Evolvement of Scheduling Theories for Autonomous Vehicles

Wanli Chang, Nan Chen, Shuai Zhao, and Xiaotian Dai

1 Introduction

There is a clear trend in the automotive industry towards autonomous vehicles which brings a series of new requirements for real-time scheduling, due to the evolving complexity. First, in the scheduling of real-time autonomous systems, scheduling theories for simple task models and uniprocessors have been well established, but multiprocessor systems are increasingly being employed and dependencies between tasks need to be considered [10]. Many existing works use a single recurrent event or time-triggered DAG tasks to model functional dependencies in a system [7, 8, 26, 46, 59, 60]. For example, a complete automotive task chain from on to control is described in [59] and converted to a single periodic DAG task. In addition, to avoid migration and cache-related preemption overhead, a non-preemptive global scheduling scheme is often deployed [15, 59]. That is, the nodes of a DAG are scheduled globally on all cores and preemption is not allowed during the execution of a node [47].

Figure 1 provides an example DAG which contains eight nodes with a set of edges. A node indicates a computation unit that must be executed sequentially and a directed edge describes the execution dependency of two nodes (e.g., node v_5

W. Chang (✉)
Hunan University, Changsha, China

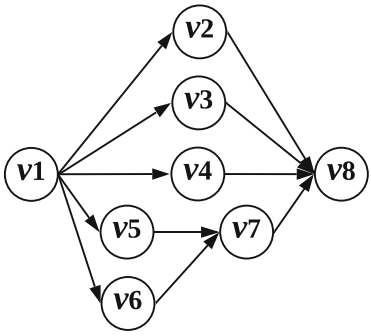
Huawei Technologies, Shenzhen, China

N. Chen · X. Dai
University of York, York, UK
e-mail: nc952@york.ac.uk; xiaotian.dai@york.ac.uk

S. Zhao
Sun Yat-sen University, Guangzhou, China
e-mail: zhaosh56@mail.sysu.edu.cn

V. K. Kukkala, S. Pasricha (eds.), *Machine Learning and Optimization Techniques for Automotive Cyber-Physical Systems*, https://doi.org/10.1007/978-3-031-28016-0_2

Fig. 1 An example DAG



and v_7). When there are adequate cores in the system, nodes with no dependency e.g., node v_2, v_3 and v_4 can be executed in parallel. However, when the number of paralleled nodes is bigger than the number of cores available, the priority ordering between nodes becomes an issue which can impose non-negligible effects to the makespan (i.e. the execution between the start of the first node and finish time of the last node) of a DAG. In the mean time, the Worst-Case Response Time (WCRT) analysis in [33, 39] are pessimistic which can result in low system schedulability. Hence, a fine-grained scheduling policy and a less pessimistic WCRT bound are necessary.

Second, the increasing demand of autonomous systems to realize both complex functionality and high performance with limited resources necessitates extensive resource sharing. For example, to facilitate partially or fully automated driving, the AUTOSAR Classic standard (which implements static task configuration with resource isolation) is evolving to AUTOSAR Adaptive with dynamic resource sharing on multiprocessor architectures [4]. Resources sharing is referred as sharing data structures, special memory locations, and code segments, which need to be accessed in a mutually exclusive fashion. Consequently, the increasing applications of shared resources in the autonomous systems can cause blocking due to contention, while conventional requirements of timing predictability and reliability still need to be satisfied. That is, the deadlines of tasks must be met while failures during task executions must be resolved.

Satisfying both timing and reliability requirements is particularly hard. Several multiprocessor resource sharing protocols have been proposed to bound and minimize blocking time, including MSRP [27] and MrsP [14]. However, reliability has not been accounted for, which is imperative in safety-critical scenarios like autonomous systems. The common fault-tolerance methods are based on redundancy, and they may be directly applied to shared resources by scheduling repeated task executions and resource accesses a sufficient number of times to get the correct output. However, this leads to severe resource contention and undermines system schedulability. Therefore, a solution for guaranteeing both reliability and schedulability for autonomous systems with the presence of shared resource is required.

Third, on communication, Ethernet as a data link layer protocol has evolved from standard computer networks to applications of in-vehicle communication

(e.g., deterministic real-time Ethernet [55]). In the emerging safety-critical systems such as highly automated vehicles, a large volume of messages with mixed types need to be transmitted on the same infrastructure, which requires deterministic and predictable timing to guarantee safety. Traditional real-time networks use non-standard Ethernet to enable high-bandwidth deterministic communication, which prohibits connectivity between different protocols and components from different vendors, as well as increases uncertainty and difficulty in timing and hazard analysis.

TSN proposed as an IEEE standard, offers an interoperable and flexible deterministic Ethernet-based solution [36]. It is widely considered as the network solution for future automobiles. The IEEE 802.1 TSN standard includes a wide range of subsets, in which one of the most important protocols is the 802.1Qbv [20, 35, 63]. The IEEE 802.1Qbv supports time-aware shaper (TAS) using TDMA (time-division multiple access)-scheduled queues to access the egress port—controlled by a gate switching logic that is driven by a synchronized global timer and a look-up scheduling table.

Control loops are often involved in the safety-critical systems, where guarantees are required on both timing of communication and control performances (measured by settling time). In general, short sampling periods enable the potential to achieve good control performance with frequent interactions between the controller and the plant. The state-of-the-art network scheduling techniques for TSN (e.g., [5, 41, 63]) cannot be directly applied, as they consider neither the hard real-time constraints on network packets nor the control performance of the system. Therefore, an integrated solution of network scheduling and controller co-design for TSN is essential for autonomous in-vehicle communications from the CPS perspective.

1.1 Organization

In this chapter, we present three interconnected fundamental works along the above directions: the real-time scheduling for DAGs on multiprocessor architectures; the reliable resource sharing in autonomous systems; and real-time scheduling and controller co-design for TSN. The rest of the chapter is organized as follows:

- **Section 2** provides the background knowledge and related research outputs of the work presented in the following sections.
- **Section 3** introduces a CPC model based on the work-conserving schedule and the classic analysis, alongside a priority ordering algorithm.
- **Section 4** presents the first fault-tolerant solution for multiprocessor MCS with shared resources. The solution contains a system execution model that is compatible with an arbitrary number of criticality levels, and a protocol, namely Multiprocessor Stack Resource Protocol Fault Tolerance (MSRP-FT) which aims to address faults during critical sections while minimizing blocking time.
- **Section 5** presents the first integrated solution of network scheduling and controller co-desig for TSN 802.1Qbv. Specifically, the first FPS approach for TSN is demonstrated. Moreover, a finer-grained analysis for the above scheduling

approach at the frame level is also included. Based on FPS and the analysis, we formulate a co-design optimization problem to decide the sampling periods and poles of real-time controllers.
- **Section 6** concludes the contents of this chapter.

2 Background

In this section, we provide the background information and related literature to motivate the research output demonstrated in the following sections. First, Sect. 2.1 reviews the work in scheduling and analysis of DAG tasks. Second, work related to fault-tolerance, resource sharing, and MCS is reviewed in Sect. 2.2. Last, relevant literature on the scheduling of TSN network is presented in Sect. 2.3.

2.1 Scheduling and Analyzing DAG Tasks in Autonomous Vehicles

The majority of the existing work on scheduling DAG tasks assumes a *work-conserving* scheduler [39]. A scheduling algorithm is said to be work-conserving if it never idles a processor when there exists pending workload. A generic bound that captures the worst-case response time of tasks scheduled globally with any work-conserving method is provided in Graham [28]. This analysis is later formalized in Melani [39] and Fonseca [25] for DAG tasks. The analysis of a single DAG task is given in Eq. (1). Notation τ_x denotes a DAG task with index x, R_x denotes the response time of τ_x, L_x denotes the length of the longest path in the DAG, W_x gives the sum of Worst-Case Execution Time (WCETs) of all nodes in the DAG, and m denotes the number of cores.

$$R_x = L_x + \left\lceil \frac{1}{m}(W_x - L_x) \right\rceil \tag{1}$$

In this analysis, the worst-case response time of a DAG task τ_x is upper bounded by the length of the critical path and the intra-task interference imposed by the non-critical nodes of τ_x itself. However, this analysis assumes the critical path can be delayed by all the concurrent nodes, which is pessimistic for scheduling methods with an explicit execution order known a priori [33, 39].

2.1.1 The State-of-the-Art in DAG Scheduling and Analysis

For homogeneous multiprocessors with a global scheme, existing scheduling (and their analysing) methods aim at reducing the makespan and tightening the worst-case analytical bound. They can be classified as either slice-based [17, 29] or node-based [18, 33]. The slice-based schedule enforces node-level preemption and

divides each node into a number of small computation units (e.g., units with a WCET of one in Chang [17]). By doing so, the slice-based methods can improve node-level parallelism but to achieve an improvement the number of preemptions and migrations need to be controlled.

The node-based methods provide a more generic solution by producing an explicit node execution order, based on heuristics derived from either the *spatial* (e.g., number of successors of a node [37] and topological order of nodes [33]) or the *temporal* (execution time of nodes [18, 54, 59]) characteristics of the DAG. Below we describe two most recent node-based methods.

In Chen et al. [18], an non-preemptive scheduling method is proposed for a single periodic DAG, which always executes the ready node with the longest WCET to improve parallelism. Chen [18] prevents anomalies from occurring when nodes are executing less than their WCETs, which can lead to an execution order different from the schedule. This is achieved by guaranteeing nodes are executed in the same order as the offline simulation. However, without considering inter-node dependencies, this schedule cannot minimize the delay on the completion of DAG.

In He et al. [33], a new response time analysis is presented, which dominates the traditional bound in Graham [28] and Melani [39] when an explicit node execution order is known a priori. That is, a node v_j can only incur a delay from the concurrent nodes that are scheduled prior to v_j. Then, a scheduling method is proposed that always executes: (i) the critical path first; and (ii) the immediate interference nodes first (nodes that can cause the most immediate delay on the currently-examined path). The novelty in He [33] is considering both topology and path length in a DAG, and provides the state-of-the-art analysis against which our approach is compared. However, the method in He [33] schedules concurrent nodes based on the length of their longest complete path (a path from the source to the sink node), i.e., nodes in the longest complete path first. This heuristic is not dependency-aware, which reduces the level of parallelism that can be exploited, and hence, lengthen the finish time of a DAG task.

2.2 Real-Time Scheduling for Reliable Autonomous Driving

In this subsection, the background information and related work about real-time scheduling of reliable autonomous system are provided. More specifically, Sect. 2.2.1 introduces common faults and solutions in the embedded systems, Sect. 2.2.2 presents the research in the field of resources sharing protocols. Section 2.2.3 demonstrates the research output related to MCS.

2.2.1 Fault Tolerance

Faults in modern embedded systems can be broadly categorized as *permanent* or *transient* faults. Transient faults affect the functionality of systems for a short period of time, where permanent faults happen repeatedly and cannot be easily recovered

from. Some software faults (bugs) are caused by erroneous program design, are permanent faults, and cannot be recovered by re-starting the operation [58]. Other software errors can be transient faults caused by unexpected interference among threads, and may be resolved by restarting the program [40]. Transient hardware faults can occur due to issues such as power supply fluctuations or electromagnetic interference which happen increasingly more frequently due to the decrease in transistor size and operating voltage [32]. Permanent hardware faults are the result of hardware damage or wear, and cannot be dealt with until the faulty component is replaced. In this chapter, we focus on transient faults which can be recovered by retrying the operation.

Three mainstream redundancy techniques are widely adopted in the literature to tolerate faults: *re-execution* [1], *checkpointing* [19], and *replication* [45]. The *re-execution* approach saves task status at the beginning and detects faults at the end. Once a fault is detected, the roll-back technique is applied and the whole task is re-executed. The *checkpointing* technique introduces additional checkpoints in a task and normally divides task execution into a set of uniform segments. Each small segment is tested for faults, and when a fault is detected the system rolls back to the most recent checkpoint and only re-executes the faulty segment. With replication, each task is replicated to several copies. The task and its replicas are released simultaneously and execute in parallel. When an execution finishes without incurring faults, the others are discarded.

Generally, fault detection mechanisms focus on analyzing the outputs of an execution. For example, in a lockstep dual-core architecture [50] or Triple Modular Redundancy architecture [4], multiple identical cores execute the same code and the system applies a majority vote to find the faulty component. Acceptance tests are often applied at the checkpoint to determine the correctness of an operation by checking a set of conditions that are expected to be met if the program has executed correctly [44]. In contrast, another type of fault-detection mechanism focuses on detecting the stimulus of the fault instead of the computation results. For example, acoustic wave detectors are adopted in the hardware architecture [56] to detect particle strikes that can result in transient faults during computation. Instead of using built-in hardware to detect faults, the Argus approach [38] uses detection equipment to monitor the variations of the circuits. Detailed descriptions and comparisons of such type of detecting mechanisms are included in [57].

2.2.2 Resource Sharing

Resource sharing in multicore real-time systems has been extensively studied in the past few decades with numerous resource sharing protocols available [2, 14, 27]. A comprehensive survey can be found in [11]. Here we describe the Multiprocessor Stack Resource Protocol (MSRP) [27].

The MSRP is a First-In-First-Out (FIFO) spin-based resource sharing protocol developed for fully-partitioned systems. In MSRP, each global resource (i.e., shared between cores) is associated with a FIFO queue. A task requesting a global resource

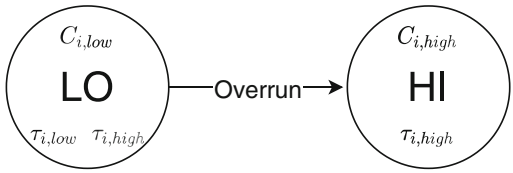


Fig. 2 The AMC model

is placed in the FIFO queue and busy-waits (spins) non-preemptively until it moves to the head of the queue, at which point it will be granted the resource. The task then keeps executing non-preemptively until it releases the resource. For a local resource (i.e., shared in one core), a priority ceiling is applied, which equals the highest priority of tasks that request the resource. A task raises its priority to the ceiling during the entire access to the local resource.

When contending for shared resources, tasks will incur additional waiting time (i.e. blocking) due to mutually exclusive executions. The blocking effects incurred by tasks for accessing shared resources under MSRP can be classified as *spin delay* and *arrival blocking* [62]. With shared resources, a task can incur spin delay either *directly* or *indirectly*. *Direct spin delay* occurs when a task is being blocked directly for accessing a shared resource by other resource accesses issued from remote cores. In this case, the task is added at the tail of the FIFO queue and spin-waits until it is granted the resource. A task incurs *indirect spin delay* when it is preempted by a local higher priority task, which in turn is blocked directly from accessing a resource. *Arrival blocking* occurs when a task is released but is then immediately blocked by a local low priority task which is running non-preemptively (resp. with a higher resource ceiling) for accessing a global (resp. local) resource.

Resource sharing protocols define rules for accessing shared resources and bound the blocking delay [11]. However, they are not developed with a particular focus on system reliability, in which a resource request has to be potentially executed multiple times sequentially to tolerate faults. Hence, the additional blocking time imposed for addressing faults cannot be effectively minimized by these protocols. Based on the above, this chapter focuses on fault-tolerance for shared resources in MCS and aims to reduce the additional blocking from tolerating faults.

2.2.3 Mixed Criticality System

Baruah et al. [6] propose an Adaptive Mixed Criticality (AMC) model which is widely regarded as the most effective approach within Fixed-Priority Preemptive Scheduling [34]. The AMC model has two system modes (LO and HI) for the system that has tasks with two criticality levels (i.e., $\mathcal{L} \in low, high$). As shown in Fig. 2, the system starts in LO mode and all tasks are allowed to execute up to $C_{i,low}$. If a task overruns these budgets, the system upgrades to the HI mode (a mode switch), in which high-criticality tasks are allowed to execute with a larger budget $C_{i,high}$ and low-criticality tasks are suspended. The AMC model assumes system

can monitor the running time of tasks and can be extended to have an arbitrary number of system modes according to the number of criticality levels in the system. Later on, concerning the quality of service (QoS) of low-criticality tasks after a mode switch, instead of dropping tasks brutally, many research [13, 30] propose mechanisms for MCS to degrade low-criticality tasks gracefully.

With the presence of faults, Pathan [42] proposes a mixed-criticality fault-tolerant algorithm called FTMC for systems with two criticality levels. In FTMC, the system would transit from a low-criticality mode to a high-criticality mode if any overrun happens or the number of transient faults incurred in the system exceeds a predefined threshold. Chen et al. [19] propose an online fault-tolerant MCS scheduling framework called the FTS-RHS. The framework applies the checkpointing recovery schemes which outperforms re-execution in scheduling. In addition, the DVFS techniques have been applied in MCS in [9] to provide systems with precise real-time and energy-efficient scheduling. Safari et al. [45] further extend the research topic by including the consideration of energy consumption in fault-tolerant MCS and propose a LETR-MC scheme for a system with two criticality levels.

With shared resources, Burns [12] applies the Original Priority Ceiling Protocol (OPCP) to the MCS on a uni-processor platform with two criticality levels. When the system transits to the high-criticality mode, low-criticality resource holders which are computing with the ceiling priority are suspended. They can continue to execute by inheriting the execution budget of their next release. Zhao et al. [61] extend the Priority Ceiling Protocol (PCP) [48] to HLC-PCP (Highest-Locker Criticality, Priority-Ceiling Protocol) to manage resource sharing in the MCS under AMC scheme. Han et al. [31] migrate the MSRP to the MCS and develop a criticality-aware utilization bound. However, none of the above works consider the presence of both shared resources and faults.

2.3 Real-Time TSN Scheduling for Automotive CPS

Time-sensitive networking is an enabler for Ethernet-based communication services that were not originally built to support hard real-time guarantees, such as OPC Unified Architecture (OPC-UA)[1] and Distributed Data Service (DDS).[2] The objective of TSN is to reduce the worst-case end-to-end latency for critical traffics. Here we briefly discuss the IEEE 802.1Qbv TSN (referred to as Qbv in the following text). A diagrammatic view of a Qbv-enabled switch is depicted in Fig. 3. From the figure, it can be seen that a Qbv TSN switch consists of the following major components:

[1] https://opcfoundation.org/about/opc-technologies/opc-ua/.

[2] https://www.omg.org/spec/DDS/1.4/PDF.

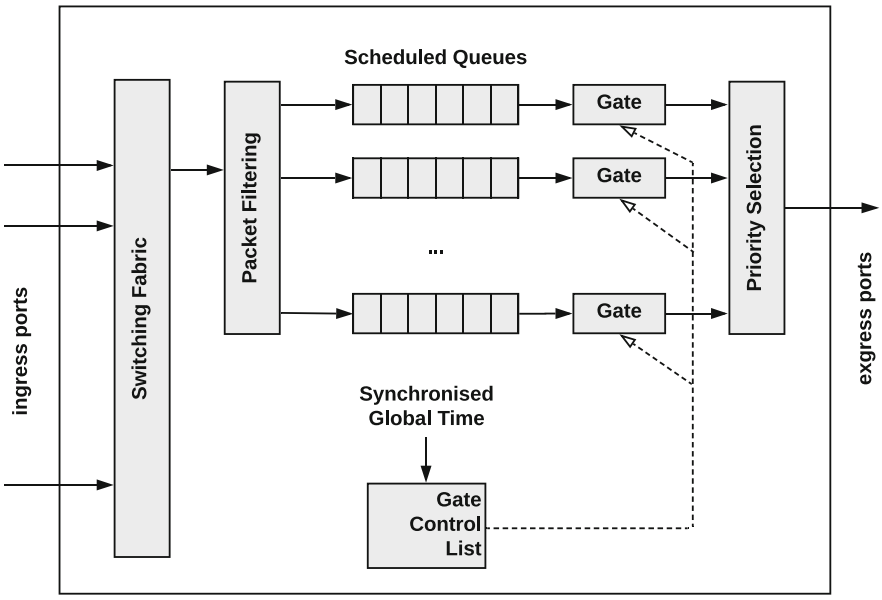


Fig. 3 An overview structure of a 802.1Qbv-capable TSN switch

- **Scheduled FIFO queues**: In a Qbv-enabled TSN switch, there are eight independent time-divided FIFO queues which are controlled by transmission gates. The incoming traffic is filtered by the packet filtering unit which sends a packet to its designated queue. This information is encoded as Class of Service (CoS) in the priority code point (PCP) header in the Ethernet frame.
- **Gate control list (GCL)**: The GCL can trigger gate-open and gate-close events periodically with a gate control cycle. The time granularity between events can be as low as $1ns$ depending on the specific implementation. The schedule is located in a GCL look-up table that is distributively configured to each TSN node. If multiple gates are opened at the same time, the policy in the priority selection unit will determine which queue is forwarded to the egress port first.
- **Time synchronization**: To allow time-divided transmission that is distributed through the network, a timer is globally synchronized with all the switches in the same network using precision time protocols (PTPs), e.g., IEEE 802.1AS or IEEE 802.1AS-Rev.

The mechanisms of Qbv TSN improve the flexibility in terms of traffic schedule and control. It enables interoperability between standard-compliant industrial devices thus allowing open data exchange. It also removes the need for physical separation of critical and non-critical communication networks. However, in a different aspect these introduce increased design complexity that needs to be elaborately handled.

3 Scheduling of DAGs on Multiprocessor Architectures

The content of this section is organized as follows. Section 3.1 presents the system and task model. Section 3.2 presents the CPC model that captures the two key factors of the DAG structure. Finally, Sect. 3.3 describes the scheduling algorithm for DAG tasks, based on the CPC model.

3.1 Task Model and Scheduling Preliminaries

A DAG task τ_x is defined by $\{T_x, D_x, \mathcal{G}_x = (V_x, E_x)\}$, with T_x denoting its minimum inter-arrival time, D_x gives a constrained relative deadline, i.e., $D_x \leq T_x$, and $\mathcal{G}_x$ is a graph defining the set of activities forming the task. The graph is defined as $\mathcal{G}_x = (V_x, E_x)$ where V_x denotes the set of nodes and $E_x \subseteq (V_x \times V_x)$ gives the set of directed edges connecting any two nodes. Each node $v_{x,j} \in V_x$ represents a computation unit that must be executed sequentially and is characterized by its Worst-Case Execution Time (WCET), $C_{x,j}$. For simplicity, the subscript of the DAG task (i.e., x for τ_x) is omitted when the system has only one DAG task.

For any two nodes v_j and v_k connected by a directed edge ($(v_j, v_k) \in E$), v_k can start execution only if v_j has finished its execution. That is, v_j is a *predecessor* of v_k, whereas v_k is a *successor* of v_j. A node v_j has at least one predecessor $pre(v_j)$ and at least one successor $suc(v_j)$, formally defined as $pre(v_j) = \{v_k \in V \mid (v_k, v_j) \in E\}$ and $suc(v_j) = \{v_k \in V \mid (v_j, v_k) \in E\}$, respectively. Nodes that are either *directly* or *transitively* predecessors and successors of a node v_j are termed as its ancestors $anc(v_j)$ and descendants $des(v_j)$ respectively. A node v_j with $pred(v_j) = \varnothing$ or $succ(v_j) = \varnothing$ is referred to as the *source* v_{src} or *sink* v_{sink} respectively. Without loss of generality, we assume each DAG has one source and one sink node. Nodes that can execute concurrently with v_j are given by $C(v_j) = \{v_k | v_k \notin (anc(v_j) \cup des(v_j)), \forall v_k \in V\}$ [33].

A DAG task has the following fundamental features. First, a path $\lambda_a = \{v_s, \cdots, v_e\}$ is a node sequence in V and follows $(v_k, v_{k+1}) \in E, \forall v_k \in \lambda_a \backslash v_e$. The set of paths in V is defined as Λ_V. A *local path* is a sub-path within the task and as such does not feature both the source v_{src} and the sink v_{sink}. A *complete path* features both. Function $len(\lambda_a) = \sum_{\forall v_k \in \lambda_a} C_k$ gives the length of λ_a. Second, the longest complete path is referred to as the *critical path* λ^*, and its length is denoted by L, where $L = \max\{len(\lambda_a), \forall \lambda_a \in \Lambda_V\}$. Nodes in λ^* are referred to as the *critical nodes*. Other nodes are referred to as *non-critical nodes*, denoted as $\vec{V} = V \backslash \lambda^*$. Finally, the workload W is the sum of a task's WCETs, i.e. $W = \sum_{\forall v_k \in V} C_k$. The workload of all non-critical nodes is referred to as the *non-critical workload*.

3.2 Concurrent Provider and Consumer Model

Equation (1) indicates that minimizing the delay from non-critical nodes to the critical path (i.e., $\frac{1}{m}(W - L)$) effectively reduces makespan of the DAG. Achieving this requires the complete knowledge of the topology (i.e., the dependency and parallelism of each node) of a DAG so that the potential delay of the critical path can be identified. To support this the CPC model is presented to fully exploit node dependency and parallelism.

The CPC model has two key stages. First, the critical path is divided into a set of consecutive sub-paths based on the potential delay it can incur. Second, for each sub-path, the CPC model identifies the non-critical nodes that can 1) execute in parallel with the sub-path and 2) delay the start of the next sub-path, based on precedence constraints.

The intuition of the CPC model is: when the critical path is executing, it utilizes just one core so that the non-critical ones can execute in parallel on the remaining $(m - 1)$ cores. The time allowed for executing non-critical nodes in parallel is termed as the *capacity*, which is the length of the critical path. Note that non-critical nodes that utilize this capacity to execute cannot cause any delay to the critical path. The sub-paths in the critical path are termed *capacity providers* Θ^* and all non-critical nodes are *capacity consumers* Θ. For each provider $\theta_i^* \in \Theta^*$, it has a set of consumers $F(\theta_i^*)$ that can execute using θ_i^*'s capacity as well as delay the next provider θ_{i+1}^* in the critical path.

Algorithm 1 presents a two-step process for constructing the CPC model of an input DAG $\mathcal{G}$ with its critical path λ^*. Starting from the head node in λ^*, capacity providers are formed by analyzing node dependency between the critical path and non-critical nodes (Line 3-9). For a provider θ_i^*, its nodes should execute consecutively without delay from non-critical nodes in terms of dependency. That is, each node in θ_i^*, other than the head node (Line 5), only has one predecessor which is the previous node in θ_i^*.

Then, for each $\theta_i^* \in \Theta^*$, its consumers $F(\theta_i^*)$ are identified as the nodes that (1) can execute concurrently with θ_i^*, and (2) can delay the start of θ_{i+1}^* (i.e., $anc(\theta_{i+1}^*) \cap \overrightarrow{V}$ in Line 12). Accordingly, nodes in $F(\theta_i^*)$ that finish later than θ_i^* will delay the start of θ_{i+1}^* (if it exists). By doing so, the CPC model provides detailed knowledge of the potential delay caused by non-critical nodes on the critical path.

Furthermore, given an arbitrary DAG structure, a consumer $v_j \in F(\theta_i^*)$ can start earlier than, synchronous with, or later than the start of θ_i^*. For synchronous and late-released consumers, they will only utilize the capacity of θ_i^*. However, an early-released consumer can execute concurrently with certain previous providers, and therefore interfere with their consumers and impose an indirect delay to those providers. For a provider θ_i^*, $G(\theta_i^*)$ (in line 13) denotes the nodes that belong to the consumer groups of later providers, but which can execute in parallel (in terms of topology) with θ_i^*.

Algorithm 1: $CPC(\mathcal{G}, \lambda^*)$: CPC model construction

Inputs : $\{\mathcal{G} = (V, E)\}$
Outputs : $\Theta^*,\ F(\theta_i^*), G(\theta_i^*), \forall \theta_j^* \in \Theta^*$
Parameters : $\lambda^*, \overrightarrow{V} = V \backslash \lambda^*$

$\Theta^* = \varnothing$;
for each $v_j \in \lambda^*$, in topological order **do**
  $\theta_i^* = \{v_j\}$; $\lambda^* = \lambda^* \backslash v_j$;
  while $pre(v_{j+1}) = \{v_j\}$ **do**
    $\theta_i^* = \theta_i^* \cup \{v_{j+1}\}$; $\lambda^* = \lambda^* \setminus v_j$;
  end
  $\Theta^* = \Theta^* \cup \theta_i^*$;
end
for each $\theta_i^* \in \Theta^*$, in topological order **do**
  $F(\theta_i^*) = anc(\theta_{i+1}^*) \cap \overrightarrow{V}$;
  $G(\theta_i^*) = \bigcup_{v_j \in F(\theta_i^*)} \{C(v_j) \cap \overrightarrow{V}\}$;
  $\overrightarrow{V} = \overrightarrow{V} \setminus F(\theta_i^*)$;
end
return $\Theta^*, F(\theta_i^*), G(\theta_i^*),\ \forall \theta_i^* \in \Theta^*$

With the CPC model, a DAG is transformed into a set of capacity providers and consumers, with a time complexity of $O(|V| + |E|)$. The CPC model provides complete knowledge of both direct and indirect delays from non-critical nodes on the critical path. For each provider θ_i^*, nodes in $F(\theta_i^*)$ can utilize a capacity of $len(\theta_i^*)$ on each of $m-1$ cores to execute in parallel while incurring potential delay from $G(\theta_i^*)$.

We now formally define the parallel and interfering workload of a capacity provider. Let $f(\cdot)$ denote the finish time of a provider θ_i^* or a consumer node v_j, $L_i = len(\theta_i^*)$ gives the length of θ_i^* and $W_i = L_i + \sum_{v_k \in F(\theta_i^*)} \{C_k\} + \sum_{v_k \in G(\theta_i^*)} \{C_k\}$ gives the total workload of θ_i^*, $F(\theta_i^*)$ and $G(\theta_i^*)$. We formally define the terms *parallel* and *interfering* workload of a provider θ_i^*. Note, $W \leq \sum_{\theta_i^* \in \Theta} W_i$ as a consumer can be accounted for more than once if it can execute concurrently with multiple providers.

Definition 1 (Parallel Workload of θ_i^*) The parallel workload α_i of θ_i^* is the workload in $W_i - L_i$ that can execute before the time instant $f(\theta_i^*)$.

For a node v_j in $F(\theta_i^*) \cup G(\theta_i^*)$, it contributes to α_i if either $f(v_j) \leq f(\theta_i^*)$ or $f(v_j) - C_j < f(\theta_i^*)$. The former case (i.e., $f(v_j) \leq f(\theta_i^*)$) indicates v_j is finished before the finish of θ_i^* and cannot cause any delay, whereas $f(v_j) - C_j < f(\theta_i^*)$ means v_j can partially execute in parallel with θ_i^* so that its delay on θ_{i+1}^* is less than C_j.

Definition 2 (Interfering Workload of θ_i^*) The interfering workload of θ_i^* is the workload in $W_i - L_i$ that executes after the time instant $f(\theta_i^*)$. For a provider θ_i^*, its interfering workload is $W_i - L_i - \alpha_i$.

With Definitions 1 and 2, Lemma 1 follows.

Lemma 1 *For providers θ_i^* and θ_{i+1}^*, the workload in W_i that can delay the start of θ_{i+1}^* is at most $W_i - L_i - \alpha_i$.*

Proof Based on the CPC model, the start of θ_{i+1}^* depends on the finish of both θ_i^* and $F(\theta_i^*)$, which is $\max\{f(\theta_i^*), \max_{v_j \in F(\theta_i^*)} f(v_j)\}$. By Definition 1, α_i will not cause any delay as it always finishes before $f(\theta_i^*)$, and hence, the lemma follows. Note that although $G(\theta_i^*)$ cannot delay θ_{i+1}^* directly, it can delay on nodes in $F(\theta_i^*)$, and in turn, causes an indirect delay to θ_{i+1}^*. □

3.3 DAG Scheduling: A Parallelism and Dependency Exploited Method

Based on the CPC model, a scheduling method is then presented to maximize node parallelism. This is achieved by a rule-based priority assignment, in which three rules are developed to statically assign a priority to each node in the DAG. Firstly to always execute the critical path first (Sect. 3.3.1), and then two rules (Sect. 3.3.2) to maximize parallelism and minimize the delay to the critical path.

The entire presented approach has general applicability to DAGs with any topology (unlike, e.g., [25], which assumes nested fork-join DAGs only). It assumes a homogeneous architecture, however, it is not restricted by the number of processors.

3.3.1 The "Critical Path First" Execution (CPFE)

In the CPC model, the critical path is conceptually modelled as a set of capacity providers. Arguably, each complete path can be seen as the providers, which offers the time interval of its path length for other nodes to execute in parallel. However, the critical path provides the maximum capacity and hence, enables the maximized total parallel workload (denoted as $\alpha = \sum_{\theta_i^* \in \Theta^*} \alpha_i$). This provides the foundation to minimize the interfering workload on the complete critical path.

Theorem 1 *For a schedule $\mathcal{S}$ with CPFE and a schedule $\mathcal{S}'$ that prioritizes a random complete path over the critical path, the total parallel workload of providers in S is always equal to or higher than that of S', i.e., $\alpha \geq \alpha'$.*

Proof The change from $\mathcal{S}$ to $\mathcal{S}'$ leads to two effects: (1) a reduction on the length of the provider path, and (2) an increase on length of one consumer path. Below we prove both effects cannot increase the parallel workload after the change.

First, suppose the length of provider θ_i^* is shortened by Δ after the change from $\mathcal{S}$ to $\mathcal{S}'$, the same reduction applies on its finish time, i.e., $f'(\theta_i^*) = f(\theta_i^*) - \Delta$. Because nodes in θ_i^* are shortened, the finish time $f(v_j)$ of a consumer node $v_j \in F(\theta_i^*) \cup G(\theta_i^*)$ can also be reduced by a value from Δ/m (i.e., a reduction on v_j's

interference, if all the shortened nodes in θ_i^* belong to $C(v_j)$) to Δ (if all such nodes belong to $pre(v_j)$) [28, 39]. By definition 1, a consumer $v_j \in F(\theta_i^*) \cup G(\theta_i^*)$ can contribute to the α_i if $f(v_j) \leq f(\theta_i^*)$ or $f(v_j) - C_j \leq f(\theta_i^*)$. Therefore, α_i cannot increase in $\mathcal{S}'$, as the reduction on $f(\theta_i^*)$ (i.e., Δ) is always equal or higher than that of $f(v_j)$ (i.e., Δ/m or Δ).

Second, let L and L' denote the length of the provider path under $\mathcal{S}$ and $\mathcal{S}'$ (with $L \geq L'$), respectively. The time for non-critical nodes to execute in parallel with the provider path is L' on each of $m-1$ cores under $\mathcal{S}'$. Thus, a consumer path with its length increased from L' to L directly leads to an increase of $(L - L')$ in the interfering workload, as at most L' in the consumer can execute in parallel with the provider.

Therefore, both effects cannot increase the parallel workload after the change from $\mathcal{S}$ to $\mathcal{S}'$, and hence, $\alpha \geq \alpha'$. □

Rule 1. $\forall v_j \in \Theta^*, \forall v_k \in \Theta \Rightarrow p_j > p_k$.

Theorem 1 leads to the first assignment rule that assigns critical nodes with the highest priority, in which p_j denotes the priority of node v_j. With Rule 1, the maximum parallel capacity is guaranteed so that an immediate reduction (i.e., α) on the interfering workload of λ^* can be obtained.

3.3.2 Exploiting Parallelism and Node Dependency

With CPFE, the next objective is to maximize the parallelism of non-critical nodes and reduce the delay on the completion of the critical path. Based on the CPC model, each provider θ_i^* is associated with $F(\theta_i^*)$ and $G(\theta_i^*)$. For $v_j \in G(\theta_i^*)$, it can execute before $F(\theta_i^*)$ and use the capacity of θ_i^* to execute, if assigned with a high priority. Under this case, v_j can (1) delay the finish of $F(\theta_i^*)$ and the start of θ_{i+1}^*, and (2) waste the capacity of its own provider. A similar observation is also obtained in [33], which avoids this delay by the heuristic of early interference node first.

Rule 2. $\forall \theta_i^*, \theta_l^* \in \Theta^* : i < l \Rightarrow \min_{v_j \in F(\theta_i^*)} p_j > \max_{v_k \in F(\theta_l^*)} p_k$.

Therefore, the second assignment rule is derived to specify the priority between consumer groups of each provider. For any two adjacent providers θ_i^* and θ_{i+1}^*, the priority of any consumer in $F(\theta_i^*)$ is higher than that of all consumers in $F(\theta_{i+1}^*)$. With Rule 2, the delay from $G(\theta_i^*)$ on $F(\theta_i^*)$ (and hence θ_{i+1}^*) can be minimized, because all nodes in $G(\theta_i^*)$ belong to consumers of following providers and are always assigned with a lower priority than nodes in $F(\theta_i^*)$.

We now schedule the consumer nodes in each $F(\theta_i^*)$. In [33], concurrent nodes with the same earliness (in terms of the time they become ready during the execution of the critical path) are ordered by the length of their longest complete path (i.e., from v_{src} to v_{sink}). However, based on the CPC model, a complete path can be divided into several local paths, each of these local paths belong to the consumer group of different providers. For local paths in $F(\theta_i^*)$, the order of their lengths can

Algorithm 2: $EA(\Theta^*, \Theta)$: priority assignment

Inputs : Θ^*, Θ
Parameters : p, p^{max}
Initialize : $p = p^{max}, \forall v_j \in \Theta^* \cup \Theta, p_j = -1$

```
/* Assignment Rule 1. */
∀v_j ∈ Θ*, p_j = p;   p = p − 1;
/* Assignment Rule 2. */
for each θ_i* ∈ Θ*, in topological order do
    while F(θ_i*) ≠ ∅ do
        /* Find the longest local path in F(θ_i*). */
        v_e, v_j ∈ F(θ_i*) :
            v_e = argmax_{v_e} {l_e(F(θ_i*)) | suc(v_e) = ∅};
            λ_{v_e} = v_e ∪ λ_{v_j}, argmax_{v_j} {l_j(F(θ_i*)) | ∀v_j ∈ pre(v_e)};
        if |pre(v_j)| > 1, ∃v_j ∈ λ_{v_e} then
            {Θ*', Θ'} = CPC(F(θ_i*), λ_{v_e});
            EA(Θ*', Θ');
            break;
        else
            /* Assignment Rule 3. */
            ∀v_j ∈ λ_{v_e}, p_j = p;   p = p − 1;
            F(θ_i*) = F(θ_i*) \ λ_{v_e};
        end
    end
end
```

be the exact opposite to that of their complete paths. Therefore, this approach can lead to a prolonged finish of $F(\theta_i^*)$.

In the constructed schedule, we guarantee a longer local path is always assigned with a higher priority in a dependency-aware manner. This derives the final assignment rule, as given below. Notation $l_j(F(\theta_i^*))$ denotes the length of the longest local path in $F(\theta_i^*)$ that includes v_j. This length can be computed by traversing $anc(v_j) \cup des(v_j)$ in $F(\theta_i^*)$ [33]. With Rules 1-3 applied to the example DAG, it finally leads to the best-case schedule with a makespan of 13.

Rule 3*. $v_j, v_k \in F(\theta_i^*) : l_j(F(\theta_i^*)) > l_k(F(\theta_i^*)) \Rightarrow p_j > p_k$

However, simply applying Rule 3 to each $F(\theta_i^*)$ is not sufficient. Given a complex DAG structure, every $F(\theta_i^*)$ can form a smaller DAG $\mathcal{G}'$, and hence, an inner nested CPC model with the longest path in $F(\theta_i^*)$ is the provider. Furthermore, this procedure can be recursively applied to keep constructing inner CPC models for each consumer group in a nested CPC model, until all local paths in a consumer group are fully independent. For each inner nested CPC model, Rules 1 and 2 should be applied for maximized capacity and minimized delay of each consumer group, whereas Rule 3 is only applied to independent paths in a consumer group for maximized parallelism (and hence, the star mark on Rule 3). This enables complete awareness of inter-node dependency and guarantees the longest path first in each nested CPC model.

Algorithm 2 provides the complete approach of the rule-based priority assignment. The method starts from the outer-most CPC model ($CPC(\mathcal{G}, \lambda^*)$), and assigns all provider nodes with the highest priority based on Rule 1 (Line 2). By Rule 2, the algorithm starts from the earliest $F(\theta_i^*)$ (Line 4) and finds the longest local path λ_{v_e} in $F(\theta_i^*)$ (Line 8-9). If there exists dependency between nodes in λ_{v_e} and $F(\theta_i^*)\backslash\lambda_{v_e}$ (Line 9), $F(\theta_i^*)$ is further constructed as an inner CPC model with the assignment algorithm applied recursively (Line 11-12). This resolves the detected dependency by dividing λ_{v_e} into a set of providers. Otherwise, λ_{v_e} is an independent local path so that priority is assigned to its nodes based on Rule 3. The algorithm then continues with $F(\theta_i^*)\backslash\lambda_{v_e}$. The process continues until all nodes in V are assigned with a priority.

The time complexity of Algorithm 2 is quadratic. At most, $|V| + |E|$ calls to Algorithm 1 are invoked to construct the inner CPC models (Line 11), which examines each node and edge in the DAG. Mutually exclusively, Lines 16-17 assign each node with a priority value. Given that the time complexity of Algorithm 1 is $O(|V| + |E|)$, we have the time complexity $O((|V| + |E|)^2)$ for Algorithm 2. Although Algorithm 2 is recursive, this result holds as a node assigned with a priority will be removed from further iterations (Line 17), i.e., each node (edge) is processed only once.

With the CPC model and the schedule, the complete process for scheduling a DAG consists of three phases: (i) transferring the DAG to CPC; (ii) statically assigning a priority to each node by the rule-based priority assignment, and (iii) executing the DAG by a fixed-priority scheduler. With the input DAG known a priori, phases (i) and (ii) can be performed offline so that the scheduling cost at run-time is effectively reduced to that of the traditional fixed-priority system.

4 Reliable Resource Sharing in Reliable Autonomous Driving

The contents of this section is organized as follows. Section 4.1 describes the system and task model assumed in this section. Section 4.2 presents a fault-tolerance solution for MCS with shared resources, which includes a system execution model and a protocol MSRP-FT for faults which occur during critical sections.

4.1 System and Task Model

This section consider a fully partitioned system containing z identical cores (m_1 to m_z) and a set of sporadic tasks (Γ) that are scheduled by the Fixed Priority Preemptive Scheduling (FPPS) scheme. For generality, the system has tasks with $\mathcal{N}$ criticality levels which are defined by the system engineer according to their importance, denoted as $\mathcal{L} \in \{A, B, \ldots, \mathcal{N}\}$ in which A is the lowest criticality and $\mathcal{N}$ is the highest. Tasks being allocated to higher criticality levels implies a severe

consequence for overall system performance if their execution in some way fails. Each task τ_i is defined by a 6-tuple $\{T_i, D_i, pri_i, m_i, l_i, \overrightarrow{C_i}\}$, including its minimum release period T_i, constrained deadline D_i (with $D_i \leq T_i$), priority pri_i, designated core m_i, criticality $l_i \in \mathcal{L}$, and a set of Worst-Case Execution Times (WCET) $\overrightarrow{C_i} = \{C_{i,A}, C_{i,B}, \ldots, C_{i,\mathcal{N}}\}$ without accessing shared resources. The verification is more conservative for a higher criticality level [6], hence $C_{i,A} \leq C_{i,B} \leq \ldots \leq C_{i,\mathcal{N}}$. The task τ_i with criticality l_i can execute up to C_{i,l_i} from its $\overrightarrow{C_i}$.

Within the system, there also exists a set of resources $\mathcal{R}$, each of which may be accessed by all tasks in the system in a mutually exclusive fashion by executing the *critical section* associated with the resource. Each shared resource r^x is defined by two notations: $\overrightarrow{c_i^x}$ and N_i^x, in which $\overrightarrow{c_i^x} = \{c_{i,A}^x, c_{i,B}^x, \ldots, c_{i,\mathcal{N}}^x\}$ denotes the set of worst-case computation time τ_i needed to execute r^k with different levels of criticality, and N_i^x gives the number of requests from τ_i in one release. In this section the execution budgets of different segments of the same task (e.g. $C_{i,A}$ and $c_{i,A}^x$) increase or decrease simultaneously with the transition of system modes (see Sect. 4.2). However, to ease the presentation, the notation c^x is used to denote the worst-case time for executing r^x by all requesting tasks with any criticality level. Nested resource sharing is not considered in this section, i.e., a task can only hold one resource at a time, but can be directly supported by group locks [62].

Transient faults which can be resolved by redundancy approaches (e.g. re-execution and replication) in this section. Each fault can only affect one task at a time and the acceptance test is applied as the fault-detection technique.

4.2 A Fault-Tolerant Solution for MCS with Shared Resources

In this section, we present a new fault-tolerant solution for generic MCS that have two or more criticality levels with shared resources, to handle both task overruns and transient faults. First, we introduce a new fault-aware system model for MCS. The system model distinguishes faults occurring in normal and critical sections, which enables different fault-tolerance schemes to be implemented. Then, based on MSRP, a novel fault-tolerance multiprocessor resource sharing protocol is presented for handling faults in critical sections, which reduces the blocking time incurred for tolerating faults and guaranteeing the reliability of the system.

4.2.1 The Fault-Tolerance System Model

To handle task overruns and faults which occur during both normal and critical sections of a MCS, a fault-tolerant system model based on the extension of the AMC model [6] is introduced. Figure 4 illustrates the execution flow of the system and tasks in the model.

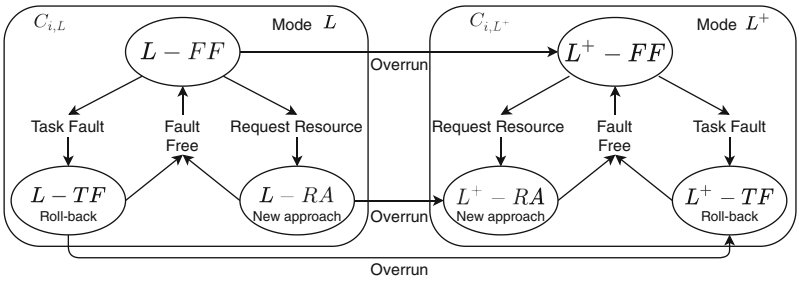


Fig. 4 The fault-tolerance system model

During a task's execution, faults can occur either in a normal or a critical section. The former is called a *task fault* and the latter a *resource fault* in this section. In the presented model, different fault-tolerant techniques are adopted to tolerate these two types of fault. The fault detection and tolerance techniques for normal and critical sections are presented in Sect. 4.2.2 and 4.2.3.

As shown in Fig. 4, each task has three execution states under a system mode (say L): fault-free (*L-FF*), task-fault (*L-TF*) and resource-access (*L-RA*). They are allowed to execute up to an execution budget $C_{i,L}$. A task executing in state *L-FF* is executing a normal section without incurring any faults. Once a fault occurs in a normal section, the task moves to state *L-TF*, at which the fault will be resolved. If a task requests a resource, it moves to state *L-RA* directly, where the fault-tolerance procedure for critical sections will be activated immediately, guaranteeing a fault-free resource access (see Sect. 4.2.3). The task moves back to state *L-FF* from *L-TF* or *L-RA* if the fault is resolved or the resource access is finished, respectively.

The system advances to the next system mode L^+ if any task in mode L overruns its budget. When an overrun occurs, tasks with criticality $l_i \geq L^+$ that are running in states *L-FF*, *L-TF* and *L-RA* will move directly to L^+-*FF*, L^+-*TF* and L^+-*RA* respectively with elevated execution budgets C_{i,L^+} and other tasks are dropped. By doing so, each overrun can bring the system to the next mode. However, there is an exception for tasks with criticality $l_i < L^+$ running in the state *L-RA* while executing with a shared resource, they are allowed to be dropped after finishing the underway critical section for the consideration of data integrity [31]. Moreover, mode changes can go in the reverse direction, when the system has less computation pressure it will resume suspended tasks and start in the lowest mode. Details of this will not be addressed here due to space constraints.

4.2.2 Fault-Tolerance of Normal Sections

In this section, we focus on transient faults which can be resolved by redundancy approaches. However, in systems with shared resources, detecting faults at the end of a task and re-executing the whole task to resolve a transient fault can lead to

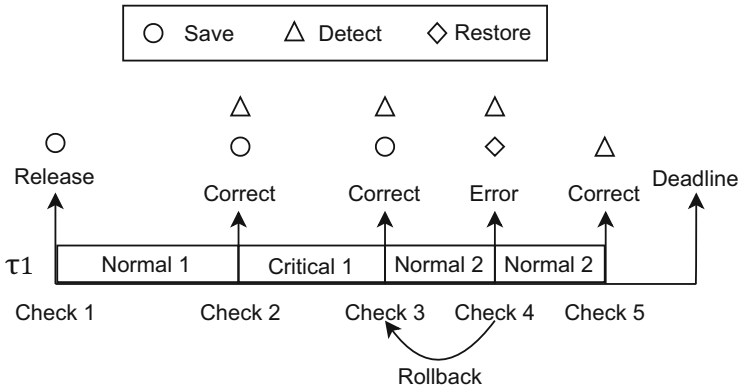


Fig. 5 Fault-tolerance in normal sections

substantial blocking time and the risk of transferring incorrect data to other tasks. To minimize the blocking time and provide reliable resource sharing, we apply different fault-tolerance approaches to handle faults that occur in normal and critical sections. This is achieved by not only inserting checkpoints at the start and end of each task but also introducing additional checkpoints around each critical section of the task. By doing so, the task execution is divided into a set of normal and critical sections. The acceptance test is assumed to be applied as the fault detection technique at each checkpoint.

In the presented fault-tolerance approach, the purposes of the checkpoints are slightly different, and so their operations vary. As shown in Fig. 5, a checkpoint (e.g. Check 1) will be set at the beginning of a task to perform a *Save* operation which involves storing the current architectural state of the system, including register files, counter values and etc. For fault-tolerance in normal sections, each checkpoint will operate a *Detect* operation to detect faults after the execution of each normal segment. If no faults are detected (e.g. at Check 2) the checkpoint will perform the *Save* operation. Otherwise, if a fault is detected (e.g. at Check 4) the task will roll back to the most recent checkpoint and perform the *Restore* operation which restores the previous data and re-performs the execution. This process repeats until the normal section is executed without any fault. Each re-attempt requires an additional *Detect* operation (e.g. at Check 5). However, for the end of the last execution segment, the *Save* operation is not needed at the checkpoint.

4.2.3 Fault-Tolerance of Critical Sections by MSRP-FT

For faults occurring in critical sections, the presented model utilizes a novel fault-tolerance multiprocessor resource sharing protocol, called MSRP-FT, in which tasks waiting for a resource can assist the resource holder to execute the associated critical section in parallel to address potential faults. The objective is to reduce

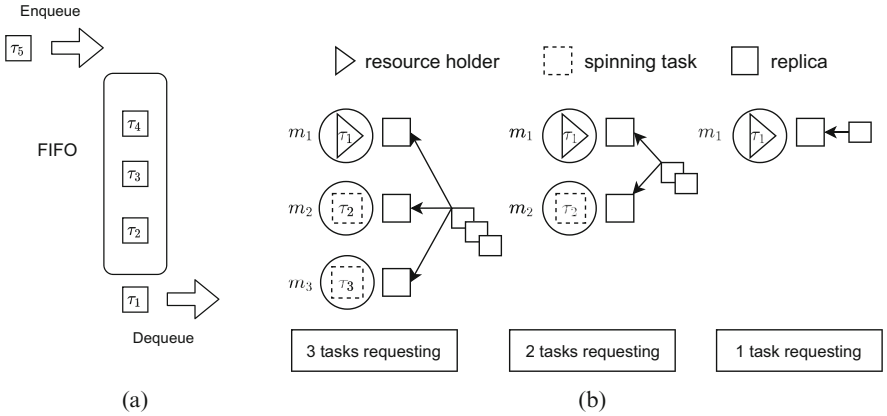


Fig. 6 Fault tolerance in critical section. (**a**) An example of a FIFO queue. (**b**) Replicas allocation based on the number of tasks in the queue

the additional blocking time caused by resolving faults in critical sections via re-executions. The mentioned MSRP-FT is introduced with the following steps.

4.2.3.1 Allocation of Replicas

Figure 6 demonstrates an example of the implementation of MSRP-FT, which is based on the resource sharing protocol MSRP. According to MSRP [27], tasks are inserted into a FIFO queue when they request a global resource. The task at the head of the queue (e.g. τ_1 in the figure) is granted the resource, other tasks spin on their own cores while checking the lock non-preemptively. With MSRP-FT, tasks are also placed at the FIFO queue when requesting shared resources. The task at the head of the FIFO queue will access the shared resource and the code segment to be executed by the head task and the internal states (e.g. variables) of the resource are replicated to a number according to the number of tasks in the FIFO queue as shown in Fig. 6b. It is worth noting that the access to the resource is always performed by the head task which obeys the mutually exclusive principle of shared resources and will not incur a race condition. Afterwards, replicas are stored in the local memory of each core and each task in the FIFO queue (including the head task) executes a replica on their host cores in parallel and updates the results on the local replica independently. If there is only one task in the FIFO queue, the head task has to execute the critical section by itself.

4.2.3.2 Submission of Replicas

Each execution of the replica is tested for faults on different cores. As shown in Fig. 7b, if a replica finishes without incurring any fault (e.g. on core m_3), it will

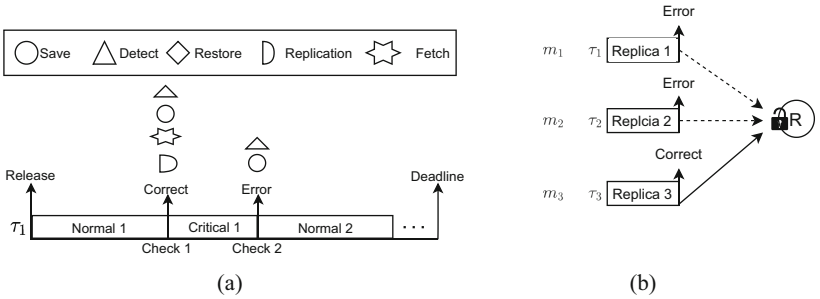


Fig. 7 Fault-tolerance in a critical section. (**a**) Operations of checkpoints around a critical section. (**b**) Submission of execution results

obtain the lock and update the shared resource with its local variables. If two overlapping requests to acquire the lock arrive, one task will commit the result and another will have no effect on the resource. The update of the resource is assumed to be conducted with an atomic action which once performed no other action can interleave with it, hence, race conditions are avoided. Once the resource is updated, other tasks are signaled to abandon the computation. In contrast, if all the resource-accessing tasks fail to obtain the correct result, they roll back and re-execute the replica until the correct result is successfully submitted. With a successful commit by any task in the FIFO queue, the head task (i.e., τ_1) is removed from the queue and continues its execution. The same procedure then repeats for the next head task within the FIFO queue.

Figure 7a shows the operations performed at the checkpoints around the critical section of τ_1. The checkpoint at the start of the critical section (e.g. Check 1) first performs *Detect* and *Save* operations to detect for faults and save the results of the execution of the previous segment, which is the same as mentioned above. It also applies *Fetch* and *Replicate* operations to fetch and replicate the corresponding operation and the shared resources to the spinning cores. A *Detect* operation is performed after the execution of the replica. Although the replica incurs faults, τ_3 already updated the result and a *Save* operation is performed to save the architectural states of the system and τ_1 continues its execution.

4.2.3.3 Working example

To clarify the implementation of the above fault-tolerance approach, the detailed execution procedure of the example stated above under two different fault-tolerance approaches is presented in Fig. 8. Figure 8a assumes that each critical section is checked for faults and any detected fault is tolerated directly by the roll-back and re-execution approach. As shown in Fig. 8a, τ_1, τ_2 and τ_3 request for a shared resource concurrently at $t = 1$. According to MSRP, τ_1 ranks first in the FIFO queue so it is granted with the resource and starts to execute its critical section immediately.

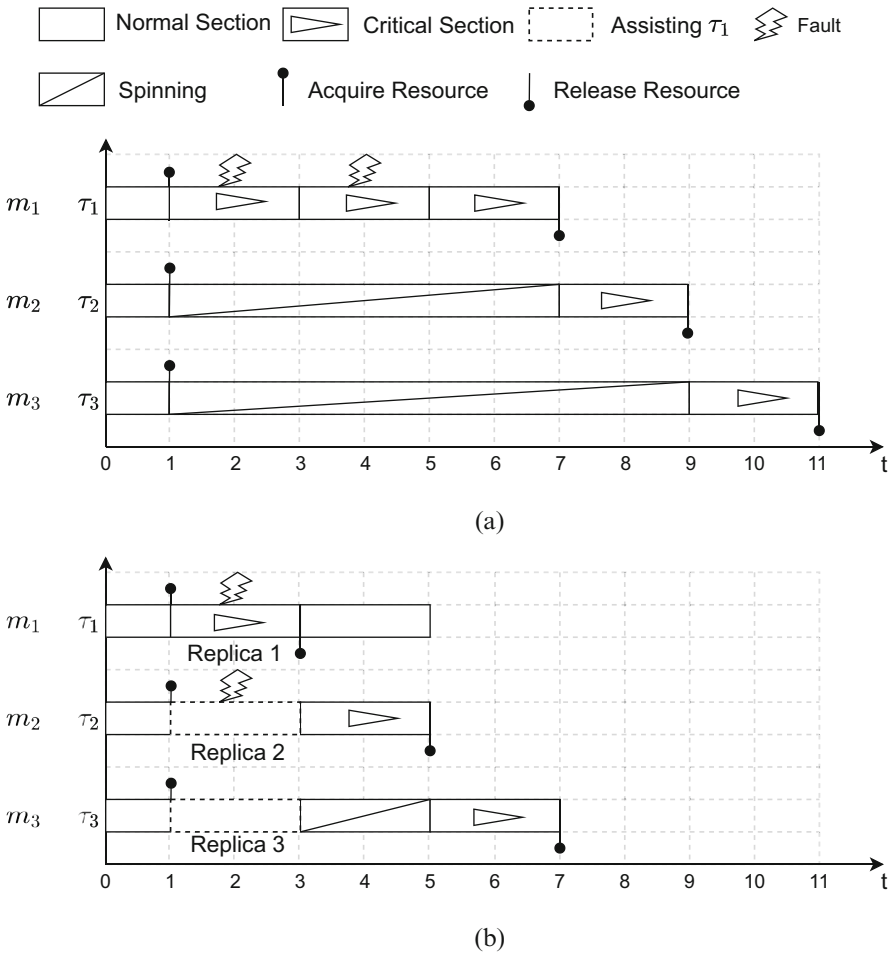


Fig. 8 A comparison between two fault-tolerance approaches under the same checkpoints setting. (**a**) Fault tolerance by simple segment re-execution. (**b**) The presented fault-tolerance method

Other tasks (τ_2 and τ_3) spin on their own cores and wait for the resource. However, τ_1 incurs two faults consecutively and re-executes its critical section twice. It finally releases the resource and leaves the FIFO queue at $t = 7$. τ_2 then becomes the head of the queue, which acquires the resource and starts its critical section from then.

With the application of the presented fault-tolerant approach, as shown in the Fig. 8b, the cores of τ_2 and τ_3 are utilized to execute $\tau_1's$ critical section in parallel instead of spinning. Although only one piece of the replica (i.e., Replica 3) is executed without faults, τ_1 can still continue its execution at $t = 3$. The chief principle of the fault-tolerant approach for critical sections is to replace wasted

cycles of the spinning tasks in the FIFO queue to provide the reliability guarantee for each critical section in a single access, in pursuance of reducing the time spent on fault-tolerance and resource contention. For local resources, each task has to execute by itself as there exists no spinning tasks on remote cores.

4.2.3.4 Implementation and Run-Time Overhead

The implementation of the above approach requires the hardware architecture to have individual cache memory or dedicated memory space for each core to store replicas during the execution of the MSRP-FT, where most commercially off-the-shelf (COTS) architectures can satisfy. From the software aspect, a global scheduler will be adopted to communicate with tasks on different cores. For example, the scheduler will signal tasks to assist the head task (i.e. the resource holder) to execute the replicas in parallel. Once a successful result is submitted, the scheduler will signal other tasks to abandon the execution on replicas. Threads control methods such as *wait()* and *notify()* can be used to construct the above communication logic.

The feasibility of a task executing operations on behalf of other tasks has been validated in [51], in which once a task is preempted while spinning in the FIFO queue, the task behind it can acquire the lock first and execute the operation on behalf of the preempted task. Burns and Wellings [14] also briefly describes how the associated computations of the preempted task holder can be executed by the spinning tasks in parallel on different cores, but a detailed system design and implementation execution framework are not provided. Although the presented fault-tolerance approach is developed within a different context and serves a different purpose, that of reducing blocking time caused by resource faults, the above work has provided sufficient evidence towards the applicability and practicability of the presented approach.

Moreover, the setting of checkpoints can bring additional overheads in terms of execution time. However, there is a clear trade-off between the number of checkpoints being set and the final schedulability benefits of the presented approach. If the task has intensive resource requests (i.e. contains voluminous critical sections), the engineer can set fewer checkpoints in a flexible manner so that a balanced result can still be achieved between the time spent for each checkpoint and the advantage brought by the approach presented.

Finally, the presented fault-tolerance method can also be applied to other FIFO spin-based resource sharing protocol, e.g. MrsP [14]. The choice of MSRP made in this chapter is due to its non-preemptive spinning feature, which provides a strong guarantee to the resource-accessing and helping process. Under MSRP, spinning tasks are prevented from being preempted while assisting the resource holder, and hence avoids prolonging the helping process as well as over-complicated execution scenarios.

5 Real-Time TSN Scheduling for Automotive CPS

In this section, we present the frame-level FPS method for TSN scheduling and analysis. We present an overview in Sect. 5.1. Followed by scheduling of TSN with FPS in Sect. 5.2 and deferred queue in Sect. 5.3. A corresponding schedulability analysis is given in Sect. 5.4. Finally, the network and control co-design is formulated in Sect. 5.5 by period and control poles assignment.

5.1 Overview of Traffic Scheduling of TSN

In this section, we present an integrated solution that solves the controller-network co-design problem. Scheduling on a single TSN switch is considered and can be extended to the entire network. As we focus on the scheduling aspect, it is assumed the network communication is ideal: (i) the depth of the queues is sufficient, i.e., no traffic overflows; (ii) the channel is error-free and has a constant transmission rate. These ease the analysis and helps to understand the nature of the problem. Relaxing them in practice needs limited modifications and will be discussed in the future. The network is subjected to two basic traffic types: scheduled and unscheduled traffic, depending on a certain level of quality-of-service (QoS) is required or not. In this section, we focus on scheduled traffic and leave unscheduled traffic be transmitted using residual bandwidth with best effort.

TSN provides time synchronization and time-division transmission, which enables global scheduling through GCLs [63]. Although the schedule of TSN can be designed by hand, it soon becomes impractical as the network turns complex and more packets are added to the network. In this section, we specify the scheduling policy adopted for TSN while control systems are considered. The presented schedule minimizes the blocking of packets (including ones sent by control tasks), to improve schedulability and control performance. We then introduce a fine-grained response time analysis that bounds the worst-case latency of packets in a single Qbv switch. Below we first discuss the system model.

System Model The system contains N periodic packets[3] $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_N\}$, including both control (Γ_c) and non-control packets (Γ_{nc}) sent by tasks from the application. Each packet τ_i is modelled as a 7-tuple $\{L_i, C_i, T_i, D_i, P_i, R_i, \Lambda_i\}$, representing the worst-case length of the packet L_i, transmission time C_i, period T_i, deadline D_i, priority P_i, worst-case latency R_i and the set of frames Λ_i in each release, respectively. Frames are transmitted in a non-preemptive fashion. A global

[3] Continuously released periodic packets will form a flow. For simplicity, we use these two terms interchangeably.

packet transmission rate v is applied to all packets, thus $C_i = L_i/v$ for τ_i. Each control packet is assigned with an implicit deadline i.e., $D_i = T_i$. To provide a more general network model for the system, the non-control packets can have arbitrary deadlines without any constraint imposed. As a consequence, at a given time instant there could be several instances of a non-control packet waiting for transmission in the switch. The priorities of all packets are assigned according to the deadline monotonic algorithm ($P_i > P_j$ if $D_i < D_j$), and each packet has a unique priority. In addition, the Maximum Transmission Unit (MTU) is considered, denoted as M, which defines the maximum data size allowed in a single transmission. For the ease of presentation, we denote M as the transmission time for sending data with a size equal to one MTU. Thus, each packet could be divided into a set of successive frames, i.e., $\Lambda_i = \{\lambda_i^1, \lambda_i^2, \ldots, \lambda_i^m\}$, with $m = \lceil L_i/M \rceil$. For a given frame λ_i^j, it inherits the analytical properties of τ_i (i.e., T_i, D_i and P_i), and has its own data length, L_i^j, and transmission time, C_i^j.

5.2 Scheduling Network Packets in TSN

In a typical Qbv switch, the network packets are queued by their arriving time (i.e., FIFO queuing) and are transmitted non-preemptively [35]. Traditionally, the synthesis of GCL schedule is performed using Satisfiability Modulo Theories (SMT) [20, 41] or Integer Linear Programming (ILP) [5]. The defined end-to-end latency imposes zero-jitter, however, with significantly reduced solution space. The scheduling in TSN networks with Quality-of-Service (QoS) requirements can be either performed at the queue level [63] or packet level [43]. With the queue-level scheduling, each FIFO queue in the Qbv switch is assigned with a priority, and packets in a queue with a higher priority are always transmitted first. However, as packets in each queue are transmitted strictly in a FIFO order, packets under the queue-level scheduling approach can incur substantial blocking, where packets with a tighter deadline but at the end of a queue cannot be favored. That is, with the queue-level scheduling, packets with different deadlines in the same FIFO queue are treated equally without concerning individual temporal requirements. For control systems, such a scheduling is not appropriate, as the delay for transmitting control packets can introduce significant impact on the control performance of the system. Thus, the packet-level (more precisely, the frame-level) scheduling is adopted to provide a finer-grained schedule, where each packet (and its frames) is scheduled strictly by its priority.

However, even with the packet-level scheduling, packets can still incur additional delay due to the FIFO queuing, as the actual transmission largely depends on the arriving time of the packets. In the worst-case, a late-arrived packet with a high priority can be blocked by all the released packets with lower priorities. To minimize the delay due to FIFO queuing, an alternative is to perform the scheduling off-line

(i.e., prior to execution), with the complete knowledge of all packets in the system.[4] The offline scheduling can be performed by assuming all packets are arrived at the same time, with a packets transmission order obtained based on their priorities. If packets have different arrival times during run-time, a simple mechanism that defers the queuing of the early-arrived low-priority frames can be adopted, to maintain the queuing order obtained from the offline FPS-NP without imposing extra latency to packet transmission (see Sect. 5.3 for deferred queuing). By maintaining the offline packets transmission order during run-time, the blocking time of each packet during transmission can be minimized to one frame only, i.e., identical to the classic non-preemptive fixed-priority scheduling (FPS-NP) [23].

Based on the above discussion, to provide a fine-grained schedule and to minimize the delay due to the queuing problems, the scheduling adopted in this section is conducted before runtime on the frames of each packet in one hyper-period, with the scheduling decisions encoded into the GCL. Once a schedule is obtained, the frames can be statically allocated to the FIFO queues according to the schedule while the scheduling decisions can be mapped to the GCL to control the gates of all queues to achieve the desired execution order. To this end, the scheduling on TSN can be successfully mapped to the traditional FPS-NP, in which each packet is scheduled strictly by its priority and can be blocked maximum once during the entire transmission.

With the described scheduling approach, we avoid the packets queuing problem and can achieve the minimized delay for all packets, in the context of a Qbv switch. This is crucial for control systems as the resulting control performance can be affected by transmission delay for the control packets. To our best knowledge, this is one of the earliest work targeting at control systems in which the timeliness and performance are sensitive to the transmission delay of certain critical (i.e., control and non-control) packets. For the non-control packets, meeting their timing requirements is essential for guaranteeing the system correctness, whereas minimizing transmission delay of the time-triggered control packets are essential crucial for control performance.

For unscheduled packet flows that do not have a temporal requirements, the traffics can be scheduled using residual bandwidth left by the critical traffics with time-aware shapers [52, 53] and queue partitioning. Supporting such flows has been well-described by the above work, and will not be re-presented in this section. Targeting at such systems, a complete scheduling solution is presented that minimizes the transmission delay for all packets, in the context of the TSN Qbv switch. Last but not least, different from [20], our approach makes no assumption on the isolation of incoming packets and the construction of the GCL, e.g., isolating certain queues for a specific packet type, to provide a more general approach for using TSN in control systems.

[4] Such an approach is feasible as the packets are deterministic i.e., the packets sent by each task are known a prior with periodic release.

5.3 Deferred Queue

As described in Sect. 5.2, for packets with different arriving times, a mechanism is required to delay the queuing of the early-arrived low priority packets so that the minimized blocking can be guaranteed. To achieve this, a *deferred queue* with priority ordering is introduced into the Qbv switch, which is integrated into the packet filtering unit (see Fig. 3) for holding early-arrived packets temporarily, until they can be added into the scheduled queues with a correct order.

Assuming simultaneous release for all packets at the start of the system, the offline FPS-NP schedule can produce a well-planned transmission order for all packet instances released in one hyperperiod, in which each packet (a set of successive frames) is scheduled strictly based on priority. For this schedule, the blocking of each packet is minimized, as in the worst case, the ready packet with the highest priority can start transmitting after the currently transmitting frame of a low priority packet has completed. During run-time, this offline scheduling order is encoded into the priority filtering unit, which provides a reference of the expect order for incoming packets.

For each incoming packets, the priority filter examines whether this packets arrives by the expected order, i.e., all its previous packets with a higher priority have arrived. If so, this packet is dispatched to the scheduled queues immediately, at which it will be select to transmit by GCL. Otherwise (i.e., certain previous high priority packets haven't arrived yet), this packet is hold by the priority filter until (a) the missing packets arrives or (b) the scheduled queues are empty and this packet has the highest priority among all the deferred packets.

Note that the condition (b) can lead to a transmission order different from the expected one, as certain packets can be transmitted before a late-arriving higher priority packet. However, this does not introduce extra delay and can help increasing the throughput. With the deferred queuing, it is possible that all scheduled queues are empty while some packets are stored in the priority filter. Under this situation, the priority filter selects the packet at the head of the queue (i.e., with the highest priority) and send its frames into the scheduled queue for a direct transmission one by one, until a higher priority packet arrives. This guarantees that the transmission never stops as long as there exist waiting packets (either in the priority filter or the scheduled queues). In addition, for the late arriving high priority packet, its blocking is still at most C_i^j, where it can be transmitted directly after the currently-transmitting frame.

5.4 Worst-Case Response Time Analysis

With the scheduling in TSN mapped to the traditional FPS-NP, the worst-case response time for transmitting a packet in a single Qbv switch can be obtained, which bounds the time duration from when the packet enters into the switch to when

the packet is transmitted. Due to the different deadline constraints of the control and non-control packets (i.e., implicit and arbitrary deadlines respectively), different analysis techniques are applied for each packet type. However, as both control and non-control packets are scheduled strictly by the FPS-NP, the basic philosophy for analyzing both types of packets is similar to that in [23], but with modifications and improvements in order to reflect the unique features of the Qbv switch and to support the analysis at the frame level.

The response time equation of a packet τ_i is given in the following equation for both control and non-control packets:

$$R_i = \max_{\forall \lambda_i^j \in \Lambda_i} \begin{cases} R_i^j(0), & \text{if } \tau_i \in \Gamma_c \\ \max_{n=0\ldots\left\lceil \frac{t_i+J_i}{T_i} \right\rceil -1} \left(R_i^j(n)\right), & \text{if } \tau_i \in \Gamma_{nc} \end{cases} \tag{2}$$

In Eq. (2), $R_i^j(n)$ denotes the response time for transmitting the nth instance of frame λ_i^j in τ_i's busy period t_i, and J_i denotes the queuing time, i.e., the time window from when the first frame of τ_i reaches the Qbv Switch, until when the last frame is queued. $\left\lceil \frac{t_i+J_i}{T_i} \right\rceil$ gives the total number of times that a non-control packet can be sent within its busy period [23].

The analysis of a control packet is relatively straight forward, as at any given time, there can only exist one instance of a control packet in the system i.e., implicit deadlines. Thus, the worst-case response time of a control packet can be safely bounded by computing the maximum response time of all its frames.[5] However, for a non-control packet, multiple instances of each of its frames can co-exist due to the arbitrary deadline. Thus, the response time of a frame (with an arbitrary deadline) must be obtained by computing the maximum response time of all its instances within the busy period t_i.

Similar to [23], the busy period of a non-control packet is computed by Eq. (3), where B_i gives the worst-case blocking that τ_i can experience due to transmitting a low priority frame and $hep(i)$ refers to all indices of packets that have equal or higher priorities than P_i, including i. The recursive calculation can starts with $t_i = B_i + C_i$, and is guaranteed to converge [23], given that the total utilization for packets in $hep(i)$ is less than 1, i.e., $\sum_{\forall j \in hep(i)} (C_j/T_j) \leq 1$. We later decompose B_i in Eq. (6).

$$t_i = B_i + \sum_{\forall k \in hep(i)} \left\lceil \frac{t_i + J_k}{T_k} \right\rceil C_k \tag{3}$$

[5] From Eq. (2), the response time of a packet equals to the response time of its last frame in each transmission, which takes into account the delay for transmitting the previous frames in one transmission.

The response time of a frame is bounded by Eq. (4), in which J_i^j denotes the time to en-queue frame λ_i^j, W_i^j gives the maximum queuing delay that λ_i^j can incur in a FIFO queue before it is selected to be transmitted and C_i^j denotes its transmission time. The time for queuing λ_i^j into a FIFO queue also contains the enqueue time of frames of τ_i that are prior to λ_i^j in one transmission. In addition, for the non-control frames, $n \cdot T_i$ is subtracted as this is the arrival time of its nth instance, relative to the start of the busy period. Note, for control frames, n is always 0.

$$R_i^j(n) = \sum_{q \in [1,j]} J_i^q + W_i^j(n) + C_i^j - n \cdot T_i \tag{4}$$

Equation (5) gives the queuing delay W_i^j of frame λ_i^j, where $hp(i)$ returns a set of packets with a priority strictly higher than P_i. This equation is also applicable to either control or non-control frames, with $n = 0$ for all control frames. Figure 9 provides an example illustrating the worst-case delay of the third ($n = 2$) instance of the second frame (i.e., $j = 2$) in packet τ_i. As shown in the figure, in the worst case, the frame (in bold) has to wait for five types of other frames to transmit before it can start, which are mapped to four types of delay, as follows. In the worst case, a frame can incur four sources of delay when waiting in a FIFO queue: (i) the blocking caused by a low-priority frame that is currently transmitting i.e., B_i; (ii) the delay by τ_i's frames prior to λ_i^j (with potential existence of multiple instances); (iii) the delay by previous instances of λ_i^j and the frames after λ_i^j in each τ_i's instance sent before λ_i^j; and (iv) the interference from the frames of each packet with a higher priority than P_i. Note that (iii) accounts for the delay cause by both the previous instances of λ_i^j itself and the frames after λ_i^j in previous instances. These delays are captured by the equation respectively.

$$\begin{aligned} W_i^j(n) = B_i + (n+1) \cdot \sum_{q \in [1, j-1]} C_i^q + n \cdot \sum_{q \in [j, |\Lambda_i|]} C_i^q \\ + \sum_{\forall \lambda_k^q \in \Lambda_k, \forall k \in hp(i)} \left\lceil \frac{W_i^j(n) + J_k^q}{T_k} \right\rceil C_k^q \end{aligned} \tag{5}$$

Finally, B_i is given by Eq. (6), where $lp(i)$ returns the packets with a priority lower than P_i. The maximum blocking time that τ_i (and any of its frames) can incur is the longest transmission time among the frames of all the lower priority packets.

$$B_i = \max_{\forall \lambda_k^q \in \Lambda_k, \forall k \in lp(i)} (C_k^q) \tag{6}$$

Equations (2)–(6) summarises the response time analysis for bounding the worst-case transmission latency (i.e., the response time) of packets in a Qbv

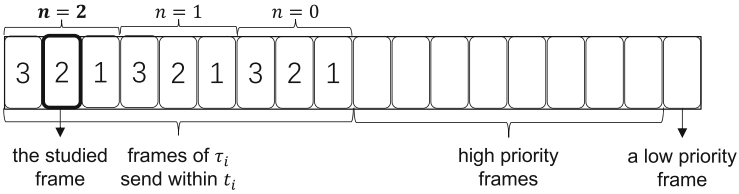


Fig. 9 The worst-case delay of a frame, which is caused by a low priority frame, high priority packets, instances of τ_i's frames prior to λ_i^j, previous instances of λ_i^j and previous instance of τ_i's frames after λ_i^j

switch for time-critical control systems. The analysis considers both implicit and arbitrary deadlines for different packet types and is fine-grained, which provides the worst-case transmission latency of each frame. Arguably, by intuition, a trivial modification that treats each frame as an independent task can be applied in an existing packet-level analysis (e.g., the one in [24]), to support the analysis at the frame-level. However, additional techniques are still required to guarantee the correct transmission order between frames that belong to the same packet and instance so that the transmission time of each individual frame can be obtained. This is achieved in our analysis by Eq. (5), which carefully examines the transmission order of different types of frames (including the ones in τ_i) and provides a tighter upper bound compared to a packet-level analysis.

The presented analysis and scheduling techniques for a single switch can be extended to support the network topology level with multiple switches and end-nodes. For the presented method, it can be implemented in each switch. For a given switch, the presented schedule takes all packets that will go through this switch and then produced a static schedule. In addition, the deferred queue is applied in each switch to handle the case in which low priority packets arrive earlier than expected. To compute the end-to-end worst-case transmission time of a packet τ_i that travels through more than one switches, the input packets of each of the switches should be given and the worst-case delay of τ_i in each switch can be effectively upper bounded by summing the worst-case delay it can incur in each switch by the above analysis.

However, with only one switch, the worst-case delay of a packet can be bounded by considering all the input packets with a synchronous release at the begin of the system. This assumption, however, may not hold in the scenario of multiple switches, in which the actual arrival time of a packet at a given switch depends on the delay it incurs at the previous switches. Thus, the analysing approach above would contain certain degree of pessimism as not all the input packets in a switch will cause a delay on τ_i, depending on their arrival times.

5.5 Controller Synthesis and Period Allocation

For a safety-critical autonomous system, for example, a self-driving car, the control functions are crucial and should always be a major concern. Further to the introduced scheduling and analysis that guarantee the timing of control packets, a well-designed controller is also required, in order to satisfy the control performance requirement and even maximize it under the schedulability constraint of the network.

Most real-time controllers targeting settling time (which will be formally defined later in this section) can run at different frequencies [3, 21, 22]. In the TSN context, this rate is bounded by (i) the maximum transmission capability; (ii) the lowest control performance requirement. Hence, there exists an optimized operational point that would produce acceptable network schedulability with maximized control performance.

5.5.1 Control Model

For a linear-time-invariant (LTI) controlled plant, its system dynamics can be described using the following differential equations:

$$\dot{x}(t) = Ax(t) + Bu(t),\ y(t) \quad = Hx(t) \tag{7}$$

in which A, B and H are system matrices that represent the system physical properties; $x(t)$ is the system state(s); $y(t)$ is the system output(s) and $u(t)$ is the control input(s). Assuming the sampling time is T_s and the sensor-to-actuator delay is within one sampling period, at discrete time instant k, the system dynamics evolve with the following equations:

$$x(k+1) = A_d x(k) + B_d u(k-1),\ y(k) \quad = Hx(k) \tag{8}$$

where $u(-1) = 0$ for $k = 0$ and

$$A_d = e^{A \cdot T_s},\ B_d = \int_0^{T_s} e^{A\tau} d\tau \cdot B \tag{9}$$

To further simplify the equation, define an augmented variable z as: $z(k) = \begin{bmatrix} x(k) & u(k-1) \end{bmatrix}^T$, and substitute $x(k)$, $u(k)$ with $z(k)$ in Eq. (8):

$$z(k+1) = \begin{bmatrix} A_d & B_d \\ 0 & 0 \end{bmatrix} z(k) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(k) \tag{10}$$

Assuming a full state-feedback controller is used, the control input $u(k)$ is calculated by:

$$u(k) = -Kz(k) + Fr(k) \tag{11}$$

where K is the feedback gain, F is the feedforward gain and $r(k)$ is the reference. By combining Eqs. (10) and (11), the system equation therefore becomes:

$$z(k+1) = \underbrace{(A_d - B_d K)}_{A_{cl}} z(k) + B_d F r(k) \tag{12}$$

To satisfy control stability, all the eigenvalues of the closed loop dynamic matrix, i.e. A_{cl} in Eq. (12), have to be inside the unit circle. The exact value of A_d and B_d is dependent on the sampling period T_s as seen from Eq. (9), which is equal to the period of the control packet, T_i. This control model will be used through the rest of this section.

5.5.2 Problem Definition

We use settling time (t_s) as the index of quality-of-control (QoC), which is widely used in control engineering as a compulsory design requirement [16]. Settling time is defined as the time duration from when a control system is subjected to a disturbance to when it enters steady-state, i.e., the current output has reached and stays within 5% deviation of the targeted output. There is an upper bound requirement on the settling time, e.g., the settling time of a control system should not be longer than 0.5 seconds.

Finding an optimal period is crucial for (i) guaranteeing the performance of the controller itself; and (ii) ensuring enough residual time slots for non-control-related packets so they can also meet their deadlines. Based on the aforementioned objectives and constraints, the period assignment problem can be solved as an optimization problem, which is formulated as follows:

$$\begin{aligned} &\text{minimize} && \mathcal{J} = \sum_{\mathbb{D}} w_j \cdot t^*_{s,j} \\ &\text{subject to} && R_i \le D_i, t_{s,j} \le t^+_{s,j} |u_j(k)| \le u_{max}, T_i = n \cdot t_{gcd}, n \in N^+ \\ &\text{where} && i \in \Gamma, \quad j \in \Gamma_c \end{aligned} \tag{13}$$

where $w_j \in (0, 1]$ is the weight (i.e., relative importance) of the corresponding control task and $\sum w_j = 1$; $t^*_{s,j} \in [0, 1]$ is the normalized settling time of the jth controller; $\mathbb{D}$ represents the solution space of all poles that can ensure control stability; $t_{s,j}$ is the settling time of the jth controller, and $t^+_{s,j}$ is the maximum allowed settling time; $u_j(k)$ is input at discrete instance k, which is constrained by u_{max} as the maximum input threshold; The last constraint defines the time-granularity of a feasible period. To benefit from harmonic periods and to reduce the size of the GCL table, each T_i must be an integer multiple of t_{gcd}, the greatest

Algorithm 3: Periods and control poles assignment

Input: $\Gamma = \{\Gamma_c, \Gamma_{nc}\}$
Output: schedulability, S^*
Initialise: feasible and best solutions: $\mathbb{S}^f = \emptyset$, $S^* = \emptyset$
`/* construct candidate solutions: */`
formulate the solution space: $\mathbb{S} = \{S_1, S_2, \ldots, S_n\}$.
`/* explore each candidate: */`
for S_k *in* $\mathbb{S}$ **do**
  if *RTA_schedulability*(Γ^k) *is True* **then**
    for j *in* Γ_c^k **do**
      $\{t_{s,j}, u_j\}$ = pso_find_control_parameters(T_j)
    end
    if $\forall j$ *in* Γ_c^k: $t_{s,j} \leq t_{s,j}^+$ *and* $|u_j| \leq u_{max}$ **then**
      $\mathcal{J}_k = \sum w_j \cdot t_{s,j}$
      $S_k \rightarrow \mathbb{S}^f$
    end
  end
end
`/* find the best candidate solution: */`
for S_k *in* $\mathbb{S}^f$ **do**
  if $\mathcal{J}_k < \mathcal{J}^*$ **then**
    $S^* = S_k$
  end
end
`/* return feasibility: */`
if S^* *is not* $\emptyset$ **then**
  return (*feasible*, S^*)
else
  return (*infeasible*, $\emptyset$)
end

common divisor of all the packet periods. This is in accordance with common practice.

5.5.3 Solving the Network and Control Co-Design Problem

In a typical control application, while the periods of non-control-related packets are inflexible, the control-related packets often have adjustable periods. This additional flexibility allows fine tuning of controller periods to achieve the best overall performance (defined as in Eq. (13)). To solve the defined problem, a controller's period and its corresponding parameters under that period both have to be decided. These two steps are dependent on each other but can be decomposed into two sub-problems, i.e., the optimization process needs to (i) find the feasible periods that can satisfy schedulability constraints; (ii) find the controller parameters under the feasible periods that would satisfy control stability and minimal performance

requirement, and on top of that, maximize the control performance as much as possible.

For the first problem, due to the existence of harmonic periods and that the number of control tasks is often small, the search space is manageable and thus can be solved through exhaustive search. For larger scale problems, heuristic methods can be used instead to find the feasible period configurations.

For the second problem, as pole placement for the minimum settling time under input constraints is a non-convex and non-linear problem, the solution space cannot be searched easily. We use Particle Swarm Optimization (PSO) to find the optimal controller parameters (by pole placement [16]) under certain sampling period that can minimize the settling time, while given the control performance and input saturation as constraints. PSO is a population-based optimization approach for iterative improvement of candidate solutions given a non-linear non-convex objective function and a metric of quality [49].

The optimization process is given in Algorithm 3. The solution space is first formulated in Line 4. The schedulability is then tested (Line 6) to obtain potential period configurations, and under each period configuration, the optimal poles of each control task can be found through PSO (Line 8). To speedup the process, the optimal poles under the feasible range of periods can be obtained in advance. The identified configuration is appended into the feasible solutions provided that the minimum control performance and the input constraints are both satisfied (Line 10-13). Finally, the best candidate that has the minimum $\mathcal{J}$ is selected from all the feasible solutions (Line 16-20). No feasible solution is found if $S^* = \emptyset$, in which case the algorithm fails to find a solution that satisfies all the constraints.

6 Conclusion

This chapter introduces the state-of-the-art techniques which cover three major directions of scheduling and analyzing autonomous systems. The presented solutions range from DAG task scheduling, and reliable resource sharing, to in-vehicle TSN networking. The goal is to provide autonomous systems with high-performance hard real-time scheduling, reliable resource sharing, and deterministic networking scheduling.

For scheduling and analyzing DAG tasks in autonomous systems, a CPC model is constructed to capture the two key factors of a DAG structure: dependency and parallelism. Then, a rule-based scheduling method is presented which maximizes node parallelism to improve the schedulability of single DAG tasks.

To provide reliable resource sharing in multiprocessor mixed-criticality systems, this chapter describes a fault-tolerance solution for multiprocessor MCS with shared resources. The presented system execution model and fault-tolerance resource sharing protocol reduces the blocking time imposed by guaranteeing reliable resource sharing.

To provide hard real-time guarantee for network, we introduced a network scheduling model using non-preemptive fixed-priority scheduling (FPS-NP) and the mapping of the schedule into the TSN gate control list. The schedulability of the network is discussed using non-preemptive response-time analysis with the consideration of multi frames and unconstrained deadlines. An optimization method is also proposed that could find the feasible solution with maximized overall quality of control constrained by network schedulability.

References

1. Al-bayati, Z., Caplan, J., Meyer, B.H., Zeng, H.: A four-mode model for efficient fault-tolerant mixed-criticality systems. In: IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE) (2016)
2. Alfranseder, M., Deubzer, M., Justus, B., Mottok, J., Siemers, C.: An efficient spin-lock based multi-core resource sharing protocol. In: IEEE International Performance Computing and Communications Conference (IPCCC) (2014)
3. Arzén, K.-E., Cervin, A., Eker, J., Sha, L.: An introduction to control and scheduling co-design. In: Proceedings of the 39th IEEE Conference on Decision and Control, vol. 5, pp. 4865–4870. IEEE, Piscataway (2000)
4. Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A., Peri, M., Pezzini, S.: Fault-tolerant platforms for automotive safety-critical applications. In: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 170–177 (2003)
5. Bansal, B.: Divide-and-conquer scheduling for time-sensitive networks. Master's Thesis, University of Stuttgart (2018)
6. Baruah, S.K., Burns, A., Davis, R.I.: Response-time analysis for mixed criticality systems. In: IEEE Real-Time Systems Symposium (RTSS) (2011)
7. Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., Wiese, A.: A generalized parallel task model for recurrent real-time processes. In: Real-Time Systems Symposium, pp. 63–72 (2012)
8. Becker, M., Dasari, D., Mubeen, Behnam, S.M., Nolte, T.: Synthesizing job-level dependencies for automotive multi-rate effect chains. In: International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 159–169 (2016)
9. Bhuiyan, A., Sruti, S., Guo, Z., Yang, K.: Precise scheduling of mixed-criticality tasks by varying processor speed. In: Proceedings of the 27th International Conference on Real-Time Networks and Systems, pp. 123–132 (2019)
10. Bhuiyan, A., Yang, K., Arefin, S., Saifullah, A., Guan, N., Guo, Z.: Mixed-criticality real-time scheduling of gang task systems. Real-Time Syst. **57**(3), 268–301 (2021)
11. Brandenburg, B.B.: Multiprocessor real-time locking protocols: a systematic review (2019). arXiv:1909.09600
12. Burns, A.: The application of the original priority ceiling protocol to mixed criticality systems. In: Proceedings of ReTiMiCS, RTCSA (2013)
13. Burns, A., Baruah, S.: Towards a more practical model for mixed criticality systems. In: Workshop on Mixed-Criticality Systems (colocated with RTSS) (2013)
14. Burns, A., Wellings, A.J.: A schedulability compatible multiprocessor resource sharing protocol–mrsp. In: IEEE Euromicro Conference on Real-Time Systems (ECRTS). IEEE, Piscataway (2013)
15. Buttazzo, G., Cervin, A.: Comparative assessment and evaluation of jitter control methods. In: Conference on Real-Time and Network Systems, pp. 163–172 (2007)

16. Chang, W., Chakraborty, S.: Resource-aware automotive control systems design: a cyber-physical systems approach. Found. Trends Electron. Des. Autom. **10**(4), 249–369 (2016)
17. Chang, S., Zhao, X., Liu, Z., Deng, Q.: Real-time scheduling and analysis of parallel tasks on heterogeneous multi-cores. J. Syst. Architect. **105**, 101704 (2020)
18. Chen, P., Liu, W., Jiang, X., He, Q., Guan, N.: Timing-anomaly free dynamic scheduling of conditional DAG tasks on multi-core systems. ACM Trans. Embed. Comput. Syst. **18**(5), 1–19 (2019)
19. Chen, G., Guan, N., Huang, K., Yi, W.: Fault-tolerant real-time tasks scheduling with dynamic fault handling. J. Syst. Architect. **102**, 101688 (2020)
20. Craciunas, S.S., Oliver, R.S., Chmelík, M., Steiner, W.: Scheduling real-time communication in IEEE 802.1 Qbv time sensitive networks. In: Proceedings of the 24th International Conference on Real-Time Networks and Systems, pp. 183–192. ACM, New York (2016)
21. Dai, X., Burns, A.: Period adaptation of real-time control tasks with fixed-priority scheduling in cyber-physical systems. J. Syst. Architect. **103**, 101691 (2020)
22. Dai, X., Chang, W., Zhao, S., Burns, A.: A dual-mode strategy for performance-maximisation and resource-efficient cps design. ACM Trans. Embed. Comput. Syst. **18**(5s), 85 (2019)
23. Davis, R.I., Kollmann, S., Pollex, V., Slomka, F.: Controller area network (CAN) schedulability analysis with FIFO queues. In 2011 23rd Euromicro Conference on Real-Time Systems, pp. 45–56. IEEE, Piscataway (2011)
24. Davis, R.I., Kollmann, S., Pollex, V., Slomka, F.: Schedulability analysis for Controller Area Network (CAN) with FIFO queues priority queues and gateways. Real-Time Syst. **49**(1), 73–116 (2013)
25. Fonseca, J., Nelissen, G., Nélis, V.: Improved response time analysis of sporadic DAG tasks for global FP scheduling. In: International Conference on Real-Time Networks and Systems, pp. 28–37 (2017)
26. Forget, J., Boniol, F., Grolleau, E., Lesens, D., Pagetti, C.: Scheduling dependent periodic tasks without synchronization mechanisms. In: Real-Time and Embedded Technology and Applications Symposium, pp. 301–310 (2010)
27. Gai, P., Lipari, G., Di Natale, M.: Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: IEEE Real-Time Systems Symposium (RTSS) (2001)
28. Graham, R.L.: Bounds on multiprocessing timing anomalies. J. Appl. Math. **17**(2), 416–429 (1969)
29. Guan, F., Qiao, J., Han, Y.: DAG-fluid: a real-time scheduling algorithm for DAGs. IEEE Trans. Comput. **70**, 471–482 (2020)
30. Guo, Z., Yang, K., Vaidhun, S., Arefin, S., Das, S.K., Xiong, H.: Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate. In: 2018 IEEE Real-Time Systems Symposium (RTSS), pp. 373–383. IEEE, Piscataway (2018)
31. Han, J.-J., Tao, X., Zhu, D., Yang, L.T.: Resource sharing in multicore mixed-criticality systems: utilization bound and blocking overhead. IEEE Trans. Parallel Distrib. Syst. **28**, 3626–3641 (2017)
32. Haque, M.A., Aydin, H., Zhu, D.: Real-time scheduling under fault bursts with multiple recovery strategy. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2014)
33. He, Q., Jiang, X., Guan, N., Guo, Z.: Intra-task priority assignment in real-time scheduling of DAG tasks on multi-cores. IEEE Trans. Parallel Distrib. Syst. **30**(10), 2283–2295 (2019)
34. Huang H.-M., Gill, C., Lu, C.: Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks. ACM Trans. Embed. Comput. Syst. **13**(4s), 1–25 (2014)
35. IEEE 802.1 Task Group: Standard for local and metropolitan area networks – bridges and bridged networks - amendment 25: enhancements for scheduled traffic. Standard, IEEE (2016)
36. Kehrer, S., Kleineberg, O., Heffernan, D.: A comparison of fault-tolerance concepts for IEEE 802.1 time sensitive networks (TSN). In: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), pp. 1–8. IEEE, Piscataway (2014)

37. Lin, H., Li, M.-F., Jia, C.-F., Liu, J.-N., An, H: Degree-of-node task scheduling of fine-grained parallel programs on heterogeneous systems. J. Comput. Sci. Technol. **34**(5), 1096–1108 (2019)
38. Meixner, A., Bauer, M.E., Sorin, D.: Argus: low-cost, comprehensive error detection in simple cores. In: IEEE/ACM International Symposium on Microarchitecture (MICRO) (2007)
39. Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., Buttazzo, G.C.: Response-time analysis of conditional DAG tasks in multiprocessor systems. In: Euromicro Conference on Real-Time Systems, pp. 211–221 (2015)
40. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In OSDI'08: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (2008)
41. Oliver, R.S., Craciunas, S.S., Steiner, W.: IEEE 802.1 Qbv gate control list synthesis using array theory encoding. In: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 13–24. IEEE, Piscataway (2018)
42. Pathan, R.M.: Fault-tolerant and real-time scheduling for mixed-criticality systems. Real-Time Syst. **50**, 509–547 (2014)
43. Piro, G., Grieco, L.A., Boggia, G., Fortuna, R., Camarda, P.: Two-level downlink scheduling for real-time multimedia services in LTE networks. IEEE Trans. Multimedia **13**(5), 1052–1065 (2011)
44. Punnekkat, S., Burns, A., Davis, R.I.: Analysis of checkpointing for real-time systems. Real-Time Syst. **20**, 83–102 (2001)
45. Safari, S., Ansari, M., Ershadi, G., Hessabi, S.: On the scheduling of energy-aware fault-tolerant mixed-criticality multicore systems with service guarantee exploration. IEEE Trans. Parallel Distrib. Syst. **30**, 2338–2354 (2019)
46. Saidi, S.E., Pernet, N., Sorel, Y.: Automatic parallelization of multi-rate fmi-based co-simulation on multi-core. In: Symposium on Theory of Modeling and Simulation , p. 5 (2017)
47. Serrano, M.A., Melani, A., Bertogna, M., Quiñones, E.: Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In: Design, Automation & Test in Europe Conference & Exhibition, pp. 1066–1071 (2016)
48. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans. Comput. **39**, 1175–1185 (1990)
49. Shi, Y., et al.: Particle swarm optimization: developments, applications and resources. In: Proceedings of the 2001 Congress on Evolutionary Computation, vol. 1, pp. 81–86. IEEE, Piscataway (2001)
50. Spainhower, L., Gregg, T.A.: IBM s/390 parallel enterprise server G5 fault tolerance: a historical perspective. IBM J. Res. Develop. **43**, 863–873 (1999)
51. Takada, H., Sakamura, K.: A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In: IEEE Proceedings Real-Time Systems Symposium (RTSS) (1997)
52. Thangamuthu, S., Concer, N., Cuijpers, P.J.L., Lukkien, J.J.: Analysis of ethernet-switch traffic shapers for in-vehicle networking applications. In 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 55–60. IEEE, Piscataway (2015)
53. Thiele, D., Ernst, R., Diemer, J.: Formal worst-case timing analysis of ethernet tsn's time-aware and peristaltic shapers. In: 2015 IEEE Vehicular Networking Conference (VNC), pp. 251–258. IEEE, Piscataway (2015)
54. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel Distrib. Syst. **13**(3), 260–274 (2002)
55. Tsai, T.-Y., Chung, Y.-L., Tsai, Z.: Introduction to packet scheduling algorithms for communication networks. In: Communications and Networking. IntechOpen, London (2010)
56. Upasani, G., Vera, X., González, A.: Setting an error detection infrastructure with low cost acoustic wave detectors. In: IEEE Annual International Symposium on Computer Architecture (ISCA) (2012)

57. Upasani, G., Vera, X., González, A.: Avoiding core's due SDC via acoustic wave detectors and tailored error containment and recovery. In: ACM/IEEE International Symposium on Computer Architecture (ISCA) (2014)
58. Vaidyanathan, K., Trivedi, K.S.: Extended classification of software faults based on aging. In: Fast Abstract, International Symposium on Software Reliability Engineering, Hong Kong. Citeseer (2001)
59. Verucchi, M., Theile, M., Caccamo, M., Bertogna, M: Latency-aware generation of single-rate DAGs from multi-rate task sets. In: Real-Time and Embedded Technology and Applications Symposium, pp. 226–238 (2020)
60. Vincentelli, A.S., Giusto, P., Pinello, C., Zheng, W., Natale, M.D.: Optimizing end-to-end latencies by adaptation of the activation events in distributed automotive systems. In: Real Time and Embedded Technology and Applications Symposium, pp. 293–302 (2007)
61. Zhao, Q., Gu, Z., Zeng, H.: HLC-PCP: aresource synchronization protocol for certifiable mixed criticality scheduling. IEEE Embed. Syst. Lett. **6**, 8–11 (2013)
62. Zhao, S., Garrido, J., Burns, A., Wellings, A.J.: New schedulability analysis for MRSP. In: IEEE Embedded and Real-Time Computing Systems and Applications (ERTCSA) (2017)
63. Zhao, L., Pop, P., Craciunas, S.S.: Worst-case latency analysis for IEEE 802.1 Qbv time sensitive networks using network calculus. IEEE Access **6**, 41803–41815 (2018)