Mitigating Timing Error Propagation in Mixed-Criticality Automotive Systems

Thorsten Piper, Stefan Winter, Oliver Schwahn, Suman Bidarahalli, Neeraj Suri
DEEDS Group, Technische Universität Darmstadt, Germany
{piper,sw,os,suman,suri}@cs.tu-darmstadt.de

DRAFT TR-TUD-DEEDS-03-2015

***Abstract*—For mixed-criticality automotive systems, the functional safety standard ISO 26262 stipulates *freedom from interference*, i.e., errors should not propagate from low to high criticality tasks. To prevent the propagation of timing errors, the automotive software standard AUTOSAR provides monitor-based timing protection, which detects and confines task timing errors. As current monitors are unaware of a criticality concept, the effective protection of a critical task requires to monitor *all* tasks that constitute a potential source of propagating errors, thereby causing overhead for worst-case execution time analysis, configuration and monitoring. Differing from the *indirect* protection of critical tasks facilitated by existing mechanisms, we propose a novel monitoring scheme that *directly* protects critical tasks from interference, by providing them with execution time guarantees. Overall, our approach provides efficient low-overhead interference protection, while also adding transient timing error ride-through capabilities.**

I. Introduction

The automotive industry is encountering growing interest in the development and integration of mixed-criticality systems [1], i.e., systems containing components with varying degrees of assurance on timing and safety. This trend arises from the increasing multitude and complexity of innovative (often software based) driver assistance features while dealing with resource constraints of limited space, energy capacity and distribution, weight and, fundamentally, costs. Essentially, the integration of the historically segregated automotive systems, which have been conservatively designed following a *one function per electronic control unit (ECU)* approach, offers cost saving potential for hardware and wiring, as it entailed up to 100 federated ECUs distributed in modern luxury cars [2].

Similar to the "gold standard for partitioning" in integrated modular avionics [3], the functional safety standard for road vehicles ISO 26262 [4] permits the integration of elements with differing criticality, as long as partitioning mechanisms can verifiably provide *freedom from interference* in both the spatial and temporal domains, i.e., regarding memory accesses and timing behavior. In automotive systems, partitioning is usually supported by hardware [5], [6], the operating system (OS) [7], [8], or a combination of both (e.g. virtualization) [9].

The established AUTOSAR standard [10] for automotive software addresses freedom from interference through a set of safety mechanisms in its Technical Safety Concept [7] that are provided as services by the AUTOSAR OS. To support partitioning in the temporal domain, the OS provides monitoring of task execution time budgets, activation frequencies, and resource lock times. The violation of a monitor policy constitutes a timing error, whose propagation is prevented by stopping the responsible task and freeing locked resources.

AUTOSAR schedules tasks based on a fixed-priority preemptive scheme [11], in which the processor executes the highest priority task among the tasks ready for execution. Without timing protection, timing errors, i.e., any fault that leads to a violation of the specified worst-case execution time (WCET) of a task, may propagate from high priority to low priority tasks. Due to rate monotonic priority assignment [12], [13], the priority and criticality of a task are usually unaligned, i.e., the most critical task is not necessarily assigned the highest priority. Thus, criticality inversion [14] may occur through timing error propagation. AUTOSAR's timing protection mechanisms, specifically execution time monitoring, aim at detecting and confining timing errors to the task where they originate. To protect critical tasks from the effects of timing errors arising in less critical tasks, all such less critical tasks require execution time monitoring.

This approach has a number of undesired implications in mixed-criticality systems:

- Critical tasks are protected *indirectly* by confining timing errors in less critical tasks. To protect a given critical task, *all* less critical tasks must be individually and correctly monitored, which is an error-prone process.
- Monitoring causes configuration and run-time overhead (cf. Section V), which can be reduced by focusing on critical tasks only.
- Worst-case execution times of uncritical or less critical tasks are often over-approximated [15], [16], which, when enforced via monitors, may impact overall system utilization negatively.

Paper contribution

On this background, we propose a novel, criticality-aware AUTOSAR run-time monitoring scheme that guarantees critical tasks a configurable execution time budget to *directly* protect them from timing errors of other tasks. The budget guarantee is enforced by monitoring a task's preemption budget (PB), which determines how long a task may be preempted without compromising its timely execution. By focusing only on critical tasks, we reduce the overhead for run-time monitoring and WCET analysis of non-critical tasks. Further, any unused computation time of critical tasks may be spent by erroneous non-critical tasks to eventually finish (transient ride-through). To put this contribution in context,

we highlight that currently AUTOSAR lacks support for mixed criticality scenarios by its monitoring mechanisms. Our work helps to provide this support.

Paper structure

We review the work related to our problem scope in Section II. In Section III, we provide background on AUTOSAR that is essential in understanding the proposed preemption budget monitor, which is presented in Section IV. In Section V, we evaluate the efficiency and overhead of our approach in a case study, and Section VI concludes the paper.

II. Related Work

Following a standalone exposition of the body of work relevant to our problem scope, we summarize its viability on addressing timing error propagation.

Related to our approach is the work by de Niz et al. [14], who identify a criticality inversion problem, for which less critical tasks may block more critical tasks in fixed-priority preemptive systems, if criticality and priority are not aligned. The authors first try to address criticality inversion by the criticality driven priority assignment scheme *Criticality As Priority Assignment* (CAPA), which has the drawback that more critical tasks with long periods may block the execution of less critical tasks with short periods, resulting in deadline misses. As an improvement, they introduce zero-slack scheduling, a scheme that is based on a dual mode task model (normal and overload) with the aim to maximize resource utilization, while providing protection from criticality inversion.

The *period transformation* approach proposed by Sha et al. [17] slices critical tasks with longer periods than less critical tasks in sections, such that each of those sections has a shorter period than any less critical task. Scheduling such a sliced set with a rate monotonic approach will result in task criticalities and priorities being aligned. However, this comes with the drawback of increased system management overhead, additional complexity of sharing data across slices, additional development effort for task slicing, and the basic requirement that tasks must be sliceable.

Ficek et al. [15], [18] developed a design workflow that provides guidance on how to effectively apply the previously discussed CAPA and period transformation approaches, together with AUTOSAR's execution time monitoring facilities to the overall system design, in order to ensure freedom from interference for critical tasks.

Baruah et al. [19], [20] propose an extension to the fixed-priority preemptive scheduling approach, in which a system executes either in LO-criticality (normal) or HI-criticality mode. If any task violates its assigned execution time budget, a switch to the HI-criticality mode occurs, and task priorities are re-assigned in such a way that a set of predefined critical tasks remains schedulable.

In summary, the related work on mixed criticality systems (a comprehensive review is given by Burns and Davis [1]) focuses mostly on scheduling algorithms and schedulability analysis and not on timing errors, or how to prevent their propagation at run-time by monitoring. Additionally, these approaches do not directly conform to the AUTOSAR model, thus limiting their usage as such. The work of Ficek et al. [15], [18] is an exception in this respect. Contrary to their work, the approach proposed in this paper does not require a (re-)design of the system to provide execution time guarantees and freedom from interference to critical tasks.

III. AUTOSAR System Model

This section provides the reader with a background on AUTOSAR's scheduling model, task model, and timing protection that are essential in understanding the problem scope and the solution of preemption budget monitoring proposed in Section IV.

The AUTOSAR OS [21] is a statically configured multitasking OS where all system objects, such as tasks and resources, are allocated at build time. The OS schedules tasks in a fixed-priority preemptive manner [11] executing the highest priority task among all ready tasks. For the assignment of task priorities [22], a rate monotonic scheme [12], [13] is commonly used, where the period or deadline of a task determines its priority (i.e., the shorter the period/deadline, the higher the priority). Task priorities are generally static, with the exception of the priority ceiling protocol [23], which is used to avoid priority inversion when resources are shared across tasks. In such situations, the priority of tasks that hold a shared resource is temporarily raised to the priority ceiling of the resource.

We define a task τ_i as a tuple

$$\tau_i = (P_i, C_i, T_i, D_i, \zeta_i)$$

where:

- P_i is the static priority,
- C_i is the worst-case execution time (WCET),
- T_i is the period,
- D_i is the deadline, and
- ζ_i is the criticality of the task.

In our model, higher values of P_i and ζ_i indicate higher priority and criticality, respectively. To achieve a clearer presentation, we assume that $D_i = T_i$ throughout the paper, unless stated otherwise. The WCET C_i constitutes the uninterrupted, maximum possible execution time of a task, and is either identified analytically (via static code analysis), experimentally (via tracing/measurement) or through simulation [24], [25]. Depending on the assurance that the employed method provides, a buffer value is usually added to the WCET to compensate for inaccuracies. Thorough WCET analysis is essential in determining the system schedule and conducting schedulability analysis, which proves whether the schedule meets the overall timing requirements and ensures that all tasks meet their deadline under *error-free* conditions.

A. Timing Error Propagation

In the presence of a timing error, which we define as a task τ_i exceeding its WCET C_i, the schedule's underlying assumptions are invalidated. Undetected timing errors may impact the

DRAFT TR-TUD-DEEDS-05-2015

timing of fault-free tasks through *error propagation*, and lead to failures in the form of deadline violation (not finishing the execution until D_i) of fault-free tasks. To illustrate such a scenario, we assume a system with tasks τ_A, τ_B and τ_C, as shown in Table I.

TABLE I
TIMING PROPERTIES OF THE EXAMPLE SYSTEM.

Task τ_i	WCET C_i	Deadline D_i	Priority P_i
A	2 ms	7 ms	3
B	2 ms	7 ms	2
C	2 ms	7 ms	1

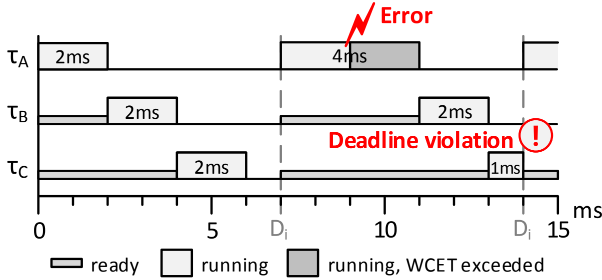


Fig. 1. Deadline violation of τ_C due to a propagated timing error.

Figure 1 depicts the task timing. During the first period from 0 ms to 7 ms, all tasks are error-free and finish before their deadline. In the second period from 7 ms to 14 ms, τ_A is subject to a timing error at 9 ms (indicated by the red bolt) that prolongs its execution time by 2 ms. The error of τ_A propagates and delays the execution of τ_B and τ_C by 2 ms. When τ_C starts to execute at 13 ms, the remaining execution time does not suffice to finish until its deadline at 14 ms. The consequence is a deadline violation and timing failure of τ_C.

B. AUTOSAR Timing Protection

The propagation of timing errors from one task to another can be detected and prevented by timing protection mechanisms (TPMs) that monitor task run-time behavior. AUTOSAR specifies and implements the following TPMs as OS services [7], [21], which are selectively enabled on a per-task basis.

- **Execution time monitoring** monitors a task's execution time and compares it to a budget. When the budget is exhausted without the task having finished, a timing error is detected.
- **Inter-arrival time monitoring** monitors a task's activation frequency within a statically configured time frame. When an activation limit is exceeded, a timing error is detected.
- **Locking time monitoring** limits the blocking time of tasks imputable to priority ceiling [23] or disabling interrupts. For each resource and each task, a lock time budget can be specified, and locking a resource for longer than its budget for that task constitutes a timing error.

Upon detection of a timing error, it can be locally confined by killing the task or its task group (OS application). Killing a task results in an abortion and failure of the task.

AUTOSAR detects timing errors accountable to WCET violations by execution time monitoring (ETM). As AUTOSAR's monitoring mechanisms currently lack support for mixed-criticality scenarios, ETM can only *indirectly* protect critical tasks by individually monitoring all tasks from which errors could potentially propagate, which is an error-prone process. In addition, ETM is inefficient compared to a criticality-aware monitoring scheme in mixed-criticality scenarios with few critical and many non-critical tasks, due to the overhead that monitoring the non-critical tasks entails. Recalling our previous example (Table I), we assume that task τ_C is critical, while tasks τ_A and τ_B are non-critical. To guarantee freedom from interference to τ_C, both τ_A and τ_B require monitoring using ETM, while a *direct* criticality-aware monitor would only require monitoring τ_C. In Section V, we compare the run-time overhead of ETM and a novel criticality-aware monitoring scheme that we propose in the next section.

IV. PREEMPTION BUDGET MONITORING

As augmentation to the existing monitoring infrastructure of AUTOSAR, we propose a monitoring scheme that is specifically suited to mixed-criticality systems. It is based on the idea that instead of monitoring the timing behavior of all non-critical tasks that constitute potential sources of timing errors, it would be more efficient to only monitor critical tasks with the aim of providing a guaranteed execution time budget to them. Differing from the *indirect* protection of critical tasks facilitated by existing mechanisms, this approach *directly* protects critical tasks from interference. In consequence of this paradigm shift, non-critical tasks are eligible for transient error ride-through (see Section IV-B).

The execution time guarantee is provided by monitoring the preemption time of critical tasks, i.e., the time spent in the *ready* state waiting for execution. If the preemption time exceeds a threshold value, which we term *preemption budget* (PB), the immediate start of the monitored task is enforced by re-queueing it with the highest priority in order to prevent interference by other tasks. For systems with only one critical task τ_i, the task's PB_i can intuitively be defined as $D_i - C_i$, as the task needs to start its *uninterrupted* execution C_i time units before its deadline D_i.

For systems with several critical tasks, determining PB_i is more complex, as potential preemptions through other PB-monitored tasks with a higher *precedence* (explained below) need to be factored in. The problem can be thought of as *response time analysis* [26], but with an inverted time line. We therefore make an argument along the lines of [26] and define the response time R_i as the sum of the WCET C_i and the total worst-case interference I_i of other critical tasks with higher precedence on τ_i through preemption.

$$R_i = C_i + I_i \quad (1)$$

TR-TUD-DEEDS-03-2015 DRAFT

The PB_i of a task τ_i then follows as

$$PB_i = D_i - R_i \tag{2}$$

We argue that the interference on task τ_i from a task τ_j through preemption is nC_j, where n denotes how often τ_j executes within R_i, or in other words, how often its period T_j fits in R_i. For non-integer values of $n = R_i/T_j$, n has to be rounded up by the ceiling function, to account for the fact that τ_j will always preempt τ_i for its full execution time C_j (due to fixed-priority preemptive scheduling). The worst-case interference from a task τ_j on task τ_i is therefore given by

$$\left\lceil \frac{R_i}{T_j} \right\rceil * C_j$$

Consequently, the total interference I_i of other tasks on τ_i follows as

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j \tag{3}$$

where $hp(i)$ is the set of tasks that might preempt τ_i due to higher monitor precedence. The precedence between PB-monitored tasks is defined as follows. The higher a task's criticality, the higher its precedence. For tasks of equal criticality, their precedence is defined in alignment to their priority.

Combining (1) and (3), the unknown term R_i appears on both sides of the equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j$$

which can be iteratively solved as follows.

Let R_i^n be the nth approximation to R_i.

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil * C_j \tag{4}$$

Starting with $R_i^0 = C_i$, the iteration terminates when $R_i^{n+1} = R_i^n$. The iteration is guaranteed to converge if the processor utilization is $\leq 100\,\%$ [26].

In Table II we provide an example for the calculation of the PB. The example system consists of three tasks: τ_B is of high criticality, τ_C is of medium criticality, and τ_A is non-critical.

TABLE II
ASSIGNING PREEMPTION BUDGETS IN A MIXED-CRITICALITY EXAMPLE.

Task τ_i	WCET C_i	Period T_i	Prio. P_i	Crit. ζ_i	PB_i
A	2 ms	6 ms	2	0	n.a.
B	2 ms	8 ms	1	2	6 ms
C	3 ms	12 ms	0	1	7 ms

The PBs are computed according to the precedence relations defined above, i.e., from the highest critical task to the lowest. Therefore, we start the computation of the PB for task τ_B. As task τ_B is the highest critical task, the set $hp(B)$ is empty, and using (4) we determine $R_B = R_B^0 = C_B = 2\,\text{ms}$. Using (2) it follows that $PB_B = 8\,\text{ms} - 2\,\text{ms} = 6\,\text{ms}$. The next critical task is τ_C, for which we have to factor in that it might be preempted by the PB monitor of task τ_B due to its higher precedence. Therefore, the $R_C = R_C^1 = C_C + \left\lceil \frac{C_C}{T_B} \right\rceil * C_B = 3\,\text{ms} + 2\,\text{ms} = 5\,\text{ms}$ and $PB_C = 12\,\text{ms} - 5\,\text{ms} = 7\,\text{ms}$.

In the presence of a timing error, the PB monitor would force the execution of τ_B at 6 ms after its activation, providing τ_B a guaranteed execution time of $C_B = 2\,\text{ms}$. For τ_C, the PB monitor would force its execution at 7 ms, providing τ_C a guaranteed execution time of $C_C = 3\,\text{ms}$, and accounting for a possible preemption by τ_B for up to $C_B = 2\,\text{ms}$.

A. Integration with AUTOSAR Task State Model

We integrate the PB monitor into the AUTOSAR task state model as depicted in Figure 2.

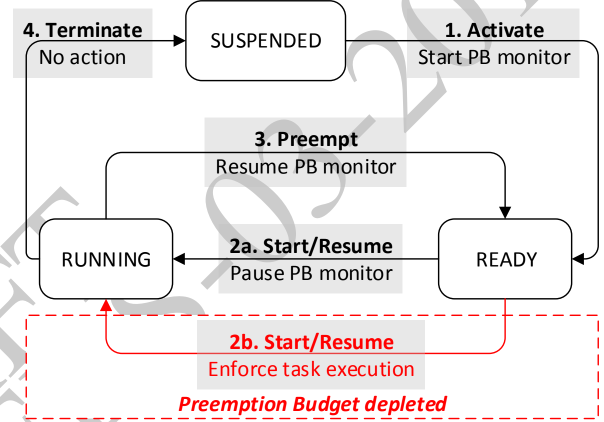


Fig. 2. Task state transitions and corresponding PB monitor actions.

After system start-up, tasks are in the *suspended* state. Once a task is activated, it enters the *ready* state and the PB monitor is started (*1. Activate*). As long as the task is in the *ready* state, its PB is consumed. In the error-free case, the task will eventually be dispatched (*2a. Start/Resume*) and enter the *running* state, from which it will either be preempted (*3. Preempt*) and be resumed later, or it will terminate after finishing its execution (*4. Terminate*). If the task leaves the *ready* state, the monitor will be paused. It will be resumed once the task re-enters the *ready* state.

In the erroneous case (*2b. Start/Resume*), the task will deplete its PB and a timer interrupt will trigger. To guarantee the critical task its WCET as execution time, it is either immediately started or enqueued with the highest priority, depending on whether higher precedence tasks have their PB depleted as well. The case study in Section V provides further details on the monitor's implementation and its performance in a transient and a permanent timing error scenario.

B. Transient Error Ride-through

A transient error is a temporary error that disappears after a limited amount of time. For example, a task with a nominal WCET of 2 ms may exceed its WCET by 1 ms and run for 3 ms before it finishes. As long as the task finishes before its deadline, it has not failed – the task made a *transient*

error ride-through. For tasks that are protected using AUTOSAR's ETM, transient error ride-throughs are infeasible, as the monitor strictly enforces the configured execution time budget. Consequently, the task is forced to fail, although it could eventually have finished, depending on the overall CPU utilization and timing constraints.

Contrary to ETM, PB monitoring allows for transient error ride-throughs. We again consider our example system as specified in Table II, but assume that only task τ_C is critical. For this modified scenario $PB_C = 12\,\text{ms} - 3\,\text{ms} = 9\,\text{ms}$, which is monitored. As shown in Figure 3, task τ_B is subject to a timing error at 10 ms (indicated by the red bolt) and continues to execute.

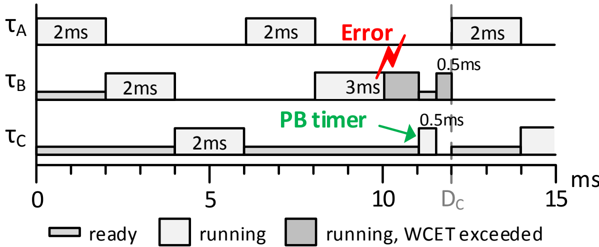


Fig. 3. Example of a transient error ride-through of task τ_B.

At 11 ms, the PB of task τ_C is exhausted (remember that the PB is only consumed in the ready state) and the monitor enforces τ_C to execute. τ_C finishes its execution after 0.5 ms and has therefore not consumed all of its WCET. The remaining 0.5 ms are used by τ_B to eventually finish its execution at 12 ms, which is before its deadline at 16 ms. Accordingly, τ_B performed a transient error ride-through.

C. Applicability to Multi-core Systems

Although much research has gone into multi-core mixed criticality scheduling (e.g. [27]), AUTOSAR uses an approach based on static partitioning [28], [29], in which a static task set is assigned to each partition/core. A possible explanation is provided by Reinhardt and Morgan [9], as "automotive software development has yet to exploit parallel processing in an efficient manner because legacy code, designed to run on single core systems, is difficult to adapt to run in parallel on multi-core systems." So, although we consider a single-core environment throughout the paper, the given analysis should be equally applicable to multi-core AUTOSAR systems with a static task partitioning.

D. Limitations and Possible Solutions

The partitioning of a system in different criticality levels using PB monitoring is subject to a set of constraints in order to avoid effects similar to those of the CAPA approach discussed in Section II. To illustrate the problem, we assume an example system with three tasks of different criticality, as shown in Table III.

According to the task precedence relationship, we first calculate the PB of τ_C and then of τ_B. It follows that

TABLE III
EXAMPLE OF CONFLICTING TASK CONSTRAINTS IN A THREE CRITICALITY SYSTEM.

Task τ_i	WCET C_i	Period T_i	Prio. P_i	Crit. ζ_i
A	2 ms	6 ms	2	0
B	2 ms	8 ms	1	1
C	10 ms	30 ms	0	2

$PB_C = 30\,\text{ms} - 10\,\text{ms} = 20\,\text{ms}$. For τ_B, we calculate $PB_B = 8\,\text{ms} - 2\,\text{ms} - 10\,\text{ms} = -4\,\text{ms}$. As PB_B is negative, assigning a PB to τ_B is infeasible. Consequently, no execution time guarantee can be given to τ_B. The explanation is straightforward: as τ_C has a higher monitor precedence, τ_C could preempt τ_B whenever its PB_C is depleted. As τ_C has a WCET C_C of 10 ms, τ_B can impossibly meet its deadline $D_B = T_B = 8\,\text{ms}$ in that case.

The described effect cannot occur for dual criticality systems (non-critical and critical), as the set of critical tasks simply represents an ordered subset of the overall system and remains schedulable, if the overall system was schedulable. For systems with more than two criticalities, the effect does not occur as long as task criticalities are aligned with task priorities. For all other cases, we propose the following solutions:

- Invert the precedence of conflicting tasks (in our example τ_B and τ_C), so that the task with the shorter period τ_B can preempt τ_C. To continually provide freedom from interference to the more critical task τ_C in such a scenario, the execution time of τ_B would require additional monitoring with ETM.
- Re-arrange task priorities by period transformation to align criticalities and priorities (similar to the approach proposed by Ficek in [18]).

V. Case Study

In this section, we evaluate our implementation of preemption budget monitoring (PBM) and compare it in terms of memory and run-time overhead to execution time monitoring (ETM). Further, we assess the effectiveness of PBM in direct comparison to ETM under the following timing error scenarios:

- **Transient:** A task exceeds its worst-case execution time to eventually finish its execution before its deadline.
- **Permanent:** A task is stuck, for example in a livelock or deadlock, and is therefore unable to finish its execution.

Our test system is a simplified adaptive cruise control (ACC) that consists of seven tasks, as shown in Table IV. We assume that the deadline of each task matches the task's respective period, and that τ_6 is the only critical task in our system.

We have implemented the test system for the XKT564L evaluation board by Freescale [30] using a widely used, commercial AUTOSAR tool suite for system integration. The XKT564L hosts a 32-bit dual core Power Architecture microcontroller unit (MCU) with 1 MiB of flash memory and 128 KiB of RAM (both ECC). The MCU is specifically

DRAFT
TR-TUD-DEEDS-03-2015

TABLE IV
TASK CONFIGURATION OF ACC CASE STUDY.

Task	WCET C_i	Period T_i	Priority P_i	Critical
τ_1	30 µs	250 µs	7	-
τ_2	50 µs	250 µs	6	-
τ_3	145 µs	500 µs	5	-
τ_4	15 µs	500 µs	4	-
τ_5	20 µs	500 µs	3	-
τ_6	15 µs	1,000 µs	2	✓
τ_7	20 µs	1,000 µs	1	-

developed for safety-critical applications and both cores are operated in lock-step mode to detect hardware run-time errors.

A. PBM - Implementation Details

Our PBM implementation is an extension to the monitoring infrastructure of a widely used, commercial AUTOSAR OS. We closely resemble the structure of the existing monitoring mechanisms to support a seamless integration of our approach. Following the task state diagram (cf. Figure 2, Section IV), we added monitor hooks to the kernel's *enqueue* and *dispatch* functions, in order to handle task activation, start/resume and preemption in the monitor. Further, we implemented a handler for PB depletion (cf. Figure 2, transition 2b) to enforce the execution of critical tasks if necessary.

TABLE V
STATIC OVERHEAD OF PBM.

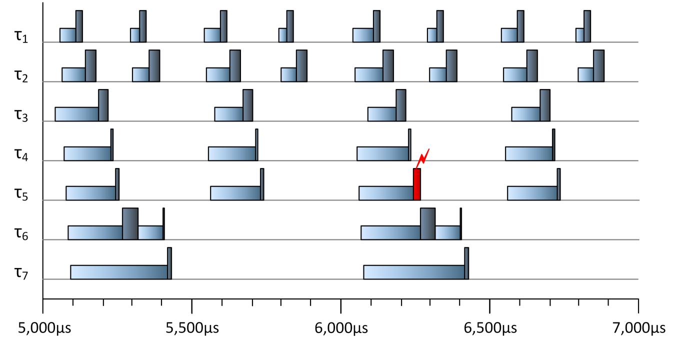
	Baseline	with PBM	Overhead
Kernel SLOC	25,736	25,985	+159 (0.6 %)
Flash ROM	43,903	44,319	+416 (0.9 %)
RAM	47,952	47,980	+28 (0.06 %)

In Table V we review the static overhead of PBM on the source code and binary levels. We utilized the tool SLOCCount [31] to determine the source lines of code (SLOC) of the baseline kernel, and compare it to the version with PBM as a measure for the implementation complexity of PBM. Overall, PBM increases the SLOC of the AUTOSAR OS kernel by 159 lines, which is an increase of 0.6 %.

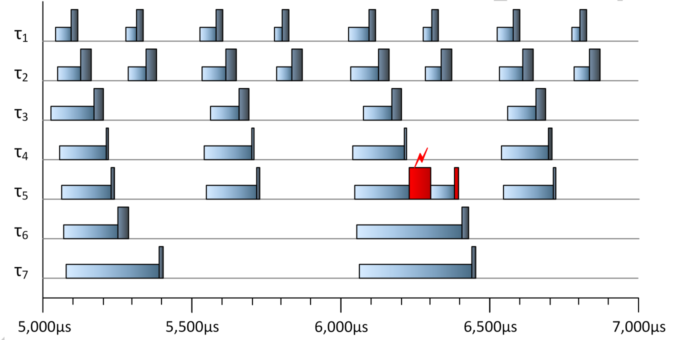
The static flash ROM and RAM consumption was obtained using the tool *objdump* from the GNU Binutils toolsuite [32]. PBM consumes an additional 416 bytes (0.9 %) of flash ROM, and 28 bytes (0.06 %) of RAM.

B. Timing Error Scenarios

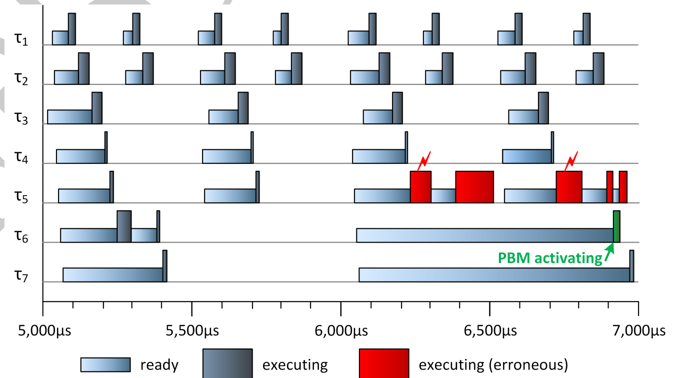
We evaluate the efficiency of PBM and compare it to ETM in a transient and a permanent timing error scenario. The errors are injected at run-time using a software-implemented fault injection (SWIFI) tool prototype for AUTOSAR that extends previous work [33]. For both error scenarios, the injection location is task τ_5 and the trigger condition is set to the third execution of the task after system start (activated around 6,100 µs).



(a) ETM: Prevents transient ride-through



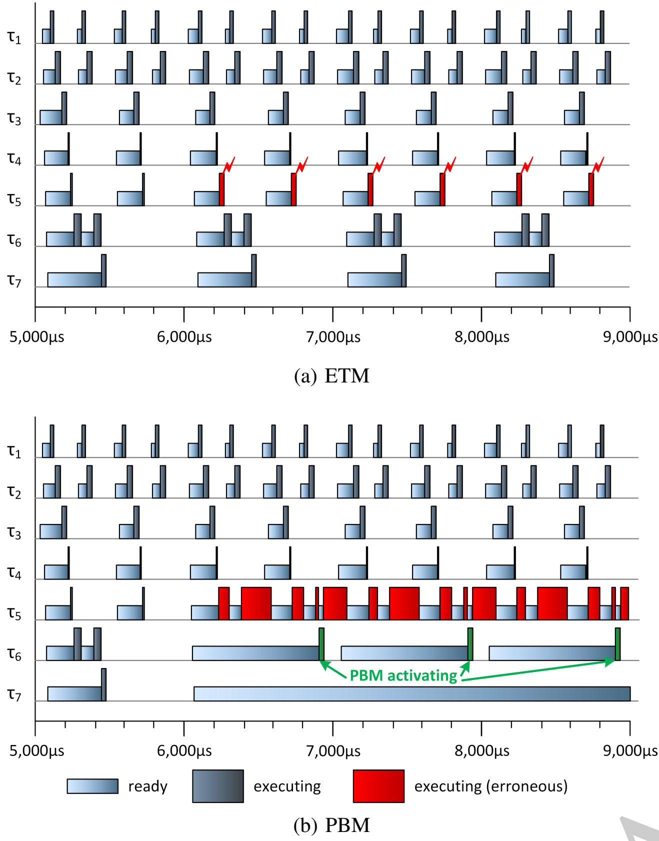
(b) PBM: Transient ride-through (case I)



(c) PBM: Transient ride-through (case II)

Fig. 4. Transient timing error scenario.

The three graphs in Figure 4 depict the transient error scenario for ETM and PBM. The graphs cover two full periods of the tasks with the longest period in the system (τ_6, τ_7). In the first period (5,000 µs to 6,000 µs), the error-free timing is shown. In the second period (6,000 µs to 7,000 µs), a transient timing error that lasts 100 µs is injected at 6,250 µs in τ_5. Figure 4a shows how ETM detects the error after τ_5 has consumed its WCET. As ETM strictly enforces the configured budget, τ_5 gets killed by the OS and thus fails. Figure 4b shows the same scenario for PBM. As the preemption budget of τ_6 is not consumed until 6,900 µs, the PBM does not interfere with τ_5, which eventually finishes its execution successfully. Figure 4c depicts a slightly different scenario with two error injections, in order to highlight the interaction between PBM



(a) ETM

(b) PBM

Fig. 5. Permanent timing error scenario.

and the erroneous task. The first injection at 6,250 µs serves the sole purpose of delaying the execution of τ_6. For the second injection at 6,750 µs, we observe that PBM gets activated around 6,900 µs and preempts τ_5. After executing for less than its WCET, τ_6 finishes and τ_5 uses the remaining budget from τ_6 to successfully finish. In all three scenarios, ETM and PBM effectively prevented task failures of the critical task τ_6, while the latter two scenarios demonstrate the transient ride-through capabilities of PBM.

The two graphs in Figure 5 illustrate the permanent error scenario for ETM and PBM. The graphs cover four full periods of the task with the longest period in the system to better observe the effect of the permanent error. In the first period (5,000 µs to 6,000 µs), the error-free timing is shown. After the first period, a permanent timing error is injected in τ_5 at 6,250 µs. Figure 5a shows how ETM repeatedly detects the error after τ_5 has consumed its WCET and kills the task. Figure 5b shows the same scenario for PBM, which repeatedly detects an imminent error propagation from task τ_5 to τ_6 and prevents it by assigning τ_6 its guaranteed execution time budget. Both monitors, ETM and PBM are effective in preventing task failures of the critical task τ_6 that are accountable to timing error propagation. At the same time, ETM and PBM are incapable of mitigating the failure of τ_5 that is evoked by its permanent error. To address such error scenarios, we recommend to additionally employ a monitor that provides complementary mitigation strategies, such as an external watchdog.

C. *Comparison of run-time overhead*

Monitoring the timing behavior of tasks entails run-time overhead whenever the monitor is invoked. In general, ETM is invoked at task start, preemption, resume, and termination. Correspondingly, PBM is invoked at task activation, start, preemption, and resume. As our case study only has few preemption/resume events, we focus our evaluation on activation, start and termination. It can be expected that the overall overhead grows linearly for ETM and PBM alike for systems with many preemptions, due to structural similarity of the monitors.

Our measurements over 500 activation, 500 start and 500 termination events demonstrate a very constant run-time behavior. For both monitors, the monitor start (at each task activation for PBM and each task start for ETM) consumes 2 µs on average, while the monitor stop routine (at each task start for PBM and each task termination for ETM) consumes 2.2 µs on average. Therefore, the monitor overhead for one period of a task's execution is fixed at 4.2 µs for ETM and PBM alike.

TABLE VI
MONITORING OVERHEAD FOR ETM AND PBM.

Task	ET_{max}	Period T_i	ETM, per		PBM, per	
			ET_{max}	Period	ET_{max}	Period
τ_1	24 µs	250 µs	17.5 %	1.7 %	-	-
τ_2	38 µs	250 µs	11.1 %	1.7 %	-	-
τ_3	113 µs	500 µs	3.7 %	0.8 %	-	-
τ_4	9 µs	500 µs	46.7 %	0.8 %	-	-
τ_5	13 µs	500 µs	32.3 %	0.8 %	-	-
τ_6	12 µs	1,000 µs	-	-	35.0 %	0.4 %
τ_7	16 µs	1,000 µs	-	-	-	-
Systemwide overhead				5.9 %		0.4 %

Table VI compares the overhead for ETM and PBM in our test system. For ETM, each task with a higher priority than the critical task τ_6 requires monitoring, to prevent error propagation to τ_6, while for PBM, only the critical task τ_6 requires monitoring. In direct comparison to the tasks' measured maximum execution time ET_{max}, the monitoring overhead is between 3.7 % and 46.7 % per monitored task.

To put these figures into a systemwide perspective, a comparison to the period of a task is more meaningful than to ET_{max} because the period determines how often a task (and thus its monitor) is executed. For ETM, the overhead per period is between 0.8 % and 1.7 %, which results in an aggregated systemwide overhead of 5.9 %. For PBM, the overhead per period is 0.4 %, which results in an aggregated systemwide overhead of 0.4 %.

D. *Summary*

Our case study showed that PBM and ETM both protect critical tasks from failures due to timing error propagation in

DRAFT
TR-TUD-DEEDS-03-2015

transient and permanent error scenarios. For transient errors, only PBM enables non-critical tasks to perform a transient error ride-through. In terms of overhead, PBM outperforms ETM by a magnitude. The expected benefit of using PBM over ETM in other scenarios largely depends on the distribution of critical and non-critical tasks, with a preference for PBM in scenarios with a low number, and for ETM in scenarios with a high number, of critical over non-critical tasks.

VI. Conclusion

We have presented preemption budget monitoring (PBM), a novel monitoring approach that guarantees freedom from interference in the temporal domain to critical tasks in mixed-criticality systems. We have implemented our approach as an extension to the existing monitoring infrastructure of a widely used, commercial AUTOSAR OS with a 0.9 % increase in binary code size and less than 0.1 % increase in memory consumption. The evaluation of our approach in an adaptive cruise control scenario showed that PBM effectively prevents the propagation of timing errors from non-critical to critical tasks with a run-time overhead that is a magnitude lower than existing approaches. PBM achieves these impressive results by monitoring only critical tasks and avoiding the overhead of monitoring non-critical tasks. Therefore, we expect PBM to perform equally well for any mixed-criticality systems, in which few critical tasks require protection from possible failures of many non-critical tasks. Furthermore, in contrast to existing approaches, PBM enables transient ride-through to allow non-critical tasks to recover from transient timing errors and thereby improves overall system reliability.

Acknowledgment

The authors would like to thank Tom Fuhrman (from General Motors, Warren, USA) for his valuable feedback and the interesting discussions. Research supported in part by DFG GRK 1362, Loewe CASED and EC-SPRIDE.

References

[1] A. Burns and R. I. Davis, “Mixed Criticality Systems - A Review,” Department of Computer Science, University of York, UK, Tech. Rep., 2014.

[2] R. N. Charette, “This Car Runs on Code,” *IEEE Spectrum*, Feb. 2009.

[3] J. Rushby, “Partitioning for Avionics Architectures: Requirements, Mechanisms, and Assurance,” NASA Langley Research Center, NASA Contractor Report CR-1999-209347, Jun. 1999.

[4] *ISO 26262: Road vehicles – Functional safety*, International Organization for Standardization, 2011.

[5] A. Wasicek, C. El-Salloum, and H. Kopetz, “A System-on-a-Chip Platform for Mixed-Criticality Applications,” in *Proc. of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2010.

[6] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, “FlexPRET: A Processor Platform for Mixed-Criticality Systems,” in *Proc. of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*, 2014.

[7] AUTOSAR, *Technical Safety Concept Status Report*, AUTOSAR Release 4.2.1, Document ID 233, 2014.

[8] D. Bertrand, S. Faucou, and Y. Trinquet, “An analysis of the AUTOSAR OS timing protection mechanism,” in *Proc. of the 14th IEEE International Conference on Emerging Techonologies and Factory Automation (ETFA)*, 2009.

[9] D. Reinhardt and G. Morgan, “An Embedded Hypervisor for Safety-Relevant Automotive E/E-Systems,” in *Proc. of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2014.

[10] *AUTOSAR (AUTomotive Open System ARchitecture)*, AUTOSAR. [Online]. Available: http://www.autosar.org/

[11] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings, “Fixed Priority Pre-emptive Scheduling: An Historical Perspective,” *Real-Time Systems*, vol. 8, no. 2-3, pp. 173–198, Mar./May 1995.

[12] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[13] J. Lehoczky, L. Sha, and Y. Ding, “The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior,” in *Proc. of Real Time Systems Symposium (RTSS)*, 1989.

[14] D. de Niz, K. Lakshmanan, and R. Rajkumar, “On the Scheduling of Mixed-Criticality Real-Time Task Sets,” in *Proc. of the 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[15] C. Ficek, N. Feiertag, and K. Richter, “Applying the AUTOSAR timing protection to build safe and efficient ISO 26262 mixed-criticality systems,” in *Proc. of Embedded Real Time Software and Systems ($ERTS^2$)*, 2012.

[16] S. Vestal, “Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance,” in *Proc. of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, 2007.

[17] L. Sha, J. P. Lehoczky, and R. Rajkumar, “Solutions for Some Practical Problems in Prioritized Preemptive Scheduling,” in *Proc. of the 7th IEEE Real-Time Systems Symposium (RTSS)*, 1986.

[18] C. Ficek, M. Sebastian, N. Feiertag, K. Richter, M. Jersak, and K. Schmidt, “Software Architecture Methods and Mechanisms for Timing Error and Failure Detection According to ISO 26262: Deadline vs. Execution Time Monitoring,” in *Proc. of the SAE World Congress*, no. 2013-01-0174, 2013.

[19] S. K. Baruah, A. Burns, and R. I. Davis, “Response-Time Analysis for Mixed Criticality Systems,” in *Proc. of the 32nd Real-Time Systems Symposium (RTSS)*, 2011.

[20] S. Baruah, A. Burns, and R. I. Davis, “An Extended Fixed Priority Scheme for Mixed Criticality Systems,” in *Proc. of Real-Time Mixed Criticality Systems (ReTiMiCS)*, 2013.

[21] AUTOSAR, *Specification of Operating System*, AUTOSAR Release 4.2.1, Document ID 034, 2014.

[22] A. Monot, N. Navet, B. Bavoux, F. Simonot-Lion *et al.*, “Multicore scheduling in automotive ECUs,” in *Proc. of Embedded Real Time Software and Systems ($ERTS^2$)*, 2010.

[23] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization,” *Computers, IEEE Transactions on*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.

[24] R. Kirner and P. Puschner, “Classification of WCET Analysis Techniques,” in *Proc. of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2005.

[25] J. Gustafsson and A. Ermedahl, “Experiences from Applying WCET Analysis in Industrial Settings,” in *Proc. of the10^{th} IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2007.

[26] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, “Applying new scheduling theory to static priority pre-emptive scheduling,” *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, Sep. 1993.

[27] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, “Mixed-criticality scheduling on multiprocessors,” *Real-Time Systems*, vol. 50, no. 1, pp. 142–177, Jan. 2014.

[28] AUTOSAR, *Guide to Multi-Core Systems*, AUTOSAR Release 4.1 Rev 3, Document ID 631, 2014.

[29] G. Morgan and A. Borg, “Multi-core Automotive ECUs: Software and Hardware Implications,” ETAS, Tech. Rep., 2009.

[30] “Freescale MPC564xL: Qorivva 32-bit MCU for Chassis and Safety Applications,” http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC564xL.

[31] D. A. Wheeler, “SLOCCount: Tools for counting physical Source Lines of Code,” http://www.dwheeler.com/sloccount/.

[32] “GNU Binutils,” http://www.gnu.org/software/binutils/.

[33] T. Piper, S. Winter, P. Manns, and N. Suri, “Instrumenting AUTOSAR for Dependability Assessment: A Guidance Framework,” in *Proc. of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.