Worst-Case Failover Timing Analysis of Distributed Fail-Operational Automotive Applications

Philipp Weiss*, Sherif Elsabbahy*, Andreas Weichslgartner† and Sebastian Steinhorst*
*Technical University of Munich, Germany; firstname.lastname@tum.de
†AUDI AG, Germany; andreas.weichslgartner@audi.de

***Abstract*—Enabling fail-operational behavior of safety-critical software is essential to achieve autonomous driving. At the same time, automotive vendors have to regularly deliver over-the-air software updates. Here, the challenge is to enable a flexible and dynamic system behavior while offering, at the same time, a predictable and deterministic behavior of time-critical software. Thus, it is necessary to verify that timing constraints can be met even during failover scenarios. For this purpose, we present a formal analysis to derive the worst-case application failover time. Without such an automated worst-case failover timing analysis, it would not be possible to enable a dynamic behavior of safety-critical software within safe bounds. We support our formal analysis by conducting experiments on a hardware platform using a distributed fail-operational neural network. Our randomly generated worst-case results are as close as 6.0% below our analytically derived exact bound. Overall, our presented worst-case failover timing analysis allows to conduct an automated analysis at run-time to verify that the system operates within the bounds of the failover timing constraint such that a dynamic and safe behavior of autonomous systems can be ensured.**

I. Introduction

With the rapid development of new functionalities to achieve autonomous driving the automotive industry sees itself confronted with increased safety requirements. Ensuring a fail-operational behavior is critical to enable autonomous driving as there will be no fallback solution without a driver in a failure scenario. After a critical failure the system is unable to react until the recovery is completed. Safety-critical applications have to fulfill strict timing requirements that also have to be met during such a failover.

To deal with increased complexity, electronic architectures and software systems are currently undergoing major changes. Instead of adding new electronic control units (ECU) for each new functionality, software is being integrated on more powerful central computers. Furthermore, new customer demands to regularly deliver the latest functionality requires automotive vendors to offer over-the-air software updates. One of the challenges with regular updates and customized software systems is to find a suitable and optimized mapping for all applications [1].

As a consequence, a dynamic resource management is required, where the mapping problem is solved at run-time as part of a software platform. A dynamic resource management allows both to integrate new applications at run-time and to repurpose resources to react to critical failures. With a dynamic resource management, safety-critical applications can be restarted after the malfunction of its electronic control unit

With the support of the Technische Universität München – Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement n° 291763.

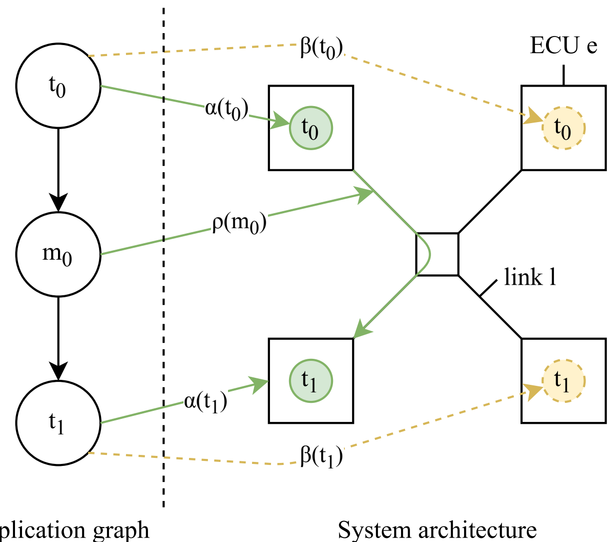


Fig. 1: Representation of our system model with an example application and system architecture. The green arrows indicate the active bindings of tasks t_0 and t_1 and the routing of message m_0, while the dashed yellow arrows indicate the passive task bindings.

on a still functional ECU. Here, graceful degradation can be used by shutting down non-critical applications on this ECU to free sufficient resources for the restarting safety-critical application. The advantage is that instead of adding costly additional hardware resources, the existing resources can be repurposed. [2]

The challenges for such a dynamic system are mainly to achieve a predictable system behavior. In this context, it is important that application requirements on the timing behavior can be verified and that a failover within the Fault Tolerant Time Interval (FTTI) can be guaranteed. [3]

In this paper, we analyze the impact of a failover on the timing behavior of distributed fail-operational applications and derive an upper bound for the worst-case failover time. The work presented in this paper can be used to evaluate and verify the worst-case failover timing behavior of fail-operational distributed applications. Instead of performing time-consuming experiments, our formal analysis can be used to evaluate whether a mapping would meet the failover timing constraints or not such that an evaluation at run-time is possible. Therefore, we make the following contributions:

- We analyze related work in the fields of timing analysis and fail-operational automotive systems in Section II. There is no work yet that has analyzed the timing effects

of a failover on distributed fail-operational applications.

- Based on the application and failover model presented in Section III, we introduce a formula to derive the application failover time in Section IV. Here, we analyze worst case scenarios to derive an upper bound for the failover time. Our upper bound can be used to achieve a predictable fail-over behavior and to verify application requirements on failover constraints.
- We support our formal analysis by conducting failover experiments on our demonstrator platform using a fail-operational distributed neural network in Section V.

II. Related Work

In our work we focus on executing a worst-case failover timing analysis to enable dynamic and safe system behaviour. Despite the fact that analyzing mapping strategies go beyond the scope of this work, our work is relevant to mapping in the sense that its results could be used to evaluate mappings at run-time. In addition, we use an agent-based mapping approach to conduct our failover experiments. Hence, the relevant related literature is mainly that concerned with timing analysis as well as fail-operational dynamic mapping approaches.

In [4] a fail-operational function-specific E/E-architecture for brake and steering control is introduced that supports dynamic configuration. A number of simulations are executed to derive the requirements for failure detection time and fault reaction time, against which the presented architecture is tested. The authors further developed their approach in [5] by adding a hardware extension to prevent state-loss that relies on CAN messages to communicate its state and they integrate the architecture into a service-oriented architecture. This approach is similar to the one we present since our architecture relies on communication through service-oriented middleware. However, we do not rely on extra hardware and we use periodic Ethernet heartbeat messages to detect the state of operation of the hardware devices.

In [6], the authors target distributed real-time embedded systems and aim to provide an automated design process for software reconfiguration. To approach this task, they define a number of "mode structures" characterized by a set of structured component types and each one comprises a number of configuration instances. The transitions from one instance to another are triggered by events related to system constraints or variations to the infrastructure. Our work goes an extra step further beyond the scope of system reconfiguration and covers the analysis for the timing behavior of system recovery in reaction to failures.

The related literature covers also topics such as graceful degradation in embedded systems, which requires a dynamic reconfiguration of the system and enables a subset of the tasks to continue running according to predefined priorities. In [7], a 2-step methodology is introduced, where the first step focuses on optimizing the design phase and resource allocation through degradation-aware reliability analysis. The second phase involves optimizing the system behavior online through a proposed algorithm relying on the data structures generated in the first phase. Our work however does not rely on design-time configuration. Instead, we propose a recovery scheme where the redundant software copies are assigned to hardware units in runtime according to the available resources.

We draw on the work presented in [2] and conduct our work on a similar platform. In this work a simulation framework for fail-operational systems is presented, in which the computational resources are dedicated to the higher priority tasks and thus improving the reliability of the system. The authors in [8] propose a predictable task migration mechanism by implementing a migration timing analysis and a feasibility check for real-time applications. Here, the goal is to enable a dynamic resource management to adapt the mapping of tasks at run-time. In our paper, we do not migrate the tasks to optimize the mapping, but assume that the tasks are already deployed redundantly such that a direct failover is possible.

None of the work concerned with discussing the timing properties of safety critical embedded systems has yet covered the topic of failover timing analysis. Instead, a maximum threshold for switching to the redundant copies of their applications is proposed such as presented in [9] or for control algorithms that guarantee the stability of embedded control systems on unreliable hardware platforms in a limited amount of time [10].

III. System Model

A. Application Model

Our system architecture consists of a set of ECUs $e \in E$ which are interconnected via switches and Ethernet links $l \in L$. Our system software consists of a set of safety-critical applications $a \in A$, where each application a can be modeled by an acyclic, directed, bipartite application graph. Each application a consists of a set of tasks $t \in T$ and a set of messages $m \in M$. For our analysis we assume that each node has at maximum one predecessor and one successor such that the application graph builds a task chain.

We assume a valid binding $\alpha : T \rightarrow E$ is given which assigns each active task instance $t \in T$ to an ECU $\alpha(t) \in E$. For our safety-critical applications a we assume that a redundant passive task instance is available in the system. Here, we assume a binding $\beta : T \rightarrow E$ is given which assigns each passive task instance $t \in T$ to an ECU $\beta(t) \in E$.

Furthermore, we assume that a routing $\rho : M \rightarrow 2^L$ is given which assigns each message $m \in M$ to a set of connected links $L' \subseteq L$ that establish a route $\rho(m)$. As the routing will also change after a failover, up to three passive routes are required of which one will become activated depending on which tasks are affected by the failover. Figure 1 depicts the application and system model with an exemplary application consisting of a task chain with two tasks.

We use the notation $\mathcal{L}_i(a)$ to define the end-to-end application latency of a single iteration i. Using composable task and communication scheduling, the interference between tasks and messages can be bounded [11], [12], such that a worst-case $\mathcal{L}_{wc}(a)$ and best-case application latency $\mathcal{L}_{bc}(a)$ can be calculated even at run-time. Similar, we define $\mathcal{L}_{bc}(t)$ and $\mathcal{L}_{wc}(t)$ as the best-case and worst-case latency from the application start until task t has finished execution. We assume that the application is periodically executed with the period P_a and that the application might operate in a pipeline such that P_a can be smaller than the worst-case application latency $\mathcal{L}_{wc}(a)$.

B. Failover Model

We define a failure $f \in F$ with $F \subseteq E$, where f identifies the failed ECU. To describe the bindings after the j-th failure, we use the notation $\alpha_j(t)$ and $\beta_j(t)$, with $\alpha_0(t)$ and $\beta_0(t)$ being the initial bindings. A failover is required once an ECU e fails to which at least one active task instance of a safety-critical application a has a binding: $\exists t \in T : \alpha_j(t) = e$. In a failover scenario we assume that affected task instances are lost and that tasks are restarted using the passive task instances such that the active binding of the affected task instances is changed to the former passive task binding: $\alpha_{j+1}(t) = \beta_j(t)$. Furthermore, a new binding for the passive task instance $\beta_{j+1}(t)$ has to be found. In a scenario where only a passive task instance is lost, no restart is required such that the binding of the active task instance remains the same and only a new binding for the passive task instance has to be found. Similarly, one of the passive routing paths between the new active task instances has to be activated. The new active routing path $\rho_j(m)$ depends on which tasks are affected by the failover and is being implicitly updated. To identify the application latencies after a failure f we use the notations $\mathcal{L}_i(a, f)$, $\mathcal{L}_{wc}(a, f)$ and $\mathcal{L}_{bc}(a, f)$.

IV. Worst-Case Failover Timing behavior

With a dynamic resource management that meets mapping decisions at run-time it is no longer possible to verify every system constraint at design-time. The timing behavior of applications is heavily influenced by timing interference from other applications which is unknown at design-time and can change once applications are updated and new functionality is added to the system.

To enable such a dynamic resource management even for safety-critical applications, it is crucial to automatically verify critical system aspects at run-time. In this context, an automated worst-case failover timing analysis is important, which allows to set bounds on the dynamic system behavior when meeting mapping decisions in order to achieve a deterministic and predictable system behavior. Even though our research focuses on automating the failover analysis to achieve dynamic system behavior, our equation can be also used to automatically evaluate the failover timing behavior at design-time such that multiple different mapping configurations can be explored.

To achieve such a predictable failover timing behavior, we first define the application failover time $\mathcal{F}(a, f)$ in Subsection IV-A. Afterwards, we present an upper bound and approach to derive the worst-case application failover time $\mathcal{F}_{wc}(a, f)$ in Subsection IV-B. Step by step, we first introduce the worst-case recovery time in Subsection IV-C, which has a major influence on the worst-case application failover time. Then, we analyze a single task failover scenario in Subsection IV-D until we finally derive a generalized formula for the worst-case application failover time in Subsection IV-E, where multiple tasks might be affected by the failover.

A. Application Failover Time

In a flawless operation mode, an application will periodically produce output data with the period P_a. Since the application latency is not constant and might differ between two consecutive iterations i and $i + 1$, we can calculate the output time difference between two application outputs in a failure-free operation as $\Delta O(a) = P_a + (\mathcal{L}_{i+1}(a) - \mathcal{L}_i(a))$, where $\mathcal{L}_{i+1}(a)$ and $\mathcal{L}_i(a)$ are the corresponding end-to-end application latencies of the iterations i and $i + 1$.

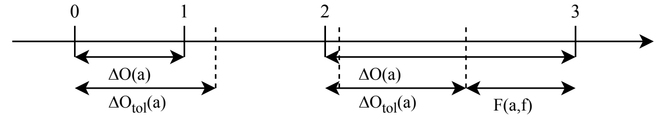


Fig. 2: Depiction of the application failover time: The vertical lines indicate the output time of the corresponding iteration, while the dashed vertical lines depict the latest output time that would be tolerated.

Therefore, the maximum delay that can be tolerated after an iteration i would be if it took the worst-case latency $\mathcal{L}_{wc}(a)$ to execute iteration $i+1$ such that the maximum tolerated output delay for the current iteration i can be calculated as

$$\Delta O_{tol}(a) = P_a + (\mathcal{L}_{wc}(a) - \mathcal{L}_i(a)). \quad (1)$$

However, when a task failure occurs, the application might take longer to produce a valid output, such that we define $\Delta O(a, f)$ as the output delay during a failover scenario with

$$\Delta O(a, f) = P_a + N_t(a, f) \cdot P_a + \mathcal{L}_{i+1}(a, f) - \mathcal{L}_i(a). \quad (2)$$

Here, $N_t(a, f) \cdot P_a$ describes the additional time delay due to iterations that are lost or missed due to failure and recovery time effects as the application is not able to operate during the downtime. Furthermore, $\mathcal{L}_{i+1}(a, f)$ is the application latency of the next valid iteration $i + 1$ after the failover with the new active task bindings.

With this we can define the application failover time $\mathcal{F}(a, f)$ as the time window between the point in time where a delay under a failure-free operation would not be tolerated any more until the point in time where a new output is available after the recovery:

$$\mathcal{F}(a, f) = \Delta O(a, f) - \Delta O_{tol}(a). \quad (3)$$

From Equation 3 it can be observed that if $\Delta O_{tol} \geq \Delta O(a, f)$, the failover time $\mathcal{F}(a, f)$ might be masked by ΔO_{tol} and not observable at the application output at all. Putting Equations 1 and 2 into Equation 3 we obtain the failover time:

$$\mathcal{F}(a, f) = N_t(a, f) \cdot P_a + \mathcal{L}_{i+1}(a, f) - \mathcal{L}_{wc}(a). \quad (4)$$

Figure 2 depicts the relation between the application failover time $\mathcal{F}(a, f)$, the output delay $\Delta O(a)$ and the maximum tolerated output delay $\Delta O_{tol}(a)$. Here, a failure occurs after iteration 2, such that one frame is missed, causing the output delay to exceed the tolerated delay.

B. Worst-Case Application Failover Time

Our goal is to derive an upper bound for Equation 4 to achieve a predictable failover behavior of the application. In a worst-case scenario $\mathcal{L}_{i+1}(a, f)$ from Equation 4 would be equal to the worst-case latency $\mathcal{L}_{wc}(a, f)$. Furthermore, we have to identify the worst-case number of iterations $N_{t,wc}(a, f)$ that are lost during the failover, such that we can calculate the worst-case application failover time as

$$\mathcal{F}_{wc}(a, f) = N_{t,wc}(a, f) \cdot P_a + \mathcal{L}_{wc}(a, f) - \mathcal{L}_{wc}(a). \quad (5)$$

Using this equation we can observe that a change to a lower worst-case latency $\mathcal{L}_{wc}(a, f)$ after a failure can in fact have a positive effect on the worst-case failover time as the next iteration after a failure would be able to propagate faster through the complete task chain and, therefore, reach the output earlier than expected beforehand.

For the total number $N_{t,wc}(a, f)$ of lost frames, we distinguish between the iterations $N_{l,wc}(a, f)$ that are lost immediately as they were currently held by a task that was affected directly by the failover and the iterations that are missed $N_{m,wc}(a, f)$ due to the system recovering too slow such that potential inputs are missed by the tasks:

$$N_{t,wc}(a, f) = N_{l,wc}(a, f) + N_{m,wc}(a, f). \quad (6)$$

However, before we can derive $N_{t,wc}(a, f)$ we have to introduce the worst-case recovery time $\tau_r(t, f)$ which is responsible for the number of iterations $N_{m,wc}(a, f)$ that will be missed due to task recovery. Afterwards, we are presenting an analysis to calculate $N_{t,wc}(a, f)$ in a simplified scenario where only a single task is affected directly by the failover and then generalize the formula for multiple recovering tasks.

C. Worst-Case Task Recovery Time

We define $\tau_r(t, f)$ for a failover f as the worst-case time frame that it takes from the occurrence of a failure until a task t is ready to receive and process the next input after recovery. Our definition of the recovery time is based on our system that is using a service-oriented middleware with a publish/subscribe pattern. Here, we make the case distinctions as presented in Equation 7, which can be found below.

For a task t it holds that $\tau_r(t, f) = 0$ if neither its active task binding nor the active task binding of its predecessor is affected by the failover.

For the case that a task is affected directly by the failure and has to be restarted, the passive task instance has to detect the failure. In the following, we assume that the worst case failure detection time τ_d is equal for every failure and every ECU. After detecting the failure, the task has to subscribe to its predecessor in order to receive its message, where we assume a worst case subscription time τ_{sub} resulting in $\tau_r(t, f) = \tau_d + \tau_{sub}$.

In case a predecessor task $p(t)$ is affected by the failover and not the task itself, the active task instance has to wait until the preceding task has restarted and sent a message to offer the service while the task itself has to wait to detect the failure until it can finally subscribe. $\tau_r(t, f) = \max(\tau_r(p(t), f) + \tau_{off}, \tau_d) + \tau_{sub}$.

In case a predecessor task $p(t)$ and the task t itself is affected by the failover, the task has to detect the failure and wait until the offer service message has arrived until it can subscribe just as in the previous case. With $\tau_r(p(t), f) + \tau_{off} > \tau_d$, we can then further simplify the formula for the last two cases to $\tau_r(t, f) = \tau_r(p(t), f) + \tau_{off} + \tau_{sub}$. With the worst-case recovery time it is now posible to derive the worst-case number of iterations that are lost during a failover.

D. Single Task Failover Scenario

In a scenario where only a single task is affected directly by a failure, the worst point in time where the most iterations are lost is when the affected active task instance has just finished the execution and the frame l_0 is lost together with the task such that for a single failing task $N_{l,wc}(t, f) = 1$.

If the passive task instance that is being started is able to recover on time $\tau_r(t, f)$ before the following frame m_0 has reached $p(t)$ no additional frame will be missed. Otherwise at least one additional frame m_0 will be missed depending on how fast the task is able to recover $\tau_r(t, f)$.

To calculate the earliest point in time τ_{m_0} where m_0 would reach the predecessor the end of the predecessor $p(t)$ of task t, we have to know how far the next frame m_0 was behind the lost frame l_0. The frames start with a time difference of exactly P_a. In the worst case, it took the lost frame l_0 the worst-case latency $\mathcal{L}_{wc}(t_l)$ to reach the output of the task, such that the following frame m_0 had the most time to minimize the time distance between them. In turn, m_0 would in the worst case propagate with the best-case latency $\mathcal{L}_{bc}(p(t))$ such that we can calculate τ_{m_0} as

$$\tau_{m_0} = P_a + \mathcal{L}_{bc}(p(t)) - \mathcal{L}_{wc}(t). \quad (8)$$

We assume that in the worst case every following frame m_i would propagate with the best case latency and therefore be exactly behind by P_a to the previous frame m_{i-1}:

$$\forall i > 0 : \tau_{m_i} = \tau_{m_{i-1}} + P_a \quad (9)$$

With this we can make the following case distinction for the worst-case number of missed frames:

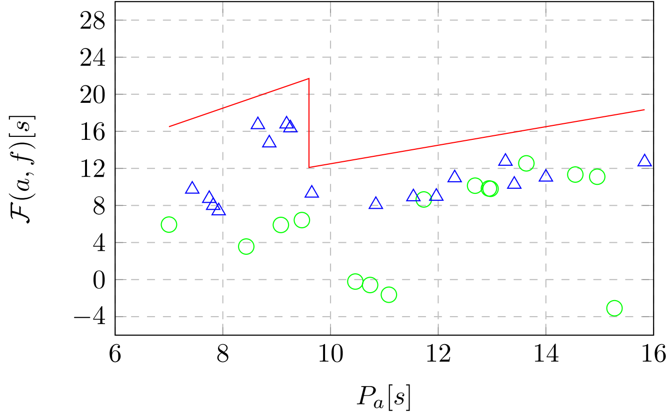
$$N_{m,wc}(t, f) = \begin{cases} 0 & \tau_{m_0} > \tau_r(t, f) \\ 1 + \lfloor \frac{\tau_r(t,f) - \tau_{m_0}}{P_a} \rfloor & \tau_{m_0} \leq \tau_r(t, f) \end{cases}. \quad (10)$$

If the frame m_0 would reach the end of $p(t)$ before the task t has recovered with the worst-case recovery time $\tau_r(t, f)$, then m_0 will be missed. As the following frames follow m_0 by P_a in the worst case, an additional number of $\lfloor \frac{\tau_r(t,f) - \tau_{m_0}}{P_a} \rfloor$ frames will be missed until the recovery is completed.

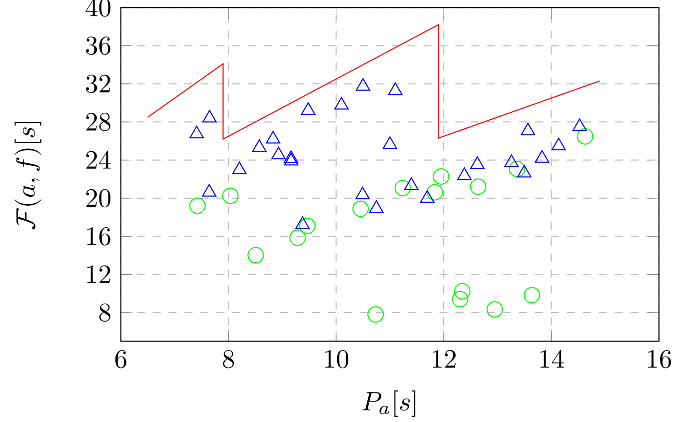
E. Multi Task Failover Scenario

For a scenario where multiple tasks have to recover, more than one frame might get lost immediately due to the failure if the application is using pipelining. In the following, we denote task t_l as the last task and t_f as the first task in the chain that has to recover. We assume that any frames that at the point of failure were between the predecessor task $p(t_f)$ of t_f and t_l are immediately lost. In the worst case, at least one frame l_0 is lost at t_l, which just has completed execution with the corresponding frame. To understand how many other frames between $p(t_f)$ and t_l are lost, we have to make similar considerations as in Subsection IV-D. The most frames are lost if the frame l_0 at t_l propagated with the worst case latency and the following frames propagated with the best case latency. Therefore, if l_1 propagated faster to $p(t_f)$ with $P_a + \mathcal{L}_{bc}(p(t_f))$ than it took for l_0 to reach the end of t_l with $\mathcal{L}_{wc}(t_l)$, at least

$$\tau_r(t,f) = \begin{cases} 0 & \alpha_{j+1}(t) = \alpha_j(t) \wedge \alpha_{j+1}(p(t)) = \alpha_j(p(t)) \\ \tau_d + \tau_{sub} & \alpha_{j+1}(t) = \beta_j(t) \wedge \alpha_{j+1}(p(t)) = \alpha_j(p(t)) \\ \tau_r(p(t),f) + \tau_{off} + \tau_{sub} & \alpha_{j+1}(t) = \alpha_j(t) \wedge \alpha_{j+1}(p(t)) = \beta_j(p(t)) \\ \tau_r(p(t),f) + \tau_{off} + \tau_{sub} & \alpha_{j+1}(t) = \beta_j(t) \wedge \alpha_{j+1}(p(t)) = \beta_j(p(t)) \end{cases} \quad (7)$$



(a) Failover scenario with failure of t_2.



(b) Failover scenario with failures of t_2 and t_7.

Fig. 3: Experimental results with $\mathcal{L}_{wc}(a) = 16.5s$, $\mathcal{L}_{wc}(a,f) = 19s$, $\tau_r(t,f) = 7.5s$. Measurements with a random failure time are marked with green circles, while the worst-case failover measurements are marked with blue triangles. The red curve representing our worst-case analysis is a strict bound that is not exceeded by any measurement. The results are as close as 6.0% below our analytically derived exact bound.

one additional frame is immediately lost during the failure. Therefore, we define

$$\tau_{l_1} = P_a + \mathcal{L}_{bc}(p(t_f)) - \mathcal{L}_{wc}(t_l) \quad (11)$$

as the time where l_1 would reach the end of execution at $p(t_f)$. Similar as before, all following frames l_i are following their predecessor by P_a:

$$\forall i > 1 : \tau_{l_i} = \tau_{l_{i-1}} + P_a \quad (12)$$

If τ_{l,l_1} is positive, which means l_1 reached $p(t_f)$ after the failure event, no other frame than l_0 is lost. Otherwise l_1 and possibly other frames are lost. Thus, we can calculate the number of lost frames as

$$N_l(a,f) = \begin{cases} 1 & \tau_{l_1} > 0 \\ 2 + \lfloor \frac{-\tau_{l_1}}{P_a} \rfloor & \tau_{l_1} \leq 0 \end{cases} \quad (13)$$

We assume that $\forall t \in T : \tau_{sub} + \tau_{off} < l_{t,bc}(t, \alpha(t,f)) + l_{c,bc}(m_{in}, \rho(m_{in}, f))$, such that the system is dominated by the failover of the first task t_f in the chain that failed. This means if t_f has recovered, we assume that all following tasks in the chain are able to recover on time such that no additional frames are lost except the frames $N_m(t_f, f)$ missed by t_f.

$$\tau_{m_0}(t_f, f) = (N_l(a,f) - 1) \cdot P_a + \tau_{l_1} \quad (14)$$

Here, we can reuse Equation 10 to calculate $N_{m,wc}(t_f, f)$ with $\tau_{m_0}(t_f, f)$, such that we can calculate the worst-case number of missed frames. By combining and simplifying the complete formula for $N_t(a,f)$ we obtain

$$N_t(a,f) = \lfloor \frac{\tau_r(t,f) + \mathcal{L}_{wc}(t_l) - \mathcal{L}_{bc}(p(t_f))}{P_a} \rfloor + 1. \quad (15)$$

By putting Equation 15 into Equation 5 we can finally derive the generalized formula to calculate the worst-case application failover time.

V. Experiments

In the following we present the results of our experiments conducted on a demonstrator setup to support our worst-case failover analysis using a fail-operational distributed neural network. In a usual system setup we would assume that multiple distributed applications are running at the same time. Here, our analysis can be applied for each application individually as long as worst-case and best-case latencies can be determined.

A. Testing Setup

For our experiments we use a setup consisting of 3 Rasperry-Pi 4B devices, with 8GB RAM and a quad-core Cortex-A72 with maximum frequency of 1.5 GHz. We chose a YOLOv3-tiny neural network implementation as our test application, which is a simplification of the structure used for object detection described in [13]. The neural network runs on Tensorflow-CPU as a backend and uses the Keras API. We split the neural network between the layers into 5 different tasks, with extra tasks for pre- and postprocessing adding up to 7 tasks in total. Through splitting the neural network and introducing it into our framework, a pipeline for the application arises, which improves the overall throughput of the neural network by adding more computational power. We use a star-topology network using a switch and TCP Ethernet connections for the communication between the devices. Our

analysis can be also applied to more complex architectures as long as the latencies for the message routings can be determined, which we assume as given for our approach. The Raspberry-Pis use a service-oriented communication middleware based on the SOME/IP standard with a publish/subscribe pattern [14]. Our framework allows to simulate ECU failures, by shutting down the framework instances on a Raspberry-Pi. Failures are detected via heartbeats and timeouts.

B. Failover Timing Measurements

For our experiments we obtain the application failover time $\mathcal{F}(a, f)$ by measuring the output delay $\Delta O(a, f)$ during a failover, while the tolerated output time interval $\Delta O_{tol}(a)$ can be calculated. Figure 3 presents the failover time of multiple experiments with a randomly varying application period P_a of a single task failure scenario (Figure 3a) and a multi-task failure scenario (Figure 3b). Experiments with a random failure time are marked with a green circle. Furthermore, we provoked worst-case scenarios by shutting down the ECU at the worst-case point in time, where results are marked with blue triangles. Some of the factors that cause the failover time to fluctuate are the varying recovery time and application latencies. A negative failover time implies that the failover occurred within the tolerated output interval $\Delta O_{tol}(a)$ such that no negative impact on the application timing behavior is observable. We used our worst-case analysis from Section IV obtain a strict upper bound which is plotted as the red curve.

Most importantly, we can observe that none of the measurements exceed the upper bound. Our randomly generated worst-case results are as close as 6.0% below our analytically derived exact bound. Note that by conducting more experiments, the measurements would come even closer. Furthermore, it can be observed that the blue worst-case measurements also result in higher failover times than the random measurements marked in green. Comparing Figure 3a with Figure 3b, we can examine that a failure, where multiple tasks are affected, overall leads to to a higher failover time as intuitively expected. Another interesting examination is that the upper bound builds a sawtooth-like curve with a decreasing slope with every step. With an increasing period P_a, the worst-case failover time increases linearly with a slope that is determined by the total number of lost iterations as described in Equation 5. However, as the system gains more time to recover with an increasing P_a, it will at one point loose one iteration less causing a step and a decreased slope. As a consequence, a slight period offset close to a step might have a big impact on the worst-case failover time.

Overall, the results show that our prediction of the worst-case failover time has adequately estimated an upper bound for the failover time experienced by the system.

VI. Conclusion

In this paper we have introduced a formal analysis to derive the worst-case failover time for fail-operational distributed automotive applications. Our upper bound allows to conduct an automated worst-case analysis to evaluate mappings at run-time. This way, unfeasible mappings can be identified and excluded such that a safe operation of safety-critical software within the bounds of the fault-tolerant timing interval can be guaranteed. We conducted experiments with a distributed fail-operational neural network on our hardware setup using an agent-based software platform, where we provoked worst-case failure scenarios to measure the failover time. Our randomly generated worst-case results are as close as 6.0% below our analytically derived exact bound.

In future work we will extend our analysis for applications with parallel paths in the application graph. Furthermore, we are planning to use the worst-case failover analysis to conduct automated mapping decisions at run-time using an agent-based mapping mechanism.

Summarized, our presented worst-case failover timing analysis allows to conduct an automated analysis at run-time to verify that the system operates within the bounds of the failover timing constraint such that a dynamic and safe behavior of autonomous systems can be ensured.

References

[1] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein, and M. Wolf, "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)*, 2018.

[2] P. Weiss, A. Weichslgartner, F. Reimann, and S. Steinhorst, "Fail-operational automotive software design using agent-based graceful degradation," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2020.

[3] T. Frese, T. Leonhardt, D. Hatebur, I. Côté, H. Aryus, and H. M., "Fault tolerance time interval," in *Proff H. (eds) Neue Dimensionen der Mobilität*, 2020.

[4] F. Oszwald, J. Becker, P. Obergfell, and M. Traub, "Dynamic reconfiguration for real-time automotive embedded systems in fail-operational context," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 206–209.

[5] F. Oszwald, P. Obergfell, M. Traub, and J. Becker, "Reliable fail-operational automotive e/e-architectures by dynamic redundancy and reconfiguration," in *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, 2019, pp. 203–208.

[6] F. Krichen, B. Hamid, B. Zalila, and B. Coulette, "Designing dynamic reconfiguration for distributed real time embedded systems," in *2010 10th Annual International Conference on New Technologies of Distributed Systems (NOTERE)*. IEEE, 2010, pp. 249–254.

[7] M. Glaß, M. Lukasiewycz, C. Haubelt, and J. Teich, "Incorporating graceful degradation into embedded system design," in *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE, 2009.

[8] B. Pourmohseni, F. Smirnov, S. Wildermann, and J. Teich, "Real-Time Task Migration for Dynamic Resource Management in Many-Core Systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 77. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, pp. 5:1–5:14.

[9] A. Sinha, A. Karmakar, B. Bhattacharya, S. Bhattacharya, and S. Ray, "Architecture of a fault tolerant system for real time embedded applications," in *ICCSC'02. 1st IEEE International Conference on Circuits and Systems for Communications. Proceedings (IEEE Cat. No.02EX605)*, 2002, pp. 194–197.

[10] D. Goswami, D. Müller-Gritschneder, T. Basten, U. Schlichtmann, and S. Chakraborty, "Fault-tolerant embedded control systems for unreliable hardware," in *2014 International Symposium on Integrated Circuits (ISIC)*, 2014, pp. 464–467.

[11] B. Akesson, A. Molnos, A. Hansson, J. A. Angelo, and K. Goossens, *Composability and Predictability for Independent Application Development,Verification, and Execution*. New York, NY: Springer New York, 2011, pp. 25–56.

[12] A. Weichslgartner, S. Wildermann, D. Gangadharan, M. Glaß, and J. Teich, "A design-time/run-time application mapping methodology for predictable execution time in mpsocs," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 5, 2018.

[13] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[14] *Scalable service-Oriented MiddlewarE over IP (SOME/IP)*, 2020. [Online]. Available: http://some-ip.com/