Design Automation for Embedded Systems (2023) 27:103–138
https://doi.org/10.1007/s10617-023-09271-x

Predictable timing behavior of gracefully degrading automotive systems

Philipp Weiss[1] · **Sebastian Steinhorst**[1]

Received: 21 July 2022 / Accepted: 24 March 2023 / Published online: 11 April 2023
© The Author(s) 2023

Abstract
Fail-operational behavior of safety-critical software for autonomous driving is essential as there is no driver available as a backup solution. In a failure scenario, safety-critical tasks can be restarted on other available hardware resources. Here, graceful degradation can be used as a cost-efficient solution where hardware resources are redistributed from non-critical to safety-critical tasks at run-time. We allow non-critical tasks to actively use resources that are reserved as a backup for critical tasks, which would be otherwise unused and which are only required in a failure scenario. However, in such a scenario, it is of paramount importance to achieve a predictable timing behavior of safety-critical applications to allow a safe operation. Here, it has to be ensured that even after the restart of safety-critical tasks a guarantee on execution times can be given. In this paper, we propose a graceful degradation approach using composable scheduling. We use our approach to present, for the first time, a performance analysis which is able to analyze timing constraints of fail-operational distributed applications using graceful degradation. Our method can verify that even during a critical Electronic Control Unit failure, there is always a backup solution available which adheres to end-to-end timing constraints. Furthermore, we present a dynamic decentralized mapping procedure which performs constraint solving at run-time using our analytical approach combined with a backtracking algorithm. We evaluate our approach by comparing mapping success rates to state-of-the-art approaches such as active redundancy and an approach based on resource availability. In our experimental setup our graceful degradation approach can fit about double the number of critical applications on the same architecture compared to an active redundancy approach. Combined, our approaches enable, for the first time, a dynamic and fail-operational behavior of gracefully degrading automotive systems with cost-efficient backup solutions for safety-critical applications.

Keywords Fail-operational systems · Graceful degradation · Automotive systems · Timing analysis

✉ Philipp Weiss
philipp.weiss@tum.de

Sebastian Steinhorst
sebastian.steinhorst@tum.de

[1] Department of Electrical and Computer Engineering, Technical University of Munich, Munich, Germany

1 Introduction

Fail-operational behaviour of safety-critical software is essential to enable autonomous driving. Without any driver as a backup solution, the failure of an ECU has to be handled by the software system. To manage increased software complexity, automotive electronic architectures are currently undergoing major changes. Instead of adding a new control unit for each new functionality, software is being integrated on a few, more powerful central control units [1].

By contrast, from a software perspective, this trend leads to a decentralization. In comparison to state-of-the-art monolithic ECU software, future software will be designed modular. Such a modular design will allow dynamic shifting of software components at run-time, including activation and deactivation of components on different ECUs. This perspective allows new strategies to enable fail-operational behaviour.

Additionally, automotive vendors are offering over-the-air software updates to regularly deliver the latest functionality. Here, customers will be able to purchase and enable features over an app store, which leads to unique and customized software systems.

Thus, a dynamic resource management is required, which maps applications at run-time as part of the software platform. A dynamic resource management allows to integrate new applications at run-time with unique solutions for an individual mix of applications. Furthermore, it enables a gracefully degrading system behaviour and allows the system to react to unplannable changes such as the defect of a hardware unit. Using a graceful degradation approach, safety-critical tasks can be restarted on other available hardware resources, while, in return, non-critical tasks are shut down to free resources. Therefore, instead of adding costly hardware redundancy to enable a fail-operational behaviour, existing resources can be repurposed. Decentralized run-time approaches have the advantage that there is no single-point-of-failure such that the system is still able to act after any ECU failure [2]. The advantage of using a passive backup solution compared to active redundancy is that almost no overhead is added in terms of required computational power.

The main challenge for such a system is to achieve a predictable system behavior. Most safety-critical applications have to meet real-time requirements, where a complete application execution has to finish within a deadline. To provide real-time guarantees, a performance analysis has to be based on a composable system such that the interference between applications can be bounded [3].

However, there is no approach yet that ensures a predictable timing behaviour of gracefully degrading systems where passive task instances are activated after a failure. To achieve a fail-operational behaviour it has to be ensured that the timing constraints are met under any circumstance. Finding a new task binding after a failure is not realistic as the backup solution has to be available immediately. Thus, an application binding can only be considered feasible if the deadline can be also met after restarting any of the passive task instances and a feasible backup solution is available for any possible failure.

In this paper, we present an agent-based mapping procedure based on a backtracking approach using a performance analysis to find feasible mappings at run-time. The agent-based system enables a decentralized control without a single point of failure. Each agent controls one task and is responsible for allocating resources and ensuring that constraints are met. The system is based on a composable scheduling technique that supports a gracefully degrading system behaviour. This allows to estimate an upper bound for the execution time not only for active tasks but also for passive tasks once they are started. Our approach includes

passive tasks in the mapping search such that there is a backup solution available that fulfills the real-time constraints under any ECU failure.

The mapping approach is intended to be performed while the car is not actively in use such that most system resources are available for performing the search. Once the system is in a stable state and running, the agents do not perform any action and, therefore, put no additional strain on the system resources. During run-time, monitoring with heartbeat messages and watchdogs is performed to detect ECU failures. The solutions found by the mapping approach include valid backup solutions for critical applications to which a fast failover can be performed after a failure has been detected. Here, we follow our main hypothesis, such that when the failure occurs a safe state can be reached with a minimum amount of communication and computation. The timing analysis of a failover itself is out of scope of this work, we refer any reader interested in the topic to the work of [4]. After a failover has been performed, safety-critical applications are able to stay operational. However, to ensure a safe continuation, the fail-operational behavior has to be re-established. Depending on the amount of remaining resources, a re-mapping of the applications could be performed during a safe halt on a parking space.

In this work, we make the following contributions:

- We analyze related work in Sect. 2 and provide an overview over related approaches in the fields of fail-operational systems, dynamic task mapping, graceful degradation and predictable timing behavior. We conclude that there is no work yet that allows an execution time analysis of gracefully degrading systems.
- After introducing our system model in Sect. 3, we introduce and adapt a state-of-the-art performance analysis based on composable scheduling to our system model in Sect. 4.
- We present our performance analysis for fail-operational systems which supports a gracefully degradable system behaviour in Sect. 5. This analysis also takes backup solutions into consideration and can evaluate whether the worst-case end-to-end application latency can still meet the deadline after switching to a backup solution during a failover. Furthermore, we introduce our gracefully degrading scheduling scheme.
- We present our agent-based run-time mapping procedure in Sect. 6 using our performance analysis to find feasible mappings that meet real-time requirements. Our approach includes passive tasks in the search such that it is ensured that all backup solutions meet the real-time constraints. Here, we also introduce three strategies which can strongly influence the degradation behavior. A reconfiguration of the system can be performed after any failure to re-establish the fail-operational behaviour of safety-critical applications.
- We evaluate our graceful degradation approach in our simulation framework in Sect. 7. Here, we compare our approach to an active redundancy approach by measuring the success rate and resource utilization over multiple experiments. Furthermore, we evaluate and discuss our three allocation and reservation strategies. Results show that around twice as much critical applications can be mapped onto the same architecture when using our graceful degradation approach compared to active redundancy approaches. We conclude that graceful degradation can greatly increase the success rate in scenarios where resources are limited if the risk of loosing non-critical functionality in a failure scenario is acceptable.

2 Related work

As our work is combining aspects from many different research areas, we organized the related work section into four subsections. First, we are presenting traditional fault-tolerant approaches in Sect. 2.1 which can be applied on device level but also on system level. Then we provide an overview over existing approaches in the field of dynamic mapping methodologies in Sect. 2.2. Afterwards, we discuss previous work on graceful degradation in Sect. 2.3. Last, we introduce work in Sect. 2.4 which has the goal of achieving predictable timing behavior using composable systems.

2.1 Fail-operational systems

In [5], the authors present an overview on existing fail-operational hardware approaches and introduce concepts for the implementation on a multi-core processor. The authors in [6] review common fault-tolerant architectures in System-on-a-Chip (SoC) solutions such as lock-step architectures, loosely synchronized processors or triple modular redundancy and perform a trade-off analysis. However, this work misses to apply fail-operational aspects on a system level instead of device level only.

The authors in [7] present a system-level simplex architecture to ensure a fail-operational behavior of applications but also to protect the underlying operating system, middleware, and microprocessor from failures. Here, the complex subsystem is driving the system as long as no failure occurs. A safety subsystem and a decision controller are running on a dedicated microcontroller. Similar authors in [8] present a fail-operational simplex architecture and address the problem of inconsistent states in Controller Area Network (CAN) controllers during a failover. As a solution, an atomic function stores the state and sends the message to avoid inconsistencies allowing to protect communication with peripherals. Although these approaches apply fail-operational capabilities on system level they are limited in flexibility and add a lot of cost as a dedicated hardware component is required.

Other work [9] addresses the automatic optimization of redundant message routings in automotive ethernet networks to enable fail-operational communication. In our work, redundant tasks are distributed over the system with redundant communication routes such that once a task is restarted there is always at least one communication path with preceding and succeeding tasks.

Overall, hardware components provided with failure detection and mitigation mechanisms are the base for enabling a dynamic fail-operational solution on system level. However, these solutions lack flexibility and are not dealing with the problem of providing a dynamic fail-operational behavior for an entire system consisting of many applications distributed over multiple ECUs.

2.2 Dynamic mapping

There has been a lot of work in the field of dynamic mapping approaches. The authors in [10] present a decentralized and dynamic mapping approach for Network-on-Chip (NoC) architectures. Their mapping approach takes bandwidth and load constraints into consideration and uses the best-neighbour strategy, which takes only the closest search space around a task into account. In [11] an agent-based run-time mapping approach for heterogeneous NoC architectures is presented. The system is based on global agents containing system state information and cluster agents which are responsible for assigning resources. Here,

the main motivation is to reduce computational effort and global traffic for monitoring the system utilization when mapping the distributed applications. The authors in [12] present a centralized run-time mapping approach with the goal of reducing network load in NoC-based Multiprocessor-System-on-a-Chip (MPSoC) systems. Here, a dedicated manager processor is taking care of mapping initial tasks of each application to a cluster. As a dynamic workload is considered, all following tasks are mapped at run-time of the application upon communication requests. Multiple heuristics considering the channel load are presented and evaluated. The authors argue that it is reasonable to use greedy algorithms as they can provide quick results in exchange for lower search space exploration quality. They conclude that compared to static optimization methods the moderate overhead of solutions found by dynamic methods is acceptable considering the gain in flexibility.

Most of these approaches have the goal of increasing flexibility and at most consider being able to reconfigure the system when a faulty component is found. However, none of them take fail-operational or timing requirements into account.

2.3 Graceful degradation

The authors in [2] introduced an agent-based approach which finds task-mappings at run-time and ensures a fail-operational behavior of critical applications by applying graceful degradation. Here, agents can allocate and reserve parts of resources to ensure degradation. However, this work does not include any base for a timing analysis such that no guarantees to timing constraints can be given. Furthermore, we introduce a performance analysis for such a gracefully degrading system which we embed at run-time to ensure that timing constraints are met even in failure scenarios. The authors in [13] have presented a design-time analysis to find valid application mappings in mixed critical systems. Here, applications can have multiple redundancies based on their fail-operational level and the system can be degraded by shutting down optional software components. By contrast, in our approach, instead of having a limited amount of redundancy, we re-establish lost redundancy after a failover. The authors in [14] present a degradation-aware reliability analysis in which tasks are grouped according to different safety levels. Using design space exploration the reliability of the degradation modes is optimized. The work in [15] calculates the utility of a system by decomposing it into feature subsets, which may be defined by functional or non-functional attributes, such that the utility of the system in different degradation modes can be analyzed. However, they do not propose any approach on designing a gracefully degrading behavior. Other work such as [16] has been proposing to relax constraints once an anomaly occurs, leading potentially to degraded system behavior.

Overall, none of the mentioned work related to graceful degradation takes timing constraints into consideration such that a safe execution of time-critical applications can not be guaranteed.

2.4 Predictable timing behavior

The authors of [3] describe the two concepts of predictability and composability which can be used to reduce complexity and to verify real-time requirements. In composable systems, applications are isolated such that they do not influence each other allowing to verify their timing behavior independently. Furthermore, using formal analysis, lower bounds on performance can be guaranteed. The work of [17] uses hybrid application mapping to combine design-time analysis with run-time application mapping. The spatial and temporal isolation techniques

together with a performance analysis are based on the concepts from [3]. At design-time a design space exploration with a formal performance analysis finds pareto-optimal configurations. A run-time manager then searches for suitable mappings of these optimized solutions. In contrast to their work, we extend the scheduling techniques and include passive tasks and messages in the performance analysis to enable graceful degradation. Furthermore, our application mapping is performed entirely at run-time with an agent-based approach.

In the real-time mixed-criticality systems community there has been work on guaranteeing reduced service to low criticality tasks after switching to a safety mode when a critical task can not meet its deadline [18]. By contrast, in our work we do not switch between two modes, but only degrade tasks if the resources of it have been reserved and are claimed by a critical task. Furthermore, the approaches in the literature are mostly not focusing on distributed systems and are not taking fail-operational aspects into consideration.

Previous work [4] has presented a formal analysis to derive the worst-case application failover time for distributed systems. The authors analyzed the impact of failure detection and recovery times on the timing behavior of distributed applications. This analysis guarantees an upper bound on the time that it would take for an application to generate a new output after the failure of one or multiple tasks. Our current work presented in the paper at hand could be used as a base for the analysis performed in [4] as an upper bound on task execution and message transmission is assumed to be given there and can now be calculated. Similar to this work the authors in [19] present a worst-case timing analysis for tasks with hot and passive standbys. However, the topic of graceful degradation is not addressed in this work.

Related to this topic authors in [20] have analyzed the timing behavior of task migration at run-time if a new mapping has to be found. Their deterministic mapping reconfiguration mechanism identifies efficient migration routes and determines the worst-case reconfiguration latency. To find new application mappings and to transition predictably to new configurations again, run-time mechanisms are combined with an off-line design space exploration. In [21] the same authors together with others present a general overview of hybrid application mapping techniques and composable many-core systems.

Overall, there has not been any work in the field of predictable timing behavior that considers a gracefully degrading system architecture. For this purpose we build mainly upon our previous work in [2] and the work from [17] to create, for the first time, a comprehensive approach that enables a safe and efficient fail-operational behavior of time-critical applications in distributed systems by predictably analyzing the execution time behavior and allowing a gracefully degrading system behavior.

3 System model

3.1 System

In the past, automotive vendors added a new ECU for each new functionality in the vehicle. Today, cars consist often of more than 100 ECUs to control functions in the domains like infotainment, chassis, powertrain or comfort [1]. Now the automotive industry is aiming towards zonal or more centralized architectures. Some vendors such as Tesla prefer a centralized architecture, where most of the functions are executed on a single ECU such as the FSD computer of Tesla [22]. Bosch is developing a vehicle-centralized, zone-oriented Electrical/Electronic (E/E) architecture with a few centralized powerful vehicle computers

integrating cross-domain functionality similar as [23, 24]. These vehicle computers are connected to actuators and sensors via zone ECUs. This reduces the required wiring and weight in vehicles but also system complexity.

In our work we focus on the deployment of bigger applications on a future system architecture which consists of a set of a few ECUs $e \in E$ which are interconnected via switches and a set of Ethernet links $l \in L$. The ECUs and Ethernet links use Time-Division Multiplexing (TDM) scheduling with pre-determined time slices which can be allocated or reserved. To dynamically activate, deactivate, and move tasks on the platform at run-time, we implemented a middleware which is based on SOME/IP [25], an automotive middleware solution. This middleware includes a decentralized service-discovery to dynamically find services in the system and a publish/subscribe scheme to publish and subscribe to events.

3.2 Criticality

In our work we are exploring graceful degradation methodologies. Here, critical applications e.g. for autonomous driving can be restarted after a failure on another ECU. Instead of exclusively reserving resources for this scenario, non-critical applications e.g. from the infotainment domain can be shut down to free resources.

According to the ISO 26262 standard, applications can be assigned one of four Automotive Safety Integrity Levels (ASILs) (A to D) [26]. However, we do not differ between criticality levels of critical applications in our work as there is no justification in shutting down applications with an assigned ASIL of A for an application with an assigned ASIL of B as the failure of any critical application can have safety-critical consequences. Instead it has to be ensured that all safety goals are met for any critical application. Therefore, we only distinct between critical and non-critical applications. In our work we assume that each critical application has fail-operational requirements. This means that the application has to stay operational even if a failure occurs that affects this application. We define *critical* and *non-critical* applications as follows:

- *Critical application* An application that has fail-operational requirements. To ensure a fail-operational behavior passive redundancy on task level is applied. In a failure scenario resources of non-critical applications might be used to keep critical applications operational.
- *Non-critical application* An application without specific safety requirements. Non-critical applications can be shut down to free resources for critical applications even if they are not directly affected by a failure.

3.3 System software

Our system software consists of a set of applications $a \in A$, which are composed of tasks $t \in T$. The tasks t of an application a can be distributed across multiple ECUs. Applications are executed periodically with a period P_a and we assume each application has to meet a deadline δ, with the period P_a being at least as long as the deadline δ. For every task we assume that the Worst-Case Execution Time (WCET) $W(t)$ is known.

We model each application a by an acyclic and directed application graph $G_A(V, E)$. Safety-critical application have to fulfill fail-operational requirements and, thus, have to remain operational even during critical ECU failures. Therefore, we assume that redundant passive task instances are required for our safety-critical applications. The vertices $V =$

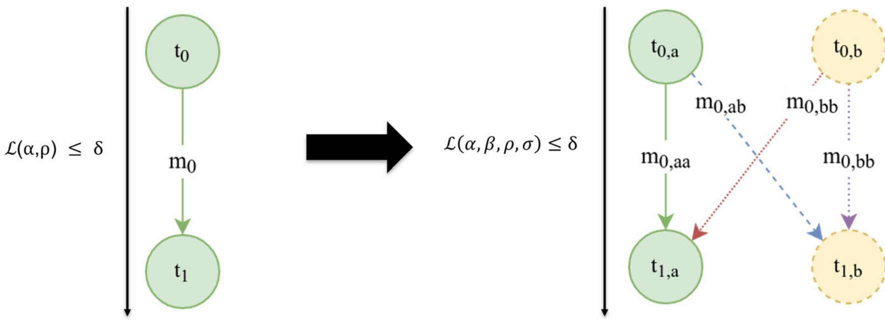


Fig. 1 An exemplary application graph $G_A(V, E)$ of a non-critical application (left) and a critical application (right). The application graph of the critical application consists of two active task instances $t_{0,a}, t_{1,b} \in T_a$ and two passive task instances $t_{0,b}, t_{1,b} \in T_b$. Furthermore, it contains one active message instance $m_{0,aa} \in M_a$ and three backup message instances $m_{0,ab}, m_{0,ba}, m_{0,bb} \in M_b$, which are required to ensure there is always a communication path between the tasks instances available. (Color figure online)

$T_a \cup T_b$ of the application graph $G_A(V, E)$ are composed of the set of active task instances T_a and the set of passive task instances T_b. The edges $E = M_a \cup M_b$ of the application graph $G_A(V, E)$ are composed of the set of active messages M_a and the set of backup messages M_b. In the following are our definitions for *active* and *passive task instances* and *active* and *backup message instances*:

- *Active task instance* A task instance of a critical or non-critical application. This is the default task instance actively executing any workload. A binding $\alpha : T \to E$ assigns an active task instance $t \in T$ to an ECU $\alpha(t) \in E$.
- *Passive task instance* A backup instance $t \in T_b$ of an active task instance which is part of a critical application. The passive task instance is only activated if its active counterpart is affected by a failure. Here, the binding $\beta : T \to E$ assigns a passive task instance $t \in T_b$ to an ECU $\beta(t) \in E$.
- *Active message instance* A routing is required for each active message instance $m \in M_a$ which is part of a critical or non-critical application. A routing $\rho : M \to 2^L$ assigns each message $m \in M_a$ to a set of connected links $L' \subseteq L$ that establish a route $\rho(m)$. We use the shortest path routing obtained through Dijkstra's algorithm such that there is only a single route between two ECUs e available [27].
- *Backup message instance* For critical applications three backup message instances $m \in M_b$ are required of which one will get activated after an ECU failure depending on which passive task instances get activated. A routing $\sigma : M \to 2^L$ assigns each backup message $m \in M_b$ to a set of connected links $L' \subseteq L$ that establish a route $\sigma(m)$.

The application graph of non-critical applications consists only of active task instances $t \in T_a$ and active messages $m \in M_a$.

Figure 1 presents a non-critical and a critical application according to our system model. The non-critical application consists of two tasks and one message being sent between the two tasks. For the safety-critical application the graph also includes two passive task instances and three backup message instances. Three backup message instances are required such that it can be ensured that always a communication between two task instances is possible regardless of which task instances are affected by a failure. The message instances $m_{0,ab}$ and $m_{0,ba}$ are required if only one of the active task instances is failing, while the message instance $m_{0,bb}$ is

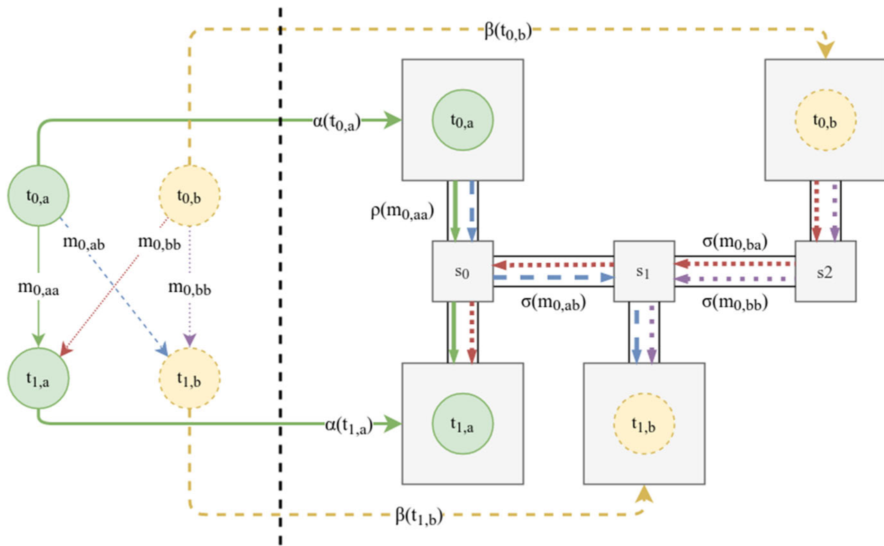


Fig. 2 Exemplary mapping of a safety-critical application onto a hardware architecture consisting of four ECUs e_0, e_1, e_2, and e_3, and three switches s_0, s_1, and s_2. The green arrows indicate the active bindings of tasks t_0 and t_1, while the dashed yellow arrows indicate the passive task bindings. The routings of the message instances are indicated by the same arrow color and style as in the application graph. (Color figure online)

required if both active task instances are affected by a failure, e.g. because they are mapped onto the same failing ECU.

Figure 2 shows the binding α of the active task instances $t \in T_a$ and the binding β of the passive task instances $t \in T_b$ onto a system architecture. The routing ρ of the active message instance $m \in M_a$ and the routings σ of the backup message instances $m \in M_b$ are also marked by colored arrows.

3.4 Failures

In this work and our experiments we focus on mitigating ECU failures which are detected by watchdogs and heartbeats. Our graceful degradation approach ensures that safety-critical applications can keep running after an ECU failure while there is no guarantee for non-critical applications. As redundancy is used for all safety-critical tasks, the system could withstand any single ECU failure and perform a failover for affected task instances. After a failover, any critical application is still able to stay operational such that any hazards can be avoided. After a time-critical failover has been performed, fail-operational capabilities have to be re-established with a new mapping process. Applications might have to be stopped temporarily if active task instances have to be remapped such that a safe mapping could only be performed during a halt. To prevent this, methods such as proposed in [28] to perform a safe real-time task migration could be applied. Our approach could be also used to mitigate transient or software failures if the corresponding failure detection mechanisms are supported by hardware or software e.g. through a lock-step architecture. If possible, a failure should be handled locally. Our solution is intended to be used as a last resort, as a failover could lead to the shut down of non-critical applications when graceful degradation is applied. While our approach considers that redundant message instances are required to ensure that

a communication is possible after a failover, the mitigation of network or switch failures are out of scope of this work. For the interested reader we recommend the work of [9] on this topic.

3.5 Failover

Our approach ensures that applications are executed withing a deadline δ and that a backup mapping is available that is also meeting this timing constraints under regular operation. In the case of an ECU failure we consider that the current application execution might not finish if an active task instance is affected directly by the failure and that application execution might be interrupted for a certain time interval. Here, it is important that a failover within the Fault Tolerant Time Interval (FTTI) can be guaranteed [29]. In the work at hand, we do not focus on giving a failover timing guarantee but that a stable application execution within a timing constraint is always readily available once the failover is done. We refer any reader interested in the topic of failover timing analysis to the work of [4]. Furthermore, we consider that no critical states have to be transferred and recovered. However, if required checkpoints can be periodically transmitted from active to passive task instances to save important state data [30]. After the failure recovery computation can be continued with the latest transmitted checkpoints.

4 Performance analysis

In this section we contribute our concept of performance analysis for distributed applications based on the work from [3, 17]. Using this performance analysis we present our performance analysis of gracefully degrading systems in Sect. 5. Instead of targeting NoC architectures as in [17] we target a distributed electronic system consisting of multiple ECUs which are connected via switches and Ethernet links. The goal of the performance analysis is to find an upper bound for the end-to-end application latency such that it can be verified whether a mapping adheres to a timing constraint. Here, we first formally introduce the end-to-end application latency in Sect. 4.1 and present our derived analytical formulas for distributed and composable system. Afterwards, we present composable scheduling together with our adapted analytical formulas for both task and message scheduling in Sect. 4.2. Compared to the work in [17] we chose TDM instead of a Round-Robin (RR) scheme for task scheduling. The disadvantage of RR scheduling is that the execution latency depends on the number of other tasks in the schedule such that the execution latency might change once tasks are added to the schedule later. By contrast, when using a schedule based on TDM, an upper bound of service intervals that can be allocated by tasks is already predefined which can be used for a worst-case estimation of the execution latency. For message scheduling we use TDM as well where we design the system such that exactly one Ethernet frame can be sent per slot and assume that all messages can be sent within one frame. In case messages should be able to be split up into multiple slots we refer the interested reader to [31] and [17].

4.1 End-to-end application latency

For time-critical applications, a mapping can only be considered feasible if the worst-case end-to-end application latency $\mathcal{L}(\alpha, \rho)$ does not exceed a given deadline δ such that the

following constraint has to be met:

$$\mathcal{L}(\alpha, \rho) \leq \delta. \tag{1}$$

The end-to-end latency of a distributed application is influenced by both executing computational tasks t and sending messages m between the tasks. However, task execution times and message transmission times will most certainly differ between each iteration. This highly depends on the paths being taken in a program and interference caused by other applications which are executed concurrently in the system. The interference time is mainly influenced by the scheduling and admission algorithms that are used for the Central Processing Unit (CPU), the network infrastructure and other shared resources. The worst-case end-to-end execution latency $\mathcal{L}(\alpha, \rho)$ is determined by the critical path as

$$\mathcal{L}(\alpha, \rho) = \max_{\forall path \in paths(G_A(V,E))} PL(path, \alpha, \rho), \tag{2}$$

where the critical path is the path through an application with the highest aggregated latency. The path latency $PL(path, \alpha, \rho)$ itself can be calculated as

$$PL(path, \alpha, \rho) = \sum_{\forall t \in path \cap T} TL(t, \alpha(t)) + \sum_{\forall m \in path \cap M} CL(m, \rho(m)), \tag{3}$$

by summing up all task latencies $TL(t, \alpha(t))$ and communication latencies $CL(m, \rho(m))$ of tasks and messages which lie in this path. To predictably calculate these worst-case task latencies $TL(t, \alpha(t))$ and worst-case communication latencies $CL(m, \rho(m))$ it is required to analytically determine an upper bound. To reduce complexity, composability is required to ensure that applications have only a bounded effect on each other. Well-known scheduling approaches such as RR or TDM temporally isolate task execution or message transmission on a resource.

In Fig. 3 an exemplary mapping of a non-critical application with annotated worst-case task and communication latencies is presented. The path latencies for the two paths $p_0 = (t_0 - m_0 - t_1)$ and $p_1 = (t_0 - m_1 - t_2)$ in the application graph can be calculated as $PL(p_0, \alpha, \rho) = 30$ ms and $PL(p_1, \alpha, \rho) = 35$ ms. The critical path p_1 leads to a worst-case end-to end application latency of $\mathcal{L}(\alpha, \rho) = 35$ ms. With a deadline of $\delta = 40$ ms Eq. 1 would be still fulfilled.

4.2 Composable scheduling

We use TDM scheduling for tasks and messages as this allows a partitioned analysis, where applications can be mapped and analyzed independent from each other.

4.2.1 Task scheduling

In general, the task latency $TL(t, \alpha(t))$ consists of the actual task execution time $TL_{exec}(t, \alpha(t))$ and the task interference time $TL_{inter}(t, \alpha(t))$, which a task spends waiting e.g. due to scheduling:

$$TL(t, \alpha(t)) = TL_{exec}(t, \alpha(t)) + TL_{inter}(t, \alpha(t)). \tag{4}$$

Thus, the worst-case task execution time without interference $TL_{exec}(t, \alpha(t))$ is a multiple of the service interval time τ_{SI} such that we can calculate it using the WCET $W(t, \alpha(t))$ as

$$TL_{exec}(t, \alpha(t)) = \lceil \frac{W(t, \alpha(t))}{\tau_{SI}} \rceil \cdot \tau_{SI}. \tag{5}$$

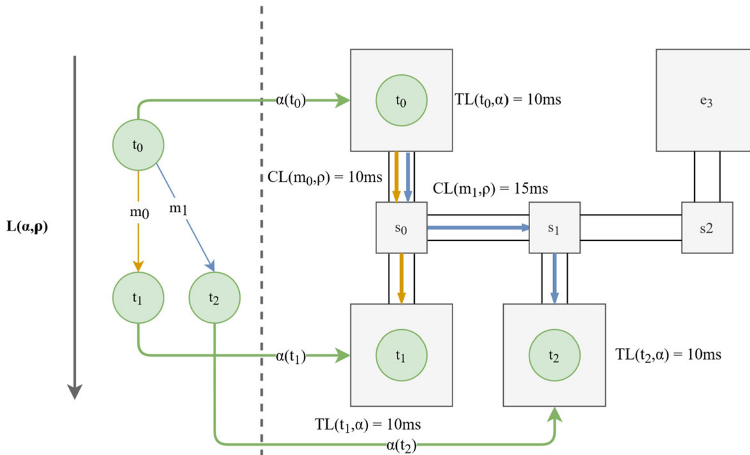


Fig. 3 Exemplary mapping of a non-critical application onto a hardware architecture consisting of four ECUs e_0, e_1, e_2, and e_3, and three switches s_0, s_1, and s_2. The green arrows indicate the binding of a task to an ECU. The message routings are marked in the color of the corresponding message in the application graph. There are two paths $p_0 = (t_0 - m_0 - t_1)$ and $p_1 = (t_0 - m_1 - t_2)$ in the application graph. Taking the task and communication latencies from the figure, the path latencies can be calculated as $PL(p_0, \alpha, \rho) = 30$ ms an $PL(p_1, \alpha, \rho) = 35$ ms. With $p1$ as the critical path, the worst-case end-to-end application latency can be calculate as $\mathcal{L}(\alpha, \rho) = 35$ ms. With a deadline of $\delta = 40$ ms the constraint in Eq. 1 would be met. (Color figure online)

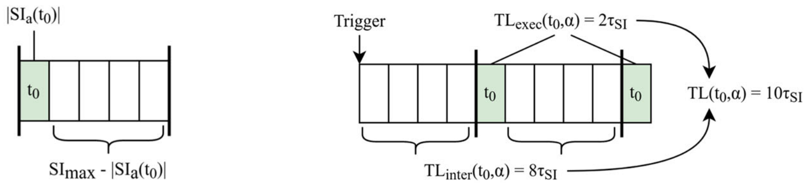


Fig. 4 Example of a task schedule with a maximum amount of allocatable service intervals of $SI_{max} = 5$. One service interval $|SI_a| = 1$ is allocated for the task t_0. With a WCET of $TL_{exec}(t_0, \alpha) = W(t_0, \alpha) = 2\tau_{SI}$, two full execution cycles are required in the worst case for the task to finish execution. With the corresponding worst-case task interference time of $TL_{inter}(t_0, \alpha) = 8\tau_{SI}$, the worst-case task latency can be calculated as $TL_{inter}(t_0, \alpha) = 10\tau_{SI}$. (Color figure online)

Given a number of service intervals $SI_a(t)$ that are allocated for a task t and a defined maximum number of service intervals SI_{max} we can determine the worst-case interference time $TL_{inter}(t, \alpha(t))$ as

$$TL_{inter}(t, \alpha(t)) = \lceil \frac{W(t, \alpha(t))}{|SI_a(t)| \cdot \tau_{SI}} \rceil \cdot (SI_{max} - |SI_a(t)|) \cdot \tau_{SI}. \tag{6}$$

Here, $SI_{max} - |SI_a(t)|$ reflects the number of service intervals that a task would have to wait until it is executed again. The first factor represents how many times the task would have to wait until its turn in the worst-case. An exemplary task schedule with the derivation of the worst-case task latency $TL(t, \alpha(t))$ is presented in Fig. 4.

4.2.2 Message scheduling

We assume that messages are sent over ethernet links oppose to the work of [17] where smaller links connect multiple processing elements on a NoC architecture. For the worst-case communication latency we can proceed likewise as for the task scheduling to calculate the worst-case communication latency $CL(m, \rho)$ with the worst-case message transmission time $CL_{trans}(m, \rho)$ and the worst-case communication interference time $CL_{inter}(m, \rho)$:

$$CL(m, \rho(m)) = CL_{trans}(m, \rho(m)) + CL_{inter}(m, \rho(m)) \tag{7}$$

For the message scheduling we use the notation SL to describe the time frame of a slot interval. We design the system such that exactly one ethernet frame with a maximum frame size of 1518 bytes can be sent in one time slot. We assume that message sizes do no exceed the Maximum Transmission Unit (MTU) of an ethernet frame such that only one slot has to be allocated per message. Using this we can calculate the tranmission time of a message over one ethernet link $CL_{trans}(m, l)$ as

$$CL_{trans}(m, l) = \tau_{SL}. \tag{8}$$

To calculate the interference time of a message over one link $CL_{inter}(m, l)$ we assume that a maximum number of slots SL_{max} is defined. As the transmission of the message requires only one slot, a message has to wait one transmission round in the worst case:

$$CL_{inter}(m, l) = (SL_{max} - 1) \cdot \tau_{SL}. \tag{9}$$

Combining these two worst-case latencies we can calculate the worst-case communication latency $CL_(m, l)$ of a message m over one link l as

$$CL_(m, l)=CL_{trans}(m, l)+CL_{inter}(m, l)=(SL_{max} - 1) \cdot SL+SL=SL_{max} \cdot \tau_{SL}. \tag{10}$$

This approach allows us to analyze the worst-case communication latency over each link individually as

$$CL(m, \rho(m)) = \sum_{\forall l \in \rho(m)} CL(m, l) \tag{11}$$

Under the assumption that all links in the system are designed equally, the communication latency only depends on the number of links that the message m is passing on its route ρ, further denoted as $hops(\rho(m))$, which allows us to further simplify the formula to

$$CL(m, \rho(m)) = hops(\rho(m)) \cdot SL_{max} \cdot \tau_{SL}. \tag{12}$$

Our formulas are based on the assumption that a message requires exactly one slot to be transmitted. In case the system should be designed more fine granularly such that a message could require multiple slots for transmission, we refer the interested reader to [31] and [17].

5 Performance analysis of gracefully degrading systems

Using a state-of-the-art analysis presented in Sect. 4 it is possible to analyze whether a configuration of an application consisting of active tasks is meeting a deadline δ. However, we need to ensure that critical applications are still able to continue full operation after any ECU failure without violating Eq. 1. Here, we add passive tasks as a backup solutions that are started once the active task is affected by a failure. Furthermore, we are enabling a

gracefully degrading behaviour as in [2] such that resources that are allocated by non-critical applications can be used by critical applications if required. The advantage of using our passive backup solution compared to active redundancy is that during operation no overhead is added in terms of required computational power.

Theoretically, it would be possible to search for a new valid configuration which satisfies the timing constraint after an ECU failure occurred. However, this approach would have multiple disadvantages accompanied with highly unpredictable behaviours. First, it can not be guaranteed that sufficient resources are available for task computations and message transmissions after a failure even if a degradation approach is used. The ECU failure reduces the system-wide resource pool such that not every application might be able to find sufficient resources. Even if the system was over-designed, the next failure would increase the uncertainty further. Second, even if sufficient resources were still available, it is uncertain if a mapping could be found that would satisfy the timing constraint. Third, it could take a lot of time to find a valid solution. Even if a valid solution was available it could take an unknown amount of time to find one. Although the unavailability of an application might be tolerable for a certain amount of time (FTTI) during a failover, it would not be predictable how long it would take to find a solution and most likely it would not be found in time. Fourth, it would be unpredictable which non-critical applications would be shut down due to degradation as this would be only decided after the occurrence of the failure. Most importantly, this means it is uncertain if a solution can be found in time yet if one is existing, which is unarguably an unacceptable behaviour for safety-critical applications. Furthermore, it would be at least desirable to also allow a more predictable degradation behaviour of non-critical applications.

Therefore, it is necessary to improve uncertainty and achieve a more predictable behaviour. To bypass having to find new solutions after an ECU failure, we have to ensure that a suitable backup solution is already available for any ECU failure in the system. To ensure this we add redundant tasks such that there is at least one instance of each task available after an ECU failure that has sufficient resources to continue operation and to communicate with other tasks. Here, we present our performance analysis of gracefully degrading systems to verify that these backup solutions always meet timing constraints in Sect. 5.1. This solution allows a predictable system behaviour as it is known if a valid backup solution is available that meets resource and timing constraints before any failure and allows to quickly switch to this backup solution. Last, we present our composable scheduling of gracefully degrading systems in Sect. 5.2, which allows to independently derive worst-case latencies while also enabling a gracefully degrading system behavior. This solution also allows to predict in which failure scenario a non-critical application will be shut-down due to graceful degradation.

5.1 End-to-end application latency

A valid binding needs to ensure that the deadline δ is not only met by the active part of the application, but also in case any of the backup solutions have to be used in a failure scenario. The activation of passive task instances could lead to a new critical path in the application which might not meet the deadline δ. The worst-case end-to-end application latency $\mathcal{L}(\alpha, \beta, \rho, \sigma)$ considers not only the worst-case end-to-end application latency $\mathcal{L}(\alpha, \rho)$ of currently active task instances but also of any possible future configurations, where passive task instances are activated. To achieve a fail-operational behaviour, a valid mapping of a safety-critical application has to fulfill the following constraint:

$$\mathcal{L}(\alpha, \beta, \rho, \sigma) \leq \delta. \tag{13}$$

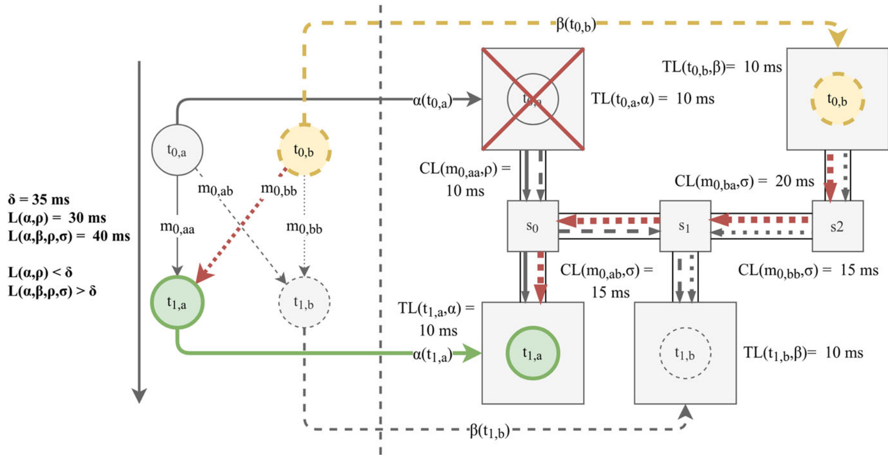


Fig. 5 Exemplary mapping of a safety-critical application onto a system architecture with annotated worst-case latencies. When using state-of-the-art performance analysis and disregarding the passive task and message instances the worst-case end-to-end application latency $\mathcal{L}(\alpha, \rho) = 30$ ms would meet the deadline $\delta = 35$ ms. However, in a failure scenario where $t_{0,a}$ was affected by a failure and $t_{0,b}$ activated, the deadline could no longer be met leading to a configuration that violates the timing constraint. Our performance analysis takes the highlighted critical path $p_2 = (t_{0,b} - m_{0,ba} - t_{1,a})$ with a worst-case path latency of $PL(p_2, \alpha, \beta, \rho, \sigma) = 40$ ms into account leading to a worst-case end-to-end application latency of $\mathcal{L}(\alpha, \beta, \rho, \sigma) = 40$ ms. Here, a violation of the timing constraint with the deadline $\delta = 35$ ms can be identified. Therefore, our performance analysis can be used to quickly evaluate different configurations and help design automation algorithms to find a valid configuration. (Color figure online)

As any activation of a passive task instance could potentially lead to a violation of this constraint, any path through the instance graph $G_B(V, E)$, including passive task instances $t \in T_b$ and backup messages $m \in M_b$, has to be considered. Therefore, we can define the worst-case end-to-end application latency $\mathcal{L}(\alpha, \beta, \rho, \sigma)$ as the critical path through the instance graph $G_B(V, E)$, which also includes all possible backup solutions:

$$\mathcal{L}(\alpha, \beta, \rho, \sigma) = \max_{\forall path \in paths(G_B(V,E))} PL(path, \alpha, \beta, \rho, \sigma), \tag{14}$$

The path latency $PL(path, \alpha, \beta, \rho, \sigma)$ depends not only on the worst-case latency of active task instances $TL(t, \alpha(t))$ and the worst-case communication latency of active messages $CL(m, \rho(m))$, but also on the potential worst-case latency of passive task instances $TL(t, \beta(t))$ and the potential worst-case latency of backup messages $CL(m, \sigma(m))$, such that it can be calculated as

$$\begin{aligned} PL(path, \alpha, \beta, \rho, \sigma) = &\sum_{\forall t \in path \cap T_a} TL(t, \alpha(t)) + \sum_{\forall t \in path \cap T_b} TL(t, \beta(t)) + \\ &\sum_{\forall m \in path \cap M_a} CL(m, \rho(m)) + \sum_{\forall m \in path \cap M_b} CL(m, \sigma(m)). \end{aligned} \tag{15}$$

From this formula it can be observed that it is also necessary to find a bound on the potential worst-case latency of passive task instances $TL(t, \beta(t))$ and the potential worst-case latency of backup messages $CL(m, \sigma(m))$.

For illustration let us assume that a deadline $\delta = 35$ ms is given for the safety-critical application as presented in Fig. 5. When disregrading passive task and message instances and using the annotated worst-case latencies from the figure, state-of-the-art performance

analysis as presentend in [17] would conclude a worst-case end-to-end application latency of $\mathcal{L}(\alpha, \rho) = 30$ ms which would meet the deadline $\delta = 35$ ms. However, in a failure scenario where $t_{0,a}$ is affected by a failure and $t_{0,b}$ activated, the deadline can no longer be met leading to a configuration that violates the timing constraint. By constrast, our performance analysis takes passive task and messages instances into account and, therefore, is able to identify the highlighted critical path $p_2 = (t_{0,b} - m_{0,ba} - t_{1,a})$ with a worst-case path latency of $PL(p_2, \alpha, \beta, \rho, \sigma) = 40$ ms. A violation of the timing constraint with a worst-case end-to-end application latency of $\mathcal{L}(\alpha, \beta, \rho, \sigma) = 40$ ms and the deadline $\delta = 35$ ms is clearly visible. Here, our performance analysis can be used to quickly evaluate different configurations and help design automation algorithms to find a valid configuration.

5.2 Composable scheduling of gracefully degrading systems

We extend state-of-the-art scheduling such as presented in [17] by introducing the concept of graceful degradation. For our analysis and experiments we apply graceful degradation to CPU resources although the concept can also be applied to link resources as well. Applied to our composable schedules, this means that service intervals can not only be allocated for a task but also reserved. A reservation indicates that the corresponding service interval is currently not in use, but might be used and turned into an allocation once the backup instance is being used. Service intervals that can be reserved are empty service intervals that have not been allocated yet or service intervals which are already allocated by non-critical applications. The allocation of slots works vice-versa, non-critical applications can allocate service intervals that are free or which are already reserved by critical applications. Critical applications on the other hand can only allocate service intervals that are completely free. If a service interval is allocated by a non-critical application and also reserved by a critical application the graceful degradation approach is applied. In case there is a failure in the system and critical passive task instances have to be started to mitigate a failure, the reservation of the resources will be turned into an active allocation and any non-critical tasks which formerly held an allocation of the corresponding slots are shut down. Depending on the application, it could then be decided if the degraded non-critical application keeps running in a degraded mode or is completely shut down.

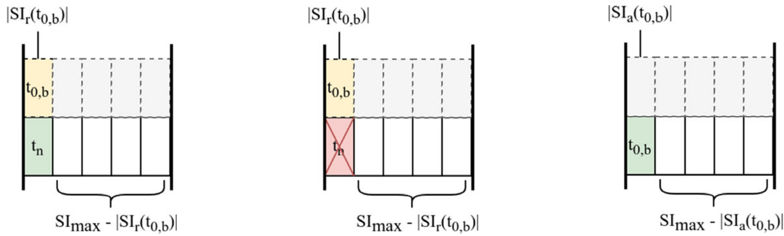
For the latency analysis of the backup solutions itself it is not relevant whether a reserved slot is also allocated and graceful degradation is applied or not as the reservation will be turned into an allocation. This procedure allows to predict before the occurrence of any failure if a valid backup solution can be found, but also allows a predictable behaviour of the graceful degradation approach. Furthermore, we can reuse our analytical formulas from Sect. 4.2 to calculate $TL(t, \beta(t))$ based on the number of reserved service intervals $SI_r(t)$ and the binding of the passive task instance $\beta(t)$:

$$TL(t, \beta(t)) = TL_{exec}(t, \beta(t)) + TL_{inter}(t, \beta(t)), \tag{16}$$

$$TL_{exec}(t, \beta(t)) = \lceil \frac{W(t, \beta(t))}{\tau_{SI}} \rceil \cdot \tau_{SI}, \tag{17}$$

$$TL_{inter}(t, \beta(t)) = \lceil \frac{W(t, \beta(t))}{|SI_r(t)| \cdot \tau_{SI}} \rceil \cdot (SI_{max} - |SI_r(t)|) \cdot \tau_{SI}. \tag{18}$$

To calculate the worst-case communication latency $CL(m, \sigma(m))$ of backup message instances, under the same assumptions as in Sect. 4.2, we can reuse Eq. 12 based on the



(a) Task schedule with reservation and allocation.
(b) Non-critical task t_n being degraded.
(c) Reservation turned into an allocation.

Fig. 6 Example of a task schedule with a maximum amount of allocatable (lower) and reservable (upper) service intervals of $SI_{max} = 5$. One service interval $|SI_r| = 1$ is reserved for the critical task instance $t_{0,b}$. The same service interval is allocated by the non-critical task instance t_n. In a failure scenario where $t_{0,b}$ has to be activated, t_n is being degraded in a first step. Afterwards, $t_{0,b}$ is taking over the allocation of the service interval. (Color figure online)

routing of the backup message instances σ:

$$CL(m, \sigma) = hops(\sigma(m)) \cdot SL_{max} \cdot \tau_{SL}. \quad (19)$$

In Fig. 6 an exemplary task schedule with reservations and allocations is presented. The upper service intervals indicate a reservation, while the lower service intervals indicate an allocation of the same service interval. In this example the first service interval is allocated by the non-critical task t_n but also reserved by the critical task instance $t_{0,b}$. In a failure scenario where the task instance $t_{0,b}$ is activated to serve as a backup solution, the non-critical t_n first looses its allocation and, thus, is being degraded. Afterwards, the reservation of the task instance $t_{0,b}$ is turned into an active allocation such that this resource can now be used exclusively by the critical task instance.

The decision which service interval should be allocated or reserved is not straightforward. When always a completely free service interval is taken first, then the schedule might run out of service intervals earlier such that not sufficient resources might be left for other tasks. On the other hand, when it is tried to overlap service intervals as much as possible but more than sufficient service intervals are available for all tasks, then an avoidable degradation might occur in a failure scenario. We present and discuss three different allocation and reservation strategies in Sect. 6.4 and evaluate them in Sect. 7. Before that, we introduce in the following section our agent-based mapping approach and the components required to implement such a gracefully degrading behavior at run-time.

6 Agents—finding feasible solutions at run-time

We use a dynamic agent-based approach to perform mapping and constraint checking at run-time. In the following we describe the components involved in the mapping process and the algorithms used by the agents. Each agent controls one task instance and is responsible for allocating resources and ensuring that constraints are met. Together the agents enable a decentralized control without a single point of failure. This concerns both active and passive task instances. The agents are able to communicate in a predefined order with each other to manage the mapping process of all task instances of an application. To dynamically activate, deactivate, and move tasks on the platform at run-time, we implemented a middleware which is based on SOME/IP [25]. This middleware includes a decentralized service-discovery

to dynamically find services in the system and a publish/subscribe scheme to publish and subscribe to events.

We assume that most system resources are available for performing the search and that the mapping approach be performed while the car is not actively in use. Once a stable mapping is found, the agents do not perform any action and, thus, do not require any additional resources. As backup mappings are already found and readily available, we follow our main hypothesis, such that when the failure occurs a safe state can be reached with a minimum amount of communication and computation. After a failover, the fail-operational behavior has to be re-established to ensure a safe travel as further failures could lead to potential hazards. A re-mapping of the applications has to be performed during a safe halt.

Our system to find feasible mappings at run-time consists mainly of agents and resource managers. Resource managers manage the resources associated with an ECU or switch. The resource managers take care of the allocation or reservation requests and assign the corresponding service intervals or slots. By assigning a slot a resource manager implicitly decides how the system will be degraded in a failure scenario and which task instances will loose resources. Therefore, changing the algorithm for assigning the slots can impact the degradation behaviour and success chance of finding a mapping. The responsibility of an agent is to find a suitable mapping for its task that meets all mandatory constraints by allocating and reserving resources at the resource manager or by moving to another ECU. Furthermore, each agent also verifies if an application-wide constraint such as the end-to-end application latency is still met. As partially a sequential mapping order is required to fulfill or verify these constraints, agents can communicate with each other to synchronize the mapping flow.

In the following we first provide an overview and formalize the constraints that have to be respected to consider a given mapping as a valid solution in Sect. 6.1. We then present a solution how the timing constraints are evaluated at run-time using our performance analysis in Sect. 6.2. Afterwards, we present the mapping process of an application using agents in Sect. 6.3. Here, we first go into detail in which order the agents find a mapping for their task to be able to meet all constraints. Afterwards, we present a code listing and describe how the search space is searched for a valid solution by the agents which solves all constraints using a backtracking algorithm. Furthermore, we describe the role of resource managers in Sect. 6.4 which are responsible for performing allocations and reservations by choosing the corresponding service intervals. We also present our allocation and reservations strategies that can be used by a resource manager which has a direct effect on how the system will be degraded. Last, we describe the recovery and reconfiguration process with the immediate failure reaction to ensure a safe fail-operational behavior in Sect. 6.5.

6.1 Constraints

The following constraints have to be respected to consider any found mapping as a valid solution.

C.1 The application-wide worst-case end-to-end application latency must meet the deadline δ:

$$\mathcal{L}(\alpha, \beta, \rho, \sigma) \leq \delta \tag{20}$$

C.2 An active task instance t_a and its corresponding passive task instance t_b may not be mapped onto the same ECU as this would contradict our fail-operational goal:

$$\alpha(t_a) \neq \beta(t_b). \tag{21}$$

C.3 The number of allocated service intervals $N_{SI,a}(e)$, reserved service intervals $N_{SI,r}(e)$ and service intervals with both allocation and reservation $N_{SI,ar}(e)$ of an ECU e must not exceed the number of maximum available service intervals SI_{max}:

$$N_{SI,a}(e) + N_{SI,r}(e) + N_{SI,ar}(e) \leq SI_{max} \tag{22}$$

C.4 The number of allocated slots $N_{SL,a}(l)$, reserved slots $N_{SL,r}(l)$ of a link l must not exceed the number of maximum available slots SL_{max}:

$$N_{SL,a}(l) + N_{SL,r}(l) \leq SL_{max} \tag{23}$$

C.5 Service intervals which are allocated by a critical task t must not be allocated or reserved by any other task:

$$\forall SI \in SI_a(t) \wedge \forall t' \in T : SI \notin SI_a(t') \wedge SI \notin SI_r(t') \tag{24}$$

C.6 Service intervals which are reserved by a critical task t must not be allocated or reserved by another critical task but may be allocated by a non-critical task:

$$\forall SI \in SI_r(t) \wedge \forall t' \in T_c : SI \notin SI_a(t') \wedge SI \notin SI_r(t') \tag{25}$$

C.7 Service intervals which are allocated by a non-critical task t must not be allocated by any other task but may be reserved by a critical task:

$$\forall SI \in SI_a(t) \wedge \forall t' \in T : SI \notin SI_a(t') \tag{26}$$

C.8 Slots which are allocated or reserved by a message m must not be allocated or reserved by another message:

$$\forall SL \in SL_a(m) \wedge \forall m' \in M : SL \notin SL_a(m') \wedge SL \notin SL_r(m') \tag{27}$$

$$\forall SL \in SL_r(m) \wedge \forall m' \in M : SL \notin SL_a(m') \wedge SL \notin SL_r(m') \tag{28}$$

Sect. 6.2 describes how solutions that respect Constraint C.1 can be found at run-time by splitting the validation problem up into smaller sub-problems which are solved by the agents. Constraint C.2 is respected by constraining the mapping order of the agents and by limiting the search space accordingly as described in Sect. 6.3. Constraints C.3 to C.8 are implicitly respected by the resource manager when assigning service intervals and slots to task and message instances as described in Sect. 6.4.

6.2 Run-time timing constraint solving

As described in Sect. 5.1 the critical path in the application graph has to be identified to verify whether Constraint C.1 can be met. Instead of verifying the timing constraint once all task instances are mapped, we continuously evaluate whether the application meets the timing constraint with the task instances mapped so far. This is done at run-time as the mapping of task instances and their predecessors is not known at design time. We split the bigger problem of verifying each single path into smaller sub-problems solved by the agents such that the constraint is continuously being verified and the search can be interrupted earlier if

the mapping process is running into a dead-end. As only the critical path is of interest, each agent calculates and stores the worst-case path latency to its task $\mathcal{L}(t, \alpha, \beta, \rho, \sigma)$ as

$$\mathcal{L}(t, \alpha, \beta, \rho, \sigma) = \max_{\forall t_p \in pred(t)} (\mathcal{L}(t_p, \alpha, \beta, \rho, \sigma) + CL(m_{t_p - t}, \rho, \sigma)) + TL(t, \alpha(t), \beta(t)). \quad (29)$$

Here, an agent requires only the worst-case path latencies $\mathcal{L}(t_p, \alpha, \beta, \rho, \sigma)$ of its predecessor tasks t_p and the corresponding worst-case communication latencies to its own task $CL(m_{t_p - t}, \rho, \sigma))$ together with the worst-case task latency $TL(t, \alpha(t), \beta(t))$. The maximum of all paths from preceding tasks t_p to this task t is then the worst-case path latency to it task $\mathcal{L}(t, \alpha, \beta, \rho, \sigma)$. Accordingly, instead of verifying Constraint C.1 once all tasks are mapped, each agent can verify the following constraint on its own:

$$\mathcal{L}(t, \alpha, \beta, \rho, \sigma) \leq \delta \quad (30)$$

This has the advantage if there are multiple sink tasks in an application graph that each agent can verify the constraint on its own. Furthermore, if the constraint can not be met by any of the tasks during the mapping process, the process can be stopped and a backtracking algorithm can be applied saving the agents from evaluating invalid solutions.

6.3 Agent

In our work each task instance is assigned to an agent which is responsible for finding a mapping by allocating and reserving resources at resource managers and by coordinating the mapping process with other agents. In the following we first present a pre-defined mapping order in which the agents of an application find a mapping. Afterwards, we describe the search space exploration of an agent using a code listing. Last, we present the backtracking approach which agents require to explore different solutions.

6.3.1 Mapping order

For the mapping process of the tasks onto the ECUs a predefined mapping flow is required to ensure that constraints are met and only valid mappings are generated. Theoretically, it would be preferable to map all tasks in parallel. However, there are limitations to the level of parallelism due to constraints and their verification at run-time. To adhere to Constraint C.2 a sequential mapping flow between active and passive task instances is required to avoid mapping both task instances onto the same ECU. Therefore, we define that active task instances have to be mapped before passive task instances. Second, to allocate and reserve resources for communication, the routing from a predecessor task to its successor has to be known. We decided that the succeeding tasks allocate and reserve the resources for all incoming messages. As a consequence, a task instance can only be mapped onto an ECU once all preceding task instances are mapped. This sequential mapping order is also required to perform the run-time constraint solving described in Sect. 6.2.

Figure 7 shows the mapping flow of an exemplary task instance graph consisting of four tasks. The mapping process starts with the active task instance of the source task $t_{0,a}$. All other task instances are initially waiting until they received a message from all their mapping predecessors. Once the active task instance has found a mapping it will inform its passive counterpart $t_{0,b}$ that it found a valid mapping. The passive task instance $t_{0,b}$ that has been

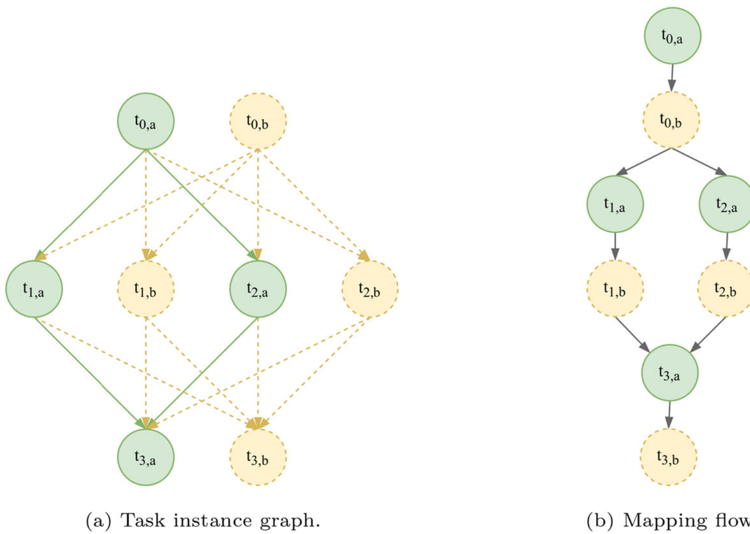


Fig. 7 An exemplary task instance graph with the corresponding mapping flow. Passive task instances are mapped after active task instances. Active task instances have to wait until the passive task instances of their predecessors are mapped. As the task instances of $t1$ and $t2$ do not have any dependencies on each other they can be mapped in parallel. However, the active task instance of $t3$ has to wait for both branches to finish to continue with the mapping process. (Color figure online)

waiting then finds a mapping itself. It ensures that the ECU to which the active task instance is mapped to is removed from the search space of the passive task instance. Afterwards, the passive task instance informs all active task instances of directly succeeding tasks. In the example $t_{0,b}$ informs $t_{1,a}$ and $t_{2,a}$ about a successful mapping. Active task instances might have to wait not only for one but multiple passive task instance of their preceding tasks depending on the dependencies in the application graph. In the figure $t_{3,a}$ has to wait for both $t_{1,b}$ and $t_{2,b}$ to finish. By contrast, neighbouring tasks with shared predecessors can search for a mapping in parallel as they have no direct dependency and open a new branch as it is the case for the task instances of $t1$ and $t2$ in the example. Any tasks instances belonging to different branches can also work in parallel. Therefore, the level of parallelism in the mapping search is bound by the number of branches or the width of the application graph. Tasks which have predecessors in multiple different branches reduce the level of parallelism and represent a bottleneck.

6.3.2 Agent mapping

Once it is the turn of an agent to find a mapping, it is its responsibility to evaluate the mapping options and ensure that resources are allocated and reserved accordingly. Listing 1 shows the pseudo-code for the mapping process of an agent. Each agent starts in the Wait() function (Line 1) where they wait until all predecessors send a signal that they found a mapping and transmit their worst-case latencies and mappings. Afterwards, the agent becomes active and is searching for a mapping. Here, at first the search space is set up consisting of all available ECUs in the system (Line 6). Then, in case a passive task instance t_b is being mapped, the ECU to which the corresponding active task instance t_a is mapped to is removed from the

search space $B(t_b)$ to adhere to Constraint C.2 (Lines 7 and 23):

$$B(t_b) = \{e \in E \mid e \neq \alpha(t_a)\} \tag{31}$$

To find the most suitable solution, the different options for the mapping of the task are evaluated (Lines 8 and 28). Here, the worst-case path latency $\mathcal{L}(t, e)$ for each available ECU $e \in E$ is calculated. Afterwards, the search space can be filtered for invalid solutions that do not adhere to Eq. 6.2 and, therefore, would violate Constraint C.1 (Lines 9 and 34). Using a heuristic the possible solutions are then sorted by the calculated by an increasing worst-case latency (Line 10). The agent then tries to allocate resources for the most fitting solution by sending allocation and reservation requests to resource managers (Lines 11 and 40). Here, service intervals have to be allocated or reserved for the executing tasks and slots allocated or reserved for all incoming messages on the links on the corresponding routing paths. The resource managers receive a request from an agent to allocate or reserve a certain amount of service intervals for CPU resources or slots for a link. The resource manager then executes the allocation or reservation and answers the agent whether the allocation or reservation could be performed successfully or not. The resource manager also implicitly decides how the system will be degraded by choosing the corresponding service intervals for the agent. However, this behavior is completely transparent to the agent and it does not know whether the activation of its passive task instance will lead to a degradation of another task and vice versa. Multiple strategies of the resource manager for allocating and reserving resources are described in Sect. 6.4. In case all resources could be successfully allocated and reserved a valid solution has been found such that the agent can update the binding for its task, move to the corresponding ECU and inform all succeeding agents about its success (Lines 18-21).

Listing 1 Code listing of our agent-based mapping approach.

```
1  Wait():
2      α, β, ℒ(t_p) ← WaitForPredecessors()
3      FindMapping(α, β, ℒ(t_p))
4
5  FindMapping(α, β):
6      E ← GetSearchSpace()
7      E ← RemoveActiveBindingFromSearchSpace(α ,E)
8      ℒ ← EvaluateSearchSpace(α, β, ℒ(t_p), E)
9      E ← FilterSearchSpace(ℒ, E)
10     E ← SortSearchSpaceByLatency(ℒ, E)
11     ϵ ← TrySolutions(E)
12
13     if ϵ == ∅
14         for t_p in pred(t):
15             t_p.Backtrack()
16         Wait()
17     else:
18         UpdateBindings(α, β, ϵ)
19         ℒ(t) ← ℒ(t, ϵ)
20         MoveToECU(ϵ)
21         InformSuccessors(α, β, ℒ(t))
22
23 RemoveActiveBindingFromSearchSpace(E):
24     if t ∈ T_b:
```

```
25          E.remove(α(t))
26      return E
27
28  EvaluateSearchSpace(α, β, E):
29      for ϵ ∈ E:
30          ρ, σ ← GetRoutings(α, β, ϵ)
31          ℒ(t, ϵ) = max∀t_p∈pred(t) (ℒ(t_p) + CL(m_{t_p−t}, ρ, σ)) + TL(t, ϵ)
32      return ℒ, E
33
34  FilterSearchSpace(ℒ, E):
35      for ϵ ∈ E:
36          if ℒ(t, ϵ) > δ:
37              E.remove(ϵ)
38      return E
39
40  TrySolutions(E):
41      while ϵ ∈ E and not ζ:
42          E.remove(ϵ)
43          ζ ← AllocateAndReserveResources(ϵ):
44
45      if not ζ:
46          return ∅
47      else:
48          return ϵ
49
50  Backtrack():
51      FreeResources()
52      InvalidateSuccessors()
53      FindMapping()
54
55  InvalidateSuccessors():
56      for t_s in succ(t):
57          t_s.Invalidate()
58
59  Invalidate():
60      FreeResources()
61      InvalidateSuccessors()
62      Wait()
```

6.3.3 Backtracking

In case that either all solutions in the search space have been filtered out or not sufficient resources could be allocated or reserved for any of the solutions, the mapping algorithm ran into a dead end. Here, a backtracking algorithm is performed. The agent informs its predecessor in the mapping flow that no solution could be found and then goes back into its initial waiting status (Lines 14–16 and 49). The informed agent then frees all resources it had previously allocated or reserved and the ECU to which it is currently mapped to is removed from the search space. Afterwards, this agent has to in turn inform all other succeeding agents

in the mapping flow that its mapping has been invalidated. This prohibits the succeeding agents in other branches from wasting time and unnecessarily occupying resources. Even if the other branches were to find a valid solution, the solution would be based on a mapping of their predecessor that is no longer valid. Furthermore, they are informed that they have to wait again for their predecessor in the mapping flow. After invalidating its successors the backtracking agent tries to find another valid solution (Line 53) and will then again inform its predecessors about the success. If there is no valid solution left or no other solution can be found, it will itself start the backtracking process of its own predecessors.

6.4 Resource manager

In the following we describe how an allocation and reservation request of an agent is performed by a resource manager. Furthermore, we introduce the three allocation and reservation strategies *Random*, *FreeFirst* and *FreeLast* and describe their respective advantages and disadvantages over each other. These three strategies are evaluated in detail in Sect. 7.

6.4.1 Resource allocation and reservation

The resource managers receive a request from an agent to allocate or reserve a certain amount of service intervals for CPU resources or slots for a link. It is the job of the corresponding resource manager to execute the allocation or reservation and to answer whether the allocation or reservation could be performed successfully or not. The resource manager decides during this process which exact service intervals or slots will be allocated or reserved. Here, it ensures that constraints C.3 to C.8 are met as it is programmed to only operate within these bounds. This allocation and reservation process also decides how the system will be degraded in a failure scenario. Section 5.2 describes in detail how a degradation would be performed on the level of service intervals. If a service interval is both reserved by a critical passive task instance and allocated by a non-critical task instance, the service interval is automatically assigned to the critical task instance once a failure occurs. In our system a critical agent can pass the information in which failure scenarios this activation should occur. The resource manager is then informed by a watchdog about any failures occuring in the system such that it can immediately react. This behavior is completely transparent to the agent and it does not know whether the activation of its passive task instance will lead to a degradation of another task.

6.4.2 Strategies

In the following we discuss the three strategies we developed to allocate and reserve resource slots. By default the resource managers use the *Random* strategy which chooses the service intervals assigned to an allocation or reservation request randomly. Changing this algorithm can impact the degradation behaviour and success chance of finding a mapping. In the following we propose two alternative strategies *FreeFirst* and *FreeLast* for assigning service intervals.

The *FreeFirst* strategy aims at minimizing the overlap between reservations and allocations by assigning service intervals first that have not been allocated or reserved. Only if no free service interval is available the algorithm will allocate or reserve other service intervals. The algorithm has the advantage of reducing the degradation effect as allocations and reservations overlap as little as possible. The downside is that in scenarios where resources are constrained,

the algorithm might lead to lower success rates of finding mappings as less free service intervals will be available. On the other hand, in more relaxed scenarios it uses all available resources to reduce the degradation effect with little effect on the success rate.

The *FreeLast* strategy aims at utilizing resources more efficiently by assigning service intervals first that already have been allocated or reserved, which maximizes the overlap between reservations and allocations. The advantage is that even in resource-constrained scenarios, the resources are used in an efficient way such that the success rate of finding mappings is increased. On the other hand, as the overlap between reservations and allocations is maximized, there will be a stronger degradation effect in case of a failure scenario. In scenarios where resources are less constrained, reservations and allocations will overlap even if additional free service intervals were available leading to avoidable degradation effects.

When choosing one of the two opposing strategies, a trade-off between degradation effect, success rate and resource-efficiency has to be made. In scenarios where degradation is desired or tolerated, the *FreeLast* strategy can lead to a lower resource utilization and higher success rates. In scenarios where many resources are available and degradation should be avoided as much as possible the *FreeFirst* strategy maximizes resource utilization to minimize degradation impact. We continue this discussion with our experimental results in Sect. 7, which gives further insights into these three strategies.

6.5 Recovery and reconfiguration

In a failure scenario with a critical application, an immediate failure reaction and failover to the backup solution is required to keep the application operational. Failures can be detected via watchdogs and heartbeats. In previous work [4], we have presented a formal analysis to derive the worst-case application failover time for distributed systems. Here, we analyzed the impact of failure detection and recovery times on the timing behavior of distributed applications. This analysis guarantees an upper bound on the time that it would take for an application to generate a new output after the failover of one or multiple task instances. Our current work can be used as a base for the analysis done in [4] as our system provides an upper bound on task execution and message transmission times.

In case an active task instance of a critical application is affected by the failure, the watchdogs notice a timeout and notify both the resource manager and the agent of the passive task instance. The resource manager immediately turns the reservation of the agent into an allocation and performs a degradation if required. The agent of the passive task instance then starts its task. As we use a service-oriented middleware with a publish/subscribe pattern, this task instance then has to subscribe to its predecessor task instances to receive the corresponding messages. In case the preceding task instances have been affected by the failure as well, they first have to advertise their service before succeeding task instances can subscribe to them. If a passive task instance of a critical application is affected by a failure no immediate failure reaction is required. In case a failover has been performed by a preceding task instance, a task instances also has to wait until the corresponding service is offered until it can re-subscribe to it.

If a task instance of a non-critical application is affected by a failure, no immediate reaction is required. In case a non-critical task instance is affected by a degradation the agent is automatically notified. The degraded non-critical task can then run in a degraded mode with less available resources or completely be shut down.

After the immediate failover, which ensures a safe fail-operational behavior of critical applications, a reconfiguration of the system can be performed. During this reconfiguration,

resources can be freed by shutting down task instances of failed non-critical applications. Furthermore, it is important to re-establish the fail-operational behavior of critical applications. Here, the agent of the source task invalidates the mapping of all its successors in the mapping order which in turn also inform all their own successors. Afterwards, the mapping process can be repeated while the car is in a safe state until a valid configuration is found.

7 Evaluation

We evaluate our performance analysis and our agent-based approach using our in-house developed simulation framework. The framework has been developed to simulate automotive hardware architectures and the execution and communication of the system software according to our system model in Sect. 3. On top of the simulation framework we implemented the agents, resource managers and strategies as described in Sect. 6. For the simulation framework we chose a process-based Discrete-Event Simulation (DES) architecture based on the SimPy framework [32]. The hardware architecture and system software are described in a specification file using the XML schema from the OpenDSE framework [33]. The simulation framework supports any kind of hardware architecture consisting of ECUs, switches, and links. To allow a dynamic behavior where tasks and agents are moving between ECUs at run-time we use a communication middleware based on the SOME/IP standard [25]. The middleware consists of a service discovery which allows to dynamically find services at run-time. Communication participants are either modelled as clients or services. Furthermore, the middleware supports remote-procedure calls and includes a publish/subscribe scheme. The framework also offers the possibility to simulate ECU failures by shutting down ECUs. ECU failures are detected via heartbeats that are periodically sent between all ECUs. Once a watchdog does not receive the heartbeat within a certain timeout interval it reports the corresponding ECU failure.

7.1 Setup

The hardware architecture that we use in our experiments and which is depicted in Fig. 8 consists of ten homogeneous ECUs which are connected in pairs to one switch. The switches are connected to each other in a ring architecture resulting in a maximum hop count of four between each ECU. We use shortest path routing based on Dijkstra's algorithm for routing the messages [27]. For our experiments we use applications which are synthetically generated by the OpenDSE framework using the TGFF algorithm [33, 34]. All applications consist of exactly ten tasks with a WCET of $W(t, \alpha(t)) = 2.5$ ms and the requirement to allocate or reserve five service intervals. We design the system such that exactly one ethernet frame with a maximum frame size of 1518 bytes can be sent in one time slot. Using ethernet links with a data rate of 1 Gbit/s we can calculate the transmission time over one link l as

$$t_{trans} = \frac{1518 \text{ byte}}{1 \text{ Gbit/s}} = 12.144\,\mu\text{s}. \tag{32}$$

Rounding this value, we design our slot interval as $\tau_{SL} = 12.5\,\mu$s. Thus, we can calculate the transmission time of a message over one ethernet link $CL_{trans}(m, l)$ as

$$CL_{trans}(m, l) = \tau_{SL} = 12.5\,\mu\text{s}. \tag{33}$$

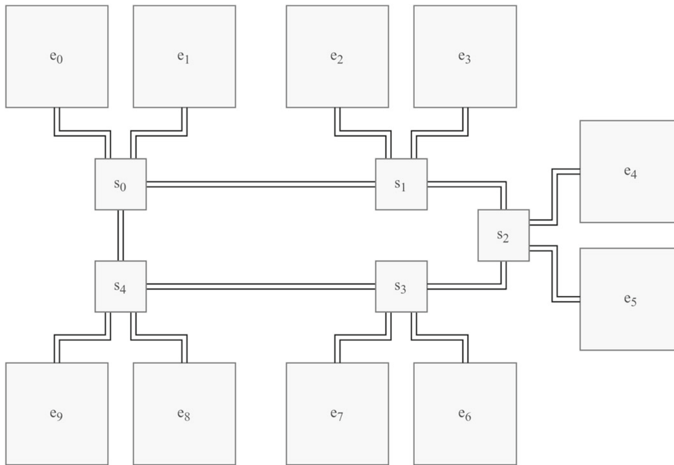


Fig. 8 The simulated hardware architecture used in our experiments. Ten homogeneous ECUs $e \in E$ are connected in pairs via ethernet links $l \in L$ to one switch $s \in S$ forming a ring architecture. Messages are routed using Dijkstra's algorithm [27]. In the worst case four hops are required to enable communication between two ECUs. Using ethernet links with a data rate of 1 Gbit/s, a maximum slot number of $SL_{max} = 1000$ and a slot interval of $\tau_{SL} = 12.5\,\mu$s, the worst-case communication latency for sending a message m over one link is $CL_(m, l) = 12.5$ ms

Choosing $SL_{max} = 1000$, we can calculate the worst-case communication latency $CL_(m, l)$ over one link according to Eq. 10 as

$$CL_(m, l) = SL_{max} \cdot \tau_{SL} = 12.5\,\text{ms}. \tag{34}$$

The resource managers on the ECUs are managing access to a schedule with $SI_{max} = 250$ service intervals and a service interval time of $\tau_{SI} = 0.5$ ms resulting in a total number of 2500 service intervals.

7.2 Experiments

We set the deadline δ for the worst-case end-to-end application latency to 1200 ms, creating scenarios with both tight and relaxed timing constraints for the randomly generated applications. For the experiments we chose 20 non-critical applications and deviated the number of critical applications N_c between 10 and 30 to construct scenarios where resources are constrained and scenarios where the resource situation is relaxed. Results present the average values of 500 runs per configuration. For each single run a new set of applications was synthetically generated. As the search process could potentially take a lot of time in cases with tight timing constraints or where resources are limited, we set an upper bound of 10,000 backtrack operations as a stopping criteria. Cases which hit this upper bound are considered as failed. We conducted the simulations on a server with an Intel Xeon Gold 6130 CPU consisting of 16 cores running at 2.1 GHz and 128GB of RAM. Each simulation run was assigned a single CPU and 8GB of RAM. Simulation runs finished within a few minutes in the majority of cases. In the worst case it took 76 min to finish the simulation.

In the following we present the mapping success rates of our constraint solving approach over a deviating number of critical applications. We compare these results to our graceful degradation approach from [2] where no timing constraints are considered and to an active

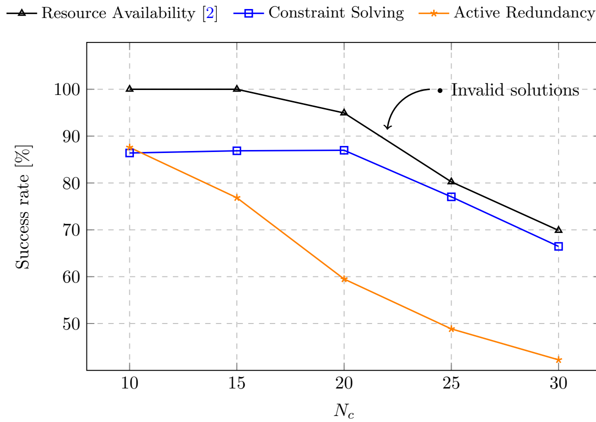


Fig. 9 Experimental results presenting the average mapping success rate of critical applications for deviating numbers of critical applications N_C. Using our constraint solving approach with graceful degradation (blue curve with square marks) the success rate is significantly higher than using active redundancy (orange curve with star marks). When not considering timing constraints and only taking resource availability (black curve with triangle marks) into account, as presented in [2], only invalid solutions would be found which is not an option. (Color figure online)

redundancy where no degradation is applied. Afterwards, we evaluate the success rate of our three allocation and reservation strategies from Sect. 6.4. Then we further analyze one scenario by presenting Cumulative Distribution Function (CDF) over the number of explorations performed for both the successful and failed cases. Furthermore, we present the number of free service intervals and the number of overlapping service intervals for all our three strategies and also compare these results to an active redundancy approach. Last, we summarize all findings from these experiments about the difference of our allocation and reservation strategies between each other and draw a conclusion about the importance of graceful degradation compared to state-of-the-art approaches.

7.2.1 Success rate—constraint solving versus state of the art

Figure 9 presents the average success rate of the application mapping processes of critical applications for a deviating number N_c of critical applications. We compare our constraint solving approach (blue curve with square marks) to our state-of-the-art approach in [2] where timing constraints are not considered and only resource availability is taken into account. Furthermore, we conducted experiments where constraint solving is applied but no degradation is allowed (orange curve with star marks) representing approaches with active redundancy or where passive redundancy is used but resources are allocated exclusively to one task. While the approach based only on resource availability (black curve with triangle marks) is able to map applications with a success rate of 100% in cases where the resource situation is relaxed the success rate steadily decreases with an increasing amount of critical applications down to a success rate of 69.7% at a number of $N_c = 30$ critical applications. Looking at our run-time constraint solving approach it can be observed that the success rate is lower even in scenarios with a relaxed resource situation as some timing constraints are too tight to find a valid solution. With an increasing number of critical applications N_c the resources become the more limiting factor such that the success rate decreases further but also comes closer to the resource availability curve. These results confirm that an approach based

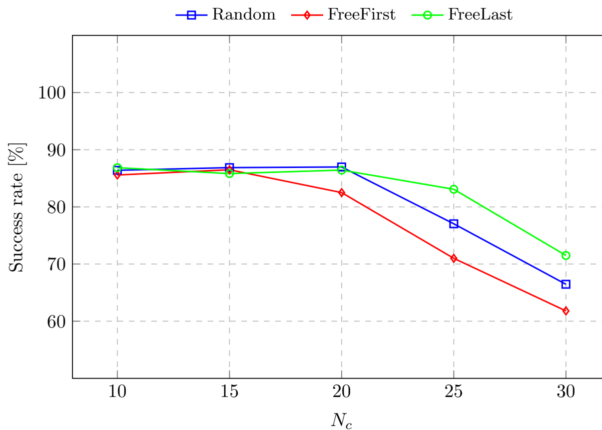


Fig. 10 Experimental results presenting the average mapping success rate of critical applications of our strategies for different numbers of critical applications N_c. The *FreeLast* heuristic (green curve with circle marks) leads to increased success rates compared to the random constraint solving (blue curve with square marks), while the *FreeFirst* heuristic (red curve with diamond marks) leads to lower success rates. With lower numbers of critical applications there is no impact of the heuristics on the success rate is visible as sufficient resources are available to allocate and reserve all required service intervals. With an increasing number of critical applications resources become more constraint and the advantages and disadvantages of the heuristics become more visible. (Color figure online)

only on resource availability would find mappings that would violate timing constraints in a high number of cases in this setup. The active redundancy approach has a similar success rate to our constraint solving approach with a number of $N_c = 10$ critical applications. However, with an increasing number of critical applications the success rate drops steadily with significant worse results than our constraint solving approach with degradation. It is clearly visible that graceful degradation can greatly increase the success rate in scenarios where resources become more limited. While it looks like the approach based on resource availability as a higher success rate, it mainly generates invalid solutions which is not an option.

7.2.2 Success rate—strategies

We also evaluate our strategies for allocating and reserving resources presented in Sect. 6.4. Figure 10 presents the success rates of critical applications of the default *Random* strategy (blue curve with square marks) and our two heuristics *FreeFirst* (green curve with circle marks) and *FreeLast* (red curve with diamond marks). For non-resource constrained cases with 10 and 15 critical applications, the values of all three algorithms are close to each other. Afterwards, in scenarios where resources become more limited with an increasing number of critical applications the success rate decreases for all three strategies. However, the three curves diverge from each other, with the *FreeLast* strategy having the highest success rates and the *FreeFirst* strategy leading to the lowest success rates. In resource relaxed scenarios the *FreeLast* heuristic is not advantageous as there are sufficient resources available to reserve and to allocate for both the *Random* and the *FreeFirst* algorithms. In more resource-constrained scenarios the advantage of the *FreeLast* strategy regarding the more efficient use of resources becomes visible, while the *FreeFirst* heuristic is leading to sub-optimal results. We discuss in Sect. 7.2.4 that there can be scenarios in which the *FreeFirst* strategy can have advantages.

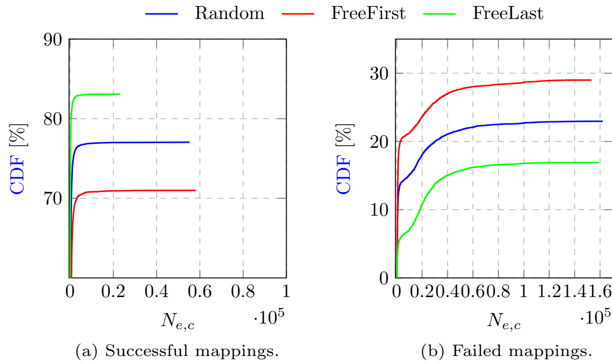
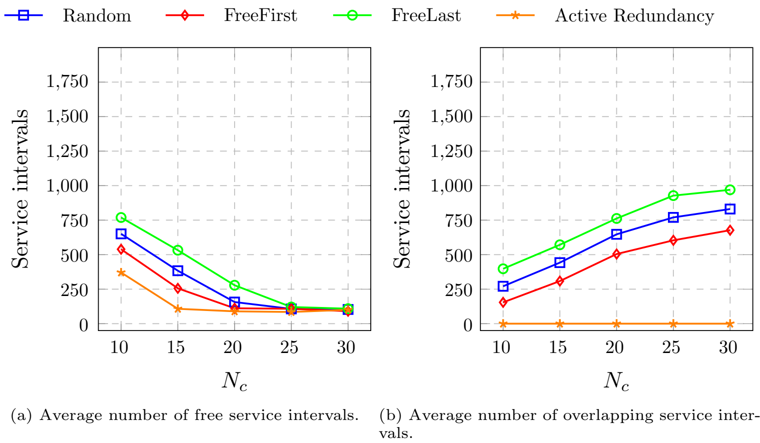


Fig. 11 CDFs of the number of total number of epxlorations of the critical applications $N_{e,c}$ in the scenario with $N_c = 25$ critical applications for both the successful cases (**a**) and the failed cases (**b**). Most applications that successfully found a mapping require clearly less than 500 explorations, with only a few exceptions requiring more explorations. Applications that could not find a mapping within the upper limit of 10,000 backtracking operations often required more than 500 explorations. (Color figure online)

7.2.3 Cumulative distribution function

To allow a more fine-grained analysis of these three strategies we present the CDFs of the number of explorations $N_{e,c}$ tested per critical application for the scenario with $N_c = 25$ critical applications split up into successful and failed mapping cases in Fig. 11. The number of explorations $N_{e,c}$ defines how many times in total the agents of a critical applications have explored a mapping by sending allocation and reservation requests for a potential solution. In a best-case scenario every agent successfully finds a mapping on the first try. In the worst-case scenario mappings are tested until our stopping criteria of 10,000 backtrack operations is reached. It can be observed that in the case of successful mappings almost all mappings are found with less than 500 allocation or reservation requests with only a few outliers. For applications that could not find a mapping within the upper limit of 10,000 backtrack operations many could confirm within 500 explorations that no mapping can be found. This implies that applications started backtracking relatively early in the search process as tasks which are further up in the application graph could not find a mapping due to limited resources such that it can be confirmed relatively quickly that no mapping can be found. However, for many applications it frequently took more than 500 explorations up to around 16,000 explorations. In these cases many task instances were already mapped until running into a shortage of resources or a violation of a timing constraint such that many explorations would have to be performed to confirm that no solution exists. The curves of the *FreeFirst* and the *FreeLast* strategies appear to have a similar trend as the curve of the *Random* strategy with a positive or negative offset. In the case of failed mappings this means for the *FreeFirst* strategy that there are more cases which are counted as failed after relatively few explorations implying that mappings could not be found due to resource constraints. On the other hand, in the case of the *FreeLast* strategy, this implies that more applications could find a mapping compared to the other two strategies as applications did not run as often into resource constraints. Therefore, these results further confirm the findings that the *FreeLast* strategy has an increased success rate due to utilizing service intervals more efficiently, while the *FreeFirst* strategy runs into resource constraints more often.



(a) Average number of free service intervals. (b) Average number of overlapping service intervals.

Fig. 12 Experimental results of the average numbers of free and overlapping service intervals with a total number of 2500 service intervals distributed over ten ECUs. The results are consistent with success rates from Fig. 10. A higher number of free service intervals is better as this allows to map more applications and increases the success rate. A higher number of overlapping service intervals directly increases the number of free service intervals and, therefore, increases the success rate. The *FreeFirst* heuristic occupies more service intervals than the default service intervals distribution algorithm, while the *FreeLast* heuristic occupies less service intervals leading to higher success rates. On the other hand, the *FreeLast* heuristic has a higher number of overlapping allocations and reservations which leads to a higher degradation impact, while the number of overlapping service intervals is lower for the *FreeFirst* heuristic. The active redundancy approach requires the most service intervals and does not result in any overlapping service intervals. (Color figure online)

7.2.4 Service interval utilization

Figure 12presents the number of free service intervals and the number of overlapping service intervals of the experiments with a deviating number of critical applications N_c. It can be observed that the curves are running almost parallel to each other with an offset until a minimum close to zero is hit. In most cases some service intervals remain unoccupied although not all applications can be mapped as the remaining service intervals are distributed over multiple ECUs and might only be a fraction of the service intervals required to map an application. The *FreeLast* strategy has the most free service intervals for the scenarios with a lower number of critical applications, while the *FreeFirst* algorithm leads to lower number of free service intervals compared to the *Random* strategy. The active redundancy approach leads to the lowest number of freely available service intervals. Comparing Figs. 12a to 9 and 10 we can see that the success rate always starts to decrease when the curve is hitting a low in the number of freely available service intervals. This observation confirms our previous findings that the number of available resources is limiting the success rate.

However, this advantage is bought with a likely more reduced functionality of non-critical applications after a failover scenario. In the case of active redundancy no service intervals are overlapping as no degradation is performed at all. The other three strategies lead to increasingly more overlapping service intervals with an increasing number of critical applications with the *FreeLast* resulting in most overlapping service intervals and the *FreeFirst* strategy resulting in the lowest number of overlapping service intervals. The more service intervals are overlapping the more likely a non-critical application will be degraded if the corresponding service interval is required by a passive task instance that is performing a failover. The number of overlapping service intervals also provides an explanation for the difference in free service

intervals between the three strategies. The *FreeLast* strategy overlaps service intervals whenever possible, keeping a maximum of free service intervals available which can be used to map further applications. The *FreeFirst* strategy instead avoids overlapping service intervals as much as possible leading to less freely available service intervals and thus limiting the amount of applications that can be mapped. This means at the same time that the *FreeLast* strategy is more likely resulting in a degradation in a failover scenario while the *FreeFirst* minimizes this risk. In scenarios with constrained resources, the *FreeLast* strategy can use the resources more efficiently by overlapping service intervals, resulting in higher success rates of finding mappings but also leading to significantly higher degradation effects on non-critical applications. In scenarios with relaxed resource situations the *FreeLast* heuristic does not make use of the plentiful available resources and might result in degradation scenarios that could be avoided when using all resources. On the other hand, the *FreeFirst* heuristic has the disadvantage of having lower success rate in resource-constrained scenarios as overlapping service intervals is avoided as far as possible. In more relaxed situations it uses all available resources to avoid degradation as much as possible leading to more optimal solutions.

7.2.5 Summary

Summarizing our observations and findings from our experimental results we have shown that it is necessary to use our constraint solving approach as approaches based only on resource availability would result in solutions violating timing constraints. Additionally, our assumptions about advantages and disadvantages of the three allocation and reservation strategies *Random*, *FreeFirst* and *FreeLast* from Sect. 6.4 have been confirmed. All results together provide a clear picture that the *FreeLast* strategy results in higher success rates by overlapping service intervals as far as possible and, thus, leaving more free service intervals to map other applications. While this strategy leads to increased success rates in resource-constrained scenarios, there is an increased degradation effect that could be avoided in scenarios with a relaxed resource situation. The *FreeFirst* strategy leads to lower success rates by avoiding to overlap service intervals as far as possible leading to less freely available service intervals and, thus, less applications that can be mapped. While this strategy reduces the degradation effect in scenarios with plentiful resources, it results in significantly lower success rates in resource-constrained scenarios compared to the other two strategies. Overall, the *FreeLast* maximizes the graceful degradation effect while the *FreeFirst* strategy minimizes it with all accompanying advantages and disadvantages for both.

Most importantly, our experiments have confirmed that graceful degradation strongly increases the success rate of finding a mapping for critical applications compared to an active or passive redundancy approach without degradation in resource-constrained scenarios. Wen using the *FreeLast* strategy, more than double the number of critical applications can fit onto the system architecture in our setup compared to an active redundancy approach. Summarized, graceful degradation can be a powerful methodology which uses resources more efficiently than common redundancy approaches and which can strongly increase the number of applications that can be mapped onto the same system architecture while providing the same fail-operational capabilities. However, when designing a system it has to be considered that more non-critical functionality is lost due to degradation in a failover scenario. Here, a trade-off between degradation impact, mapping success rate and resource availability has to be carefully evaluated.

8 Limitations

Proving whether a valid mapping and schedule solution exists or not is an NP-complete problem which would take exponential time in the worst case [10, 21, 35]. In our approach we assume that the problem is solved on a few powerful ECUs interconnected by high-speed Ethernet and that most of the resources are available for the mapping process. The mapping process itself is not performed during a time critical phase and only while the car is safely parked such that longer search times might be acceptable. Our simulation times on a single CPU with 8 GB of RAM typically ranged from a few minutes up to 76 min. However, to address scalability issues and in-car resource limitations of our approach there is interesting research on hybrid mapping methods such as presented in [35]. Here, meta-heuristic optimization approaches are used to find multiple pareto-optimal solutions at design time. Run-time backtracking approaches then perform constraint solving. In [35] the backtracking approach finds a solution for a comparable problem size within a few milliseconds. However, it has to be assumed that communication within a NoC architecture is faster than on a system level. Future work could include a hybrid mapping approach with the pre-computation of possible mappings at design time such that new mappings after a failover are found faster.

A known disadvantage of the decentralized resource control is the amount of messages that are required for resource monitoring [10]. In our case the messages are required for allocating and reserving resources. Therefore, the search time is highly dependent on the amount of messages sent. Here, in-car networks could pose a bottleneck during the mapping search. For an in-car implementation and the given problem size we would assume that in the worst case 18 messages (1 for the CPU, 8 for the links, two-way) are sent per exploration over 4 hops in the worst case. With a transmission time of $CL_{trans}(m, l) = 12.5\ \mu$s and the maximum number of explorations $N_{e,c} = 16000$ (see Figure 11) measured, the time taken to send all monitoring messages of one application would equate to $\tau = 18 * 4 * 12,5\ \mu s * 16.000 = 14.4$ s. Multiplying this result with the number of applications $N = 50$ would result in a total time of $\tau_{tot} = 720$ s spent on the transmission of messages. For our given problem size we would assume the transmission times as acceptable since these are pessimistic estimates. However, high-speed in-car connections are a requirement to solve the mapping problem with a decentralized control in a reasonable time frame.

Another limitation of our approach is that after a failover has been performed, no safe operation of the critical applications can be guaranteed as another critical failure could lead to potentially hazardous situations. Here, the car has to come to a stop and start a re-mapping process to re-establish the fail-operational behavior. Adding another layer of redundancy could potentially resolve this issue but would increase the problem complexity greatly as the communication between all redundant instances would need to be ensured. Hybrid mapping approaches could reduce the time spent on finding a new mapping greatly. Additionally, there is research on performing real-time task migration [28]. Using such an approach a new mapping could be applied while the car is actively driving. If this can be achieved in a safe way and within an application's operation conditions, a reconfiguration could be performed without having to stop the car.

9 Conclusion

In this paper, we have presented an agent-based approach which enables, for the first time, graceful degradation for real-time automotive applications. Our approach guarantees predictability for end-to-end timing constraints and enables a graceful system degradation while ensuring fail-operational requirements of critical applications. The advantage is that resources can be utilized more efficiently in mixed critical systems if a reduced functionality of non-critical applications after a failover can be accepted. To enable this behavior we have first introduced state-of-the-art predictable timing analysis for composable scheduling and adapted it to our system model. Then, we extended this predictable timing analysis for fail-operational systems to include backup solutions in the analysis such that it can be ensured that there is always a solution available that fulfills the end-to-end application latency constraint. Furthermore, we introduced our composable scheduling of gracefully degrading systems which allows critical backup solutions to reserve service intervals which are allocated by non-critical applications. In a failure scenario, once a critical backup solution has to be started, it can take over the resources such that the system is being degraded in an intended way. The advantage of using our passive backup solution compared to active redundancy is that almost no overhead is added in terms of required computational power. We also presented our agent-based approach which uses our run-time constraint solving approach to find solutions that meet resource and timing constraints. Here, a pre-defined mapping order of the tasks is required to ensure that constraints can be met. In case the mapping search is running into a dead end, backtracking is applied to explore other possible solutions. Resource managers receive requests from agents to allocate and reserve resources and decide which service intervals will be assigned. Here, we introduced three new strategies *Random*, *FreeLast* and *FreeFirst*, for the assignment of the resources which heavily influence how the system will be degraded. We performed multiple experiments on our simulation platform to evaluate our approach. We have shown that it is necessary to use our constraint solving approach as approaches based only on resource availability would result in solutions violating timing constraints. Compared to state-of-the art approaches such as active redundancy, our graceful degradation approach could fit double the amount of critical applications on the same platform before influencing the mapping success rate. The experiments proved that the *FreeLast* strategy further enhances the effect of graceful degradation effect by overlapping as much service intervals as possible, while the *FreeFirst* strategy reduces the graceful degradation by avoiding any overlapping service intervals as far as possible. Our experiments have shown that graceful degradation can greatly increase the success rate in scenarios where resources are limited if the risk of loosing non-critical functionality in a failure scenario is acceptable. In summary, by enabling graceful degradation for real-time applications with our predictable timing analysis and agent-based approach, resources can be utilized more flexibly and efficiently while guaranteeing a safe and dynamic behavior of automotive systems.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Saidi S, Steinhorst S, Hamann A, Ziegenbein D, Wolf M (2018) Future automotive systems design: research challenges and opportunities: special session. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS)
2. Weiss P, Weichslgartner A, Reimann F, Steinhorst S (2020) Fail-operational automotive software design using agent-based graceful degradation. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 1169–1174. https://doi.org/10.23919/DATE48585.2020.9116322
3. Akesson B, Molnos A, Hansson A, Angelo JA, Goossens K (2011) Composability and Predictability for independent application development, verification, and execution, pp 25–56. https://doi.org/10.1007/978-1-4419-6460-1_2
4. Weiss P, Elsabbahy S, Weichslgartner A, Steinhorst S (2021) Worst-case failover timing analysis of distributed fail-operational automotive applications. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 1294–1299. https://doi.org/10.23919/DATE51398.2021.9473950
5. Kohn A, Käßmeyer M, Schneider R, Roger A, Stellwag C, Herkersdorf A (2015) Fail-operational in safety-related automotive multi-core systems. In: 10th IEEE international symposium on industrial embedded systems (SIES), pp 1–4. https://doi.org/10.1109/SIES.2015.7185051
6. Baleani M, Ferrari A, Mangeruca L, Sangiovanni-Vincentelli A, Peri M, Pezzini S (2003) Fault-tolerant platforms for automotive safety-critical applications. In: Proceedings of the 2003 international conference on compilers, architecture and synthesis for embedded systems. CASES '03, pp 170–177. https://doi.org/10.1145/951710.951734
7. Bak S, Chivukula DK, Adekunle O, Sun M, Caccamo M, Sha L (2009) The system-level simplex architecture for improved real-time embedded system safety. In: 15th IEEE real-time and embedded technology and applications symposium, pp 99–107. https://doi.org/10.1109/RTAS.2009.20
8. Oszwald F, Obergfell P, Traub M, Becker J (2019) Reliable fail-operational automotive e/e-architectures by dynamic redundancy and reconfiguration. In: 2019 32nd IEEE international system-on-chip conference (SOCC), pp 203–208. https://doi.org/10.1109/SOCC46988.2019.1570547977
9. Smirnov F, Reimann F, Teich J, Han Z, Glaß M (2018) Automatic optimization of redundant message routings in automotive networks. In: Proceedings of the 21st international workshop on software and compilers for embedded systems, pp 90–99. https://doi.org/10.1145/3207719.3207725
10. Weichslgartner A, Wildermann S, Teich J (2011) Dynamic decentralized mapping of tree-structured applications on NoC architectures. In: Proceedings of the fifth ACM/IEEE international symposium, pp 201–208. https://doi.org/10.1145/1999946.1999979
11. Faruque M, Krist R, Henkel J (2008) Adam: run-time agent-based distributed application mapping for on-chip communication. In: Proceedings of the 45th annual design automation conference, pp 760–765. https://doi.org/10.1145/1391469.1391664
12. de Souza Carvalho EL, Calazans NLV, Moraes FG (2010) Dynamic task mapping for MPSoCs. IEEE Des Test 27(5):26–35. https://doi.org/10.1109/MDT.2010.106
13. Becker K, Voss S (2015) Analyzing graceful degradation for mixed critical fault-tolerant real-time systems. In: 18th international symposium on real-time distributed computing (ISORC), pp 110–118. https://doi.org/10.1109/ISORC.2015.10
14. Glaß M, Lukasiewycz M, Haubelt C, Teich J (2009) Incorporating graceful degradation into embedded system design. In: Proceedings of the conference on design, automation and test in Europe, pp 320–323. https://doi.org/10.1109/DATE.2009.5090681
15. Shelton CP, Koopman P, Nace W (2003) A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems. In: Proceedings of the 8th International workshop on object-oriented real-time dependable systems (WORDS), pp 156–163. https://doi.org/10.1109/WORDS.2003.1218078
16. Herlihy MP, Wing JM (1991) Specifying graceful degradation. IEEE Trans Parallel Distrib Syst 2(1):93–104. https://doi.org/10.1109/71.80192
17. Weichslgartner A, Wildermann S, Gangadharan D, Glaß M, Teich J (2018) A design-time/run-time application mapping methodology for predictable execution time in MPSOCS. ACM Trans Embed Comput Syst. https://doi.org/10.1145/3274665
18. Guo Z, Yang K, Vaidhun S, Arefin S, Das SK, Xiong H (2018) Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate. In: 2018 IEEE real-time systems symposium (RTSS), pp 373–383. https://doi.org/10.1109/RTSS.2018.00052
19. Kim J, Bhatia G, Rajkumar R, Jochim M (2012) Safer: system-level architecture for failure evasion in real-time applications. In: 2012 IEEE 33rd real-time systems symposium, pp 227–236. https://doi.org/10.1109/RTSS.2012.74

20. Pourmohseni B, Wildermann S, Glaß M, Teich J (2017) Predictable run-time mapping reconfiguration for real-time applications on many-core systems. In: Proceedings of the 25th international conference on real-time networks and systems, pp 148–157. https://doi.org/10.1145/3139258.3139278
21. Pourmohseni B, Glaß M, Henkel J, Khdr H, Rapp M, Richthammer V, Schwarzer T, Smirnov F, Spieck J, Teich J et al (2020) Hybrid application mapping for composable many-core systems: overview and future perspective. J Low Power Electron Appl. https://doi.org/10.3390/jlpea10040038
22. WikiChip: Tesla FSD computer. https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip Accessed 16 Aug 2022
23. Lunt M (2016) E/E-architecture in a connected world. https://www.asam.net/index.php?eID=dumpFile&t=f&f=798&token=148b5052945a466cacfe8f31c44eb22509d5aad1 Accessed 16 Aug 2022
24. Bosch: vehicle-centralized, zone-oriented E/E architecture with vehicle computers. https://www.bosch-mobility-solutions.com/en/mobility-topics/ee-architecture/ Accessed 16 Aug 2022
25. Scalable service-oriented MiddlewarE over IP (SOME/IP) (2021) http://some-ip.com/
26. International Organization for Standardization: ISO 26262 (2011) Road vehicles—functional safety—part 1–9, 1st edn. International Organization for Standardization
27. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numer Math 1(1):269–271. https://doi.org/10.1007/BF01386390
28. Pourmohseni B, Smirnov F, Wildermann S, Teich J (2020) Real-time task migration for dynamic resource management in many-core systems. In: Workshop on next generation real-time embedded systems (NG-RES 2020). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/OASIcs.NG-RES.2020.5
29. Frese T, Leonhardt T, Hatebur D, Côté I, Aryus H-J, Heisel M (2020) Fault tolerance time interval: how to define and handle. In: Neue dimensionen der mobilität: technische und betriebswirtschaftliche aspekte, pp 559–567. https://doi.org/10.1007/978-3-658-29746-6_45
30. Weiss P, Daporta E, Weichslgartner A, Steinhorst S (2021) Checkpointing period optimization of distributed fail-operational automotive applications. In: 2021 24th Euromicro conference on digital system design (DSD), pp 389–395. https://doi.org/10.1109/DSD53832.2021.00066
31. Heisswolf J, König R, Kupper M, Becker J (2013) Providing multiple hard latency and throughput guarantees for packet switching networks on chip. Comput Electr Eng 39(8):2603–2622. https://doi.org/10.1016/j.compeleceng.2013.06.005
32. SimPy T (2021) SimPy discrete event simulation library for Python, Version 4.0.1. https://simpy.readthedocs.io
33. Reimann F, Lukasiewycz M, Glaß M, Smirnov F (2021) OpenDSE—Open design space exploration framework. http://opendse.sourceforge.net/
34. Dick RP, Rhodes DL, Wolf W (1998) Tgff: task graphs for free. In: Proceedings of the sixth international workshop on hardware/software codesign. (CODES/CASHE'98), pp 97–101
35. Schwarzer T, Roloff S, Richthammer V, Khaldi R, Wildermann S, Glaß M, Teich J (2018) On the complexity of mapping feasibility in many-core architectures. In: 2018 IEEE 12th International symposium on embedded multicore/many-core systems-on-chip (MCSoC), pp 176–183. https://doi.org/10.1109/MCSoC2018.2018.00038

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.