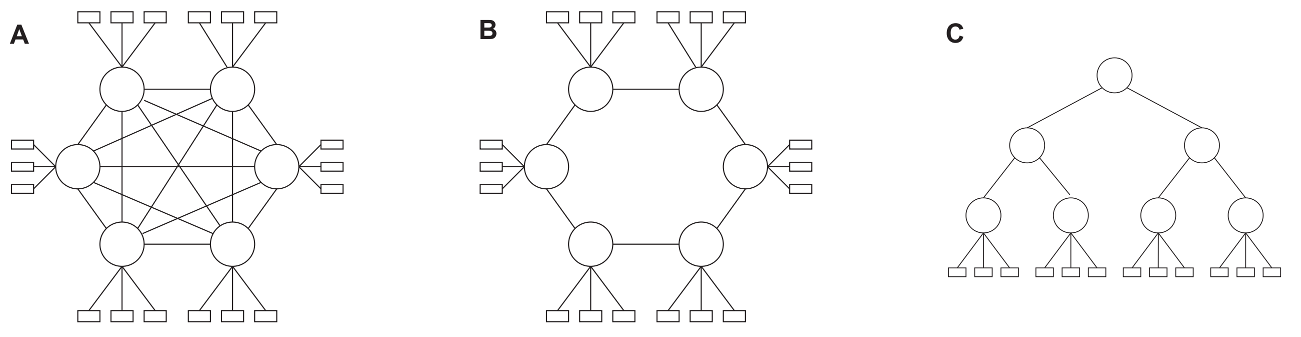

FIGURE 9 | Topologies used for experiments. In each topology, a switch has 3 end systems attached (for tree: Leaf nodes only). **(A)** Mesh topology. **(B)** Ring topology. **(C)** Tree topology with depth = 2.

compared to our SA metaheuristic. In addition, it results in fewer jitter constraints satisfied, compared to our Greedy.

5.2 Experimental Results for TSN-Based Systems

In this section, we considered that the communication is done via a TSN backbone. We have used synthetic test cases, for which we generated various TSN networks. We have used three different types of graphs with varying degrees of connectivity. The topologies (mesh, ring, and tree) are shown in **Figure 9**.

The number of end systems, switches, and chains are given in **Table 3**, similar to the setup used in (Craciunas and Serna Oliver, 2016), except for the number of chains. For each end system, 16 tasks are created, 8 which communicate and 8 which do not. Each communicating task sends a message to another communicating task. Thus, there will be $|ES|\cdot 4$ flows in the network. The utilization is set to be 50% for each end system, 25% of which corresponds to communicating tasks and 75% to the rest. Task WCETs are chosen such that they are divisible with the macrotick and fit within the assigned utilization. Message lengths are chosen at random between 84 and 1,542 Bytes. The macrotick of the end systems is set to 250 μs, and the granularity of the links are set to 1 μs. The speed of links from end systems to switches is set to 100 Mbps and to 1 Gbps between switches.

We have used three sets of randomly chosen periods, all in milliseconds, $P1 = \{10, 20, 25, 50, 100\}$, $P2 = \{10, 30, 100\}$, and $P3 = \{50, 75\}$. We use the shortest paths for routes. The chains are generated with a maximum task length of 15, and a minimum of 2, and lengths are chosen from a uniform random distribution. Two consecutive tasks in a chain must either be on the same end system or be communicating via the TSN network. In order to compare the performance between the two metaheuristics, 10 runs were performed on each of the test cases for a total of 360 runs for each of the metaheuristics. The time given for each size is as follows. 300 s for small test cases, 1,200 s for medium, 4,800 s for large, and 19,200 s for the largest test cases, called "huge".

Figure 10 shows on the y-axis the percentage of test cases solved for each size and topology. The results are grouped per topology, mesh, ring, and tree, and for each topology, we use different sets of periods, $P1$ to $P3$. A test case is "solved" if all the requirements are satisfied. On the y-axis, 1 means that 100% of the requirements were satisfied, whereas 0 means that no requirements could be satisfied. On the x-axis, we show the type of test case, small, medium, large, and huge. As we can see, our GA and SA solutions can successfully solve all the test cases, except for some of the "huge" test cases, especially in the tree topologies, where a few requirements could not be satisfied. In those situations, GA performs better than SA. When considering the cost of the solutions (the value of **Eq. 2**), we noticed that SA is better than GA in terms of the cost function for small, medium, and large test cases. However, in the huge test cases, GA not only is able to find feasible solutions more consistently but is also able to find solutions of lower cost.

Finally, we were also interested in our approaches' ability to find a feasible solution as fast as possible. That is, we wanted to determine what is the earliest time when all the requirements are satisfied. Once such a solution is found, the metaheuristics continue the optimization until the time limit is reached. Hence, we modified SA and GA to return once a feasible

TABLE 3 | Number of switches, end systems and chains for each topology and size of test case.

Size	Topology	Switches	End systems	Tasks	Chains	Flows
Small	Mesh, Ring	2	4	64	16	16
	Tree, depth = 1	4	6	96	16	24
Medium	Mesh, Ring	4	16	256	32	64
	Tree, depth = 2	13	36	576	32	144
Large	Mesh, Ring	8	48	768	64	192
	Tree, depth = 3	15	48	768	64	192
Huge	Mesh, Ring	16	192	3,072	128	768
	Tree, depth = 2	43	432	6,912	128	1,728

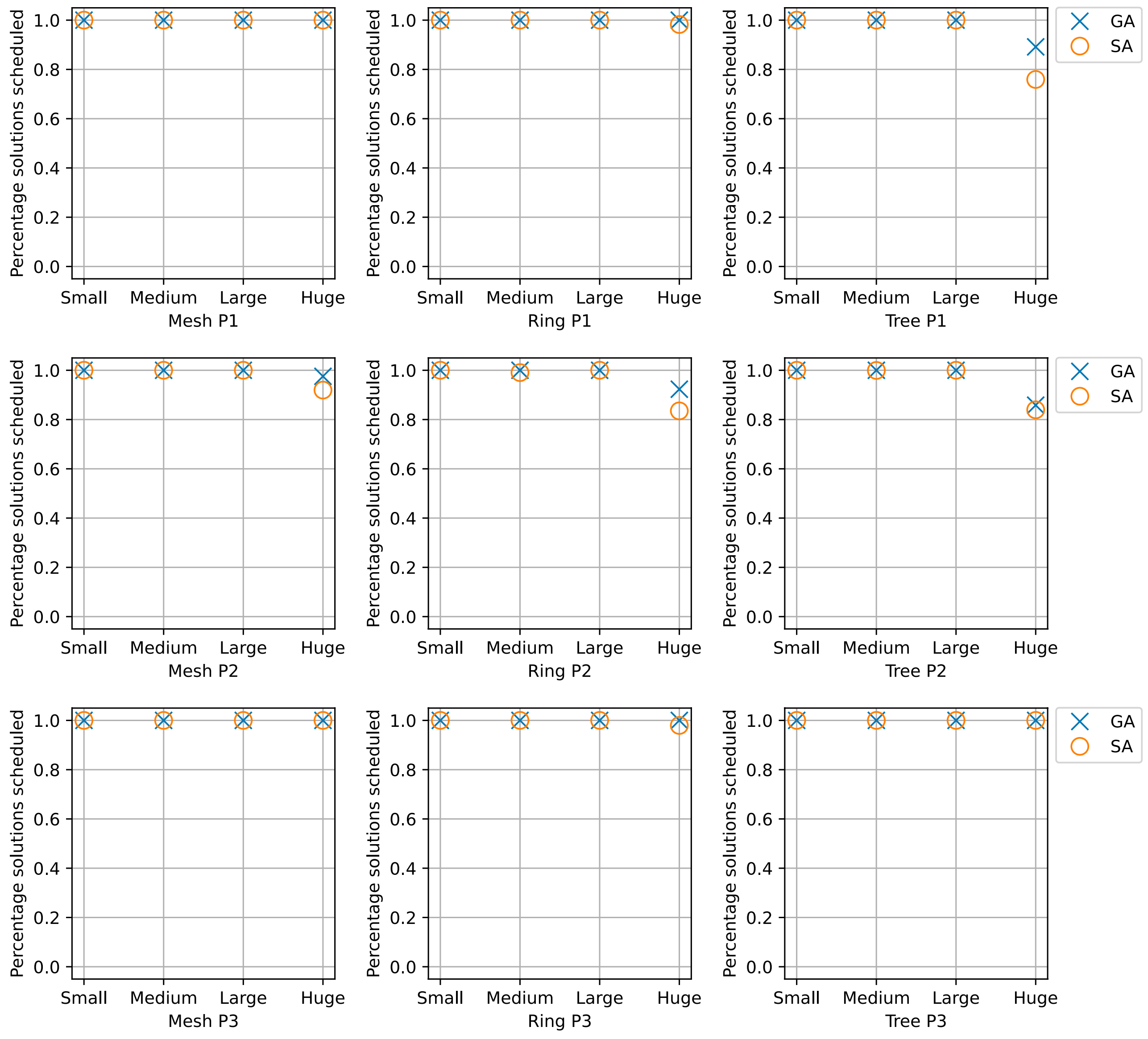

FIGURE 10 | Comparison of SA and GA in terms of percentage of solved solutions, i.e., all the tasks and flows are successfully scheduled and the constraints, e.g., chain latencies, are satisfied.

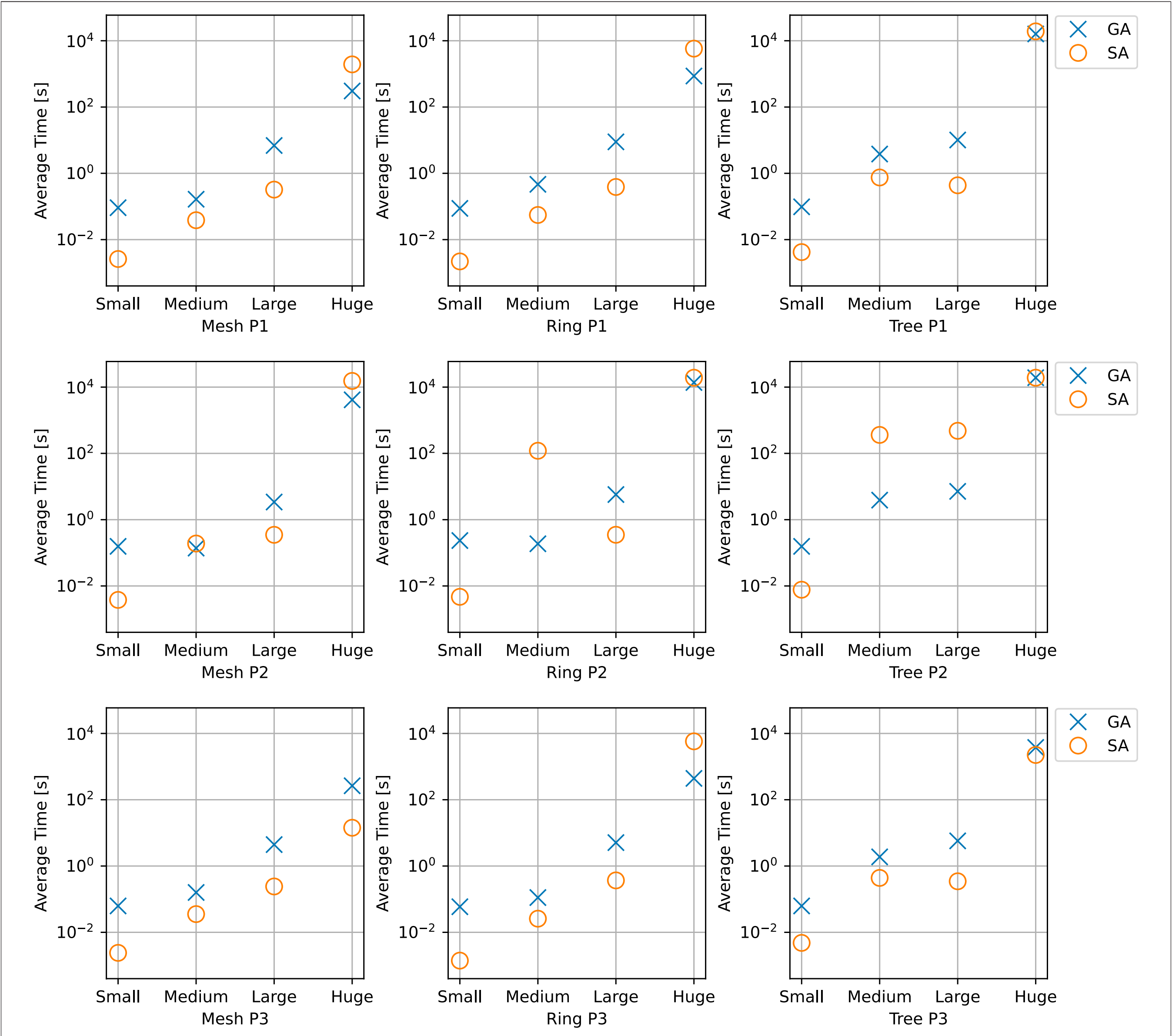

FIGURE 11 | Comparison of SA and GA runtimes when searching for the first feasible solution.

solution was found. The runtime results are shown in **Figure 11**. The figure shows that when SA can find a feasible solution for a test case, it generally finds it faster than GA. However, there are situations where GA outperforms SA.

6 CONCLUSION 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. We have also considered a realistic communication backbone implemented using the IEEE 802.1 Time-Sensitive Networking standard, and our optimization derives the schedule tables for the TSN messages.

Our optimization strategy uses metaheuristics (Simulated Annealing and Genetic Algorithm) to explore the solution space, combined with a scheduling heuristic to jointly solve the task and message 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.

Our evaluation has shown that SA is superior in finding feasible solutions fast and with a lower cost function value compared to GA, whereas GA outperforms SA for very large TSN-based test cases where. As future work, we want to implement a hybrid multi-objective metaheuristic (Blum and Roli, 2008) that combines SA and GA and considers several optimization objectives, such as reducing the number of task preemptions in order to reduce context switch overhead and reducing the number of switch queues used by the TSN messages.

DATA AVAILABILITY STATEMENT

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

AUTHOR CONTRIBUTIONS

All authors listed have made a substantial, direct, and intellectual contribution to the work and approved it for publication.

FUNDING

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).

ACKNOWLEDGMENTS

This paper is an extended version of (McLean et al., 2020) based on original technical material contained in a thesis of the first author (McLean, 2019) which is archived as a technical report (McLean et al., 2019) and extended with material included in a thesis of the second author (Hansen, 2020).

REFERENCES

Abdelzaher, T. F., and Shin, K. G. (1999). Combined Task and Message Scheduling in Distributed Real-Time Systems. *IEEE Trans. Parallel Distrib. Syst.* 10, 1179–1191. doi:10.1109/71.809575

Aichouch, M., Prévotet, J.-C., and Nouvel, F. (2013). "Evaluation of the Overheads and Latencies of a Virtualized RTOS," in 2013 8th IEEE International Symposium on Industrial Embedded Systems (IEEE), 81–84. doi:10.1109/sies.2013.6601475

Barzegaran, M., Cervin, A., and Pop, P. (2020). Performance Optimization of Control Applications on Fog Computing Platforms Using Scheduling and Isolation. *IEEE Access* 8, 104085–104098. doi:10.1109/access.2020.2999322

Becker, M., Dasari, D., Mubeen, S., Behnam, M., and Nolte, T. (2017). End-to-end Timing Analysis of Cause-Effect Chains in Automotive Embedded Systems. *J. Syst. Architecture* 80, 1. doi:10.1016/j.sysarc.2017.09.004

Becker, M., Dasari, D., Mubeen, S., Behnam, M., and Nolte, T. (2016a). "Synthesizing Job-Level Dependencies for Automotive Multi-Rate Effect Chains," in 2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (IEEE), 159–169. doi:10.1109/rtcsa.2016.41

Becker, M., Dasari, D., Nicolic, B., Akesson, B., Nelis, V., and Nolte, T. (2016b). "Contention-free Execution of Automotive Applications on a Clustered many-core Platform," in 2016 28th Euromicro Conference on Real-Time Systems (IEEE), 14–24. doi:10.1109/ecrts.2016.14

Biondi, A., and Di Natale, M. (2018). "Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm," in 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (IEEE), 240–250. doi:10.1109/rtas.2018.00032

Blum, C., and Roli, A. (2008). "Hybrid Metaheuristics: an Introduction," in *Hybrid Metaheuristics* (Springer), 1–30. doi:10.1007/978-3-540-78295-7_1

Bunzel, S. (2011). AUTOSAR - the Standardized Software Architecture. *Informatik Spektrum* 34, 79–83. doi:10.1007/s00287-010-0506-7

Burke, E. K., and Kendall, G. (2005). *Search Methodologies*. Springer.

Buttazzo, G. C. (2011). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications (Real-Time Systems Series)*. Springer-Verlag.

Chetto, H., Silly, M., and Bouchentouf, T. (1990). Dynamic Scheduling of Real-Time Tasks under Precedence Constraints. *J. Real-Time Syst.* 2, 181–194. doi:10.1007/bf00365326

Choi, S., and Agrawala, A. K. (2000). "Scheduling of Real-Time Tasks with Complex Constraints," in *Performance Evaluation: Origins and Directions* (IEEE), 253–282. doi:10.1007/3-540-46506-5_11

Craciunas, S. S., and Oliver, R. S. (2016). Combined Task- and Network-Level Scheduling for Distributed Time-Triggered Systems. *Real-time Syst.* 52, 161–200. doi:10.1007/s11241-015-9244-x

Craciunas, S. S., Serna Oliver, R., Chmelik, M., and Steiner, W. (2016). "Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks," in Proc. 24th International Conference on Real-Time Networks and Systems (IEEE), 183–192. doi:10.1145/2997465.2997470

Craciunas, S. S., Serna Oliver, R., and Ecker, V. (2014). "Optimal Static Scheduling of Real-Time Tasks on Distributed Time-Triggered Networked Systems," in Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (IEEE), 1–8. doi:10.1109/etfa.2014.7005128

Deb, K., Agrawal, S., Pratap, A., and Meyarivan, T. (2000). "A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II," in Proceedings of the International Conference on Parallel Problem Solving from Nature (IEEE), 849–858. doi:10.1007/3-540-45356-3_83

Di Natale, M., and Stankovic, J. A. (2000). Scheduling Distributed Real-Time Tasks with Minimum Jitter. *IEEE Trans. Comput.* 49, 303–316. doi:10.1109/12.844344

Dürr, F., and Nayak, N. G. (2016). "No-wait Packet Scheduling for IEEE Time-Sensitive Networks (TSN)," in Proceedings of the 24th International Conference on Real-Time Networks and Systems (IEEE), 203–212.

Ernst, R., Kuntz, S., Quinton, S., and Simons, M. (2018). The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092). *Dagstuhl Rep.* 8, 122–149.

Fohler, G. (1994). *Flexibility in Statically Scheduled Real-Time Systems. Ph.D. Thesis, Technisch- Naturwissenschaftliche Fakultät*. Vienna, Austria: Technische Universität Wien.

Forget, J., Boniol, F., and Pagetti, C. (2017). "Verifying End-To-End Real-Time Constraints on Multi-Periodic Models," in Proceedings IEEE Emerging Technology and Factory Automation (IEEE), 1–8. doi:10.1109/etfa.2017.8247612

Forget, J., Grolleau, E., Pagetti, C., and Richard, P. (2011). "Dynamic Priority Scheduling of Periodic Tasks with Extended Precedences," in Proceedings IEEE Emerging Technology and Factory Automation (IEEE), 1–8. doi:10.1109/etfa.2011.6059015

Garey, M. R., and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Gietelink, O., Ploeg, J., Schutter, B. D., and Verhaegen, M. (2006). Development of Advanced Driver Assistance Systems with Vehicle Hardware-In-The-Loop Simulations. *Vehicle Syst. Dyn.* 44, 1. doi:10.1080/00423110600563338

Goldberg, D. E., and Lingle, R., Jr (1987). "A Study of Permutation Crossover Operators on the TSP," in Proceeding of the Second International Conference on Genetic Algorithms and Their Applications. Editor J. J. Grefenstette (Hillsdale, New Jersey: Lawrence Erlbaum), 224–230.

Goldberg, D. E., and Lingle, R., Jr (1985). "Alleles, Loci and the TSP," in Proceeding of the First International Conference on Genetic Algorithms and Their Applications. Editor J. J. Grefenstette (Hillsdale, New Jersey: Lawrence Erlbaum), 154–159.

Hammond, M., Qu, G., and Rawashdeh, O. A. (2015). "Deploying and Scheduling Vision Based Advanced Driver Assistance Systems (ADAS) on Heterogeneous Multicore Embedded Platform," in 2015 9th International Conference on Frontier of Computer Science and Technology (IEEE), 172–177. doi:10.1109/fcst.2015.69

Hansen, E. A. J. (2020). *Configuration of Computer-Platforms for Autonomous Driving ApplicationsMaster's Thesis*. Kongens Lyngby, Denmark: Technical University of Denmark.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2012). "Parallel Algorithm Configuration," in International Conference on Learning and Intelligent Optimization (IEEE), 55–70. doi:10.1007/978-3-642-34413-8_5

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stützle, T. (2009). ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Intell. Res.* 36, 267–306.

IEEE (2016a). *802.1AS-Rev - Timing and Synchronization for Time-Sensitive Applications*. IEEE. Available at: https://www.ieee802.org/1/pages/802.1as.html (Accessed 08 19, 2021).

IEEE (2015). *802.1Qbv-2015 - IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic*. IEEE. Available at: https://ieeexplore.ieee.org/servlet/opac?punumber=8613093 (Accessed 08 19, 2021).

IEEE (2016b). *Official Website of the 802.1 Time-Sensitive Networking Task Group*. IEEE. Available at: https://1.ieee802.org/tsn/(Accessed 06 11, 2019).

Isović, D., and Fohler, G. (2000). "Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints," in Proceedings IEEE Real-Time Systems Symposium (IEEE), 207–216.

Larranaga, P., Kuijpers, C., Poza, M., and Murga, R. (1997). Decomposing Bayesian Networks. Triangulation of the Moral Graph with Genetic Algorithms. *Stat. Comput.* 7, 19–34.

Leung, J. Y.-T., and Merrill, M. L. (1980). A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. *Inf. Process. Lett.* 11, 115–118. doi:10.1016/0020-0190(80)90123-4

Liu, C. L., and Layland, J. W. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. Acm* 20, 46–61. doi:10.1145/321738.321743

Lukasiewycz, M., Schneider, R., Goswami, D., and Chakraborty, S. (2012). "Modular Scheduling of Distributed Heterogeneous Time-Triggered Automotive Systems," in 17th Asia and South Pacific Design Automation Conference (IEEE), 665–670. doi:10.1109/aspdac.2012.6165039

Dataset Marija Sokcevic (2020). Partitioned Complexity. Available at: https://www.tttech-auto.com/expert_insight/expert-insights-partitioned-complexity/ (Accessed 06 15, 2021).

McLean, S. D., Craciunas, S. S., Hansen, E. A. J., and Pop, P. (2020). "Mapping and Scheduling Automotive Applications on ADAS Platforms Using Metaheuristics," in 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (IEEE), 329–336. doi:10.1109/etfa46521.2020.9212029

McLean, S. D. (2019). *Mapping and Scheduling of Real-Time Tasks on Multi-Core Autonomous Driving platformsMaster's Thesis*. Kongens Lyngby, Denmark: Technical University of Denmark.

McLean, S. D., Pop, P., and Craciunas, S. S. (2019). *Mapping and Scheduling of Real-Time Tasks on Multi-Core Autonomous Driving Platforms*. Kongens Lyngby, Denmark: Tech. rep., Technical University of Denmark.

Mehmed, A., Steiner, W., and Rosenblattl, M. (2017). "A Time-Triggered Middleware for Safety-Critical Automotive Applications," in Presented at the 22nd International Conference on Reliable Software Technologies—Ada-Europe (IEEE).

Mubeen, S., and Nolte, T. (2015). "Applying End-To-End Path Delay Analysis to Multi-Rate Automotive Systems Developed Using Legacy Tools," in 2015 IEEE World Conference on Factory Communication Systems (IEEE), 1–4. doi:10.1109/wfcs.2015.7160585

Niedrist, G. (2018). "Deterministic Architecture and Middleware for Domain Control Units and Simplified Integration Process Applied to ADAS," in *Fahrerassistenzsysteme 2016* (Wiesbaden: Springer Fachmedien Wiesbaden), 235–250. doi:10.1007/978-3-658-21444-9_15

Peng, D.-T., Shin, K. G., and Abdelzaher, T. F. (1997). Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems. *IIEEE Trans. Softw. Eng.* 23, 745–758. doi:10.1109/32.637388

Pop, P., Raagaard, M. L., Craciunas, S. S., and Steiner, W. (2016). Design Optimisation of Cyber-physical Distributed Systems Using IEEE Time-sensitive Networks. *IET Cyber-phys. Syst.* 1, 86–94. doi:10.1049/iet-cps.2016.0021

Pop, T., Eles, P., and Peng, Z. (2003). "Schedulability Analysis for Distributed Heterogeneous Time/event Triggered Real-Time Systems," in 15th Euromicro Conference on Real-Time Systems (IEEE), 257–266.

Raagaard, M. L., and Pop, P. (2017). *Optimization Algorithms for the Scheduling of IEEE 802.1 Time-Sensitive Networking (TSN)*. Kongens Lyngby, Denmark: Tech. rep., Technical University of Denmark.

Rajeev, A. C., Mohalik, S., Dixit, M. G., Chokshi, D. B., and Ramesh, S. (2010). "Schedulability and End-To-End Latency in Distributed ECU Networks: Formal Modeling and Precise Estimation," in Proceedings of the Tenth ACM International Conference on Embedded Software (IEEE), 129–138.

Sagstetter, F., Andalam, S., Waszecki, P., Lukasiewycz, M., Stähle, H., Chakraborty, S., et al. (2014). "Schedule Integration Framework for Time-Triggered Automotive Architectures," in Proceedings of the 51st Annual Design Automation Conference (IEEE), 1–6. doi:10.1145/2593069.2593211

Serna Oliver, R., Craciunas, S. S., and Steiner, W. (2018). "IEEE 802.1Qbv Gate Control List Synthesis Using Array Theory Encoding," in 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (IEEE), 13–24. doi:10.1109/rtas.2018.00008

Sinnen, O. (2007). *Task Scheduling for Parallel Systems*. Wiley & Sons.

Sommer, S., Camek, A., Becker, K., Buckl, C., Zirkler, A., Fiege, L., et al. (2013). "RACE: A Centralized Platform Computer Based Architecture for Automotive Applications," in 2013 IEEE International Electric Vehicle Conference (IEVC) (IEEE), 1–6. doi:10.1109/ievc.2013.6681152

Steiner, W., Bauer, G., Hall, B., and Paulitsch, M. (2011). "TTEthernet: Time-Triggered Ethernet," in *Time-Triggered Communication* (Boca Raton, United States: CRC).

Syswerda, G. (1991). "Schedule Optimization Using Genetic Algorithms," in *Handbook of Genetic Algorithms* (New York: Van Nostrand Reinhold), 332–349.

Tindell, K., and Clark, J. (1994). Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Microprocess. Microprogram* 40, 1. doi:10.1016/0165-6074(94)90080-9

Dataset TTTech Computertechnik AG (2018). Automated Driving Offering. Available at: https://www.tttech-auto.com/products/automated-driving/.

Verucchi, M., Theile, M., Caccamo, M., and Bertogna, M. (2020). "Latency-aware Generation of Single-Rate Dags from Multi-Rate Task Sets," in 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (IEEE), 226–238. doi:10.1109/rtas48715.2020.000-4

Zuepke, A., Bommert, M., and Lohmann, D. (2015). "AUTOBEST: a United AUTOSAR-OS and ARINC 653 Kernel," in 21st IEEE Real-Time and Embedded Technology and Applications Symposium (IEEE), 133–144. doi:10.1109/rtas.2015.7108435

Conflict of Interest: Author SC was employed by the company TTTech Computertechnik AG.

The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.