공학석사학위논문

Real-Time Fault Tolerance using Performance/Lockstep Switch

퍼포먼스/락스텝 스위칭을 활용한 실시간 장애 허용 시스템

2023년 08월

서울대학교 대학원
컴퓨터공학부
전 해 주

Real-Time Fault Tolerance using Performance/Lockstep Switch

퍼포먼스/락스텝 스위칭을 활용한 실시간 장애 허용 시스템

지도교수 이 창 건

이 논문을 공학석사 학위논문으로 제출함

2023년 05월

서울대학교 대학원

컴퓨터공학부

전 해 주

전해주의 공학석사 학위논문을 인준함

2023년 06월

위 원 장	김 지 홍	(인)
부위원장	이 창 건	(인)
위 원	김 태 현	(인)

Abstract

Real-Time Fault Tolerance using Performance/Lockstep Switch

Haejoo Jeon
Department of Computer Science and Engineering
The Graduate School
Seoul National University

In modern automotive systems, ECU executes various tasks, from safety-critical to non-critical functions. Soft errors, caused by transient faults in hardware or software, can threaten the overall system reliability and safety. Therefore, it is essential to establish efficient resource allocation and scheduling strategies to mitigate the impact of these errors. This paper proposes three key ideas to address these issues: state-specific criticality, dynamic core execution mode, and dropping non-critical tasks when an error occurs. First, we introduce the concept of assigning task criticality based on different states and contexts, considering the specific physical environment in which the task is executed. Second, we propose a dynamic core execution mode that adjusts how each core executes based on the criticality of the task. Finally, we present a failure handling strategy that aborts the execution of non-critical tasks in the event of a failure and ensures the re-execution of critical tasks. Experimental results show that these ideas can significantly reduce the number of required cores. We also discuss potential extensions of this research.

keywords : Real-Time System, Fault Tolerance, Dynamic Switch

Student Number : 2021-20935

Contents

List of Figures

1 Introduction

In recent years, the exponential growth of artificial intelligence(AI) and rapid advancements in semiconductor processing technologies have accompanied it. This synergistic progress has sparked significant interest in autonomous driving, where the execution of numerous AI tasks on high-performance semiconductors has become a reality. However, ensuring safety becomes essential for autonomous driving to achieve widespread commercialization [1]. Among the myriad challenges that hinder this objective, a prominent one is the occurrence of transient hardware bit-flips known as soft errors. These soft errors are induced by the impact of energetic particles, such as neutrons, on the transistors of integrated circuits [2]. Importantly, these soft errors can manifest regardless of the completeness or defectiveness of the underlying hardware or software components, making them an unavoidable concern in system design. Anticipating and preventing soft errors in advance is difficult due to their inherent probabilistic nature.

Moreover, the rate of soft errors is expected to increase exponentially as semiconductor technologies advance. Previous researches, such as [3, 4], have provided empirical evidence demonstrating that the frequency of soft errors escalates with the increasing complexity of semiconductors, diminishing transistor sizes, and reduced operating voltages. Also, it has also been suggested that these soft errors could be life-threatening especially in autonomous driving [5]. To mitigate the adverse effects of soft errors and enhance the resilience of autonomous driving systems, redundancy has emerged as a prevalent technique. Redundancy-based approaches aim to detect and recover from errors by em-

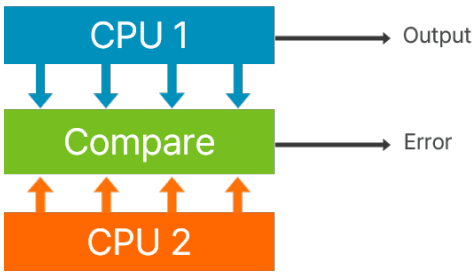

Figure 1: Lockstep

ploying duplicate or triplicate instances of critical components. The lockstep approach has gained widespread adoption in real-time systems where integrity and strict correctness are important [6]. Lockstep execution involves running multiple replicas of a task on separate cores and comparing their outputs at each cycle. If a difference is detected, an exception is raised, indicating the presence of a potential error. The lockstep approach offers exceptional accuracy, with an error detection rate exceeding 99% [7]. However, it comes at the cost of inefficient resource utilization and the underutilization of parallelism in multicore systems. In autonomous driving, where ECUs(Electronic Control Units) must execute computationally intensive AI tasks, failing to harness the parallelism offered by multicore architectures can lead to bad resource utilization and reduced system performance.

Recognizing the limitations of traditional lockstep approaches and the need to leverage the parallelism inherent in multicore systems, a novel semiconductor technology has emerged. This technology enables dynamic switching between lockstep mode and performance mode, ensuring the stability of critical tasks while fully utilizing the parallelism of non-critical tasks. Companies like ARM

have initiated hardware-level support for this feature, facilitating the development and commercializing of ECUs with performance/lockstep switch functionality. These ECUs enable seamless transitions between lockstep mode, which maximizes error detection accuracy, and performance mode, which optimizes multicore parallelism to enhance system performance.

In light of these advancements, this study presents a comprehensive scheduling framework specifically tailored to the demands of autonomous driving systems. The proposed framework facilitates dynamic mode switching between lockstep mode, with its high error detection rate, and performance mode, enabling efficient multicore parallelism utilization. By leveraging the unique capabilities of newly emerging ECUs, this framework aims to optimize resource utilization, enhance system efficiency, and ensure autonomous driving systems' reliable and safe operation.

2 Background

In autonomous mobility applications, such as autonomous vehicles, Urban Air Mobility (UAM), and robotics, achieving accurate and error-free performance within specified deadlines is of utmost importance. However, due to various unpredictable factors beyond our control, errors can occur, potentially threatening the integrity of critical tasks. To address this challenge, researchers and engineers have developed a range of error detection and recovery methodologies to ensure reliable task execution in the presence of errors. Redundancy, as a prevalent error detection technique, has garnered significant attention, with lockstep emerging as a widely adopted approach in autonomous vehicles due to its exceptional error detection rate exceeding 99%. Other software-based techniques, such as control flow check (CFC) [8], valid range check [9], and SW-based instruction-level redundant execution [10, 11], offer alternatives that utilize a single core but may exhibit lower performance. System designers must select the appropriate method considering the desired error detection rate and resource utilization trade-offs.

Functional safety standards for the automotive industry, including autonomous driving, are documented in ISO 26262. This internationally recognized standard establishes a comprehensive framework for developing and evaluating safety-critical systems in road vehicles. ISO 26262 covers critical hardware and software design, verification, and validation. The standard defines a risk classification system known as Automotive Safety Integrity Level (ASIL) to assess the safety risks associated with specific functions or systems. ASIL ratings are categorized into four levels, taking into account severity, expo-

sure, and controllability criteria [12]. Notably, highest level ASIL D requires a single-point fault metric (SPFM) of at least 99% to mitigate safety risks effectively.

Previous studies have pursued diverse approaches to tackle the challenges posed by varying task criticalities. For instance, [13] suggests distinct behaviors for tasks with different levels of criticality, effectively addressing their requirements. Similarly, [14] introduces a four-mode model that considers fault tolerance and Quality of Service (QoS) considerations for low criticality tasks. In our study, we adopt a hierarchical scheduling framework that facilitates the implementation of various error detection methods within a multicore environment.

Resource interfaces are vital in system design within the hierarchical scheduling framework, enabling seamless communication and efficient resource sharing between parent and child schedulers. At the parent level, the resource interface allocates resources to child schedulers based on their specific requirements. It empowers the parent scheduler to effectively coordinate resource allocations, ensuring optimal resource utilization across the system. At the child level, the resource interface provides child schedulers with a means to request resources from the parent scheduler. Through this interface, child schedulers can communicate their resource requirements to the parent scheduler, securing the necessary resources to complete their tasks successfully.

Resource interfaces are pivotal in a hierarchical scheduling framework's efficiency and reliability. By facilitating efficient communication and resource allocation between upper and lower-level schedulers, these interfaces ensure

that tasks are completed within their designated timelines while maintaining system reliability.

In our study, we leverage the Periodic Resource Model(PRM) technique to allocate a given resource hierarchically. PRM introduces two parameters, Pi and Theta, which guarantee a specified resource supply Θ within a defined time frame Π. As presented in [15], Shin & Lee has proposed supply and demand functions for PRM schedulability tests. Additionally, [16] extends the applicability of PRM to multicore systems and defines corresponding supply and demand functions accordingly. To establish the demand function, we draw inspiration from the demand model introduced in G-EDF by BAR [17], employing a similar formula to define the demand calculation.

3 Proposed Idea

3.1 Adaptive Dynamic Criticality Level Decisions

In a mixed-criticality system, the criticality of tasks is various. Some tasks may have catastrophic consequences if they encounter errors, while others may tolerate faults without significant impact. Traditionally, task criticality is determined without considering the specific physical environment in which the tasks are executed. However, when we examine real-world scenarios, such as autonomous driving, it becomes evident that the criticality of a task can vary depending on the context.

Let us consider the task of pedestrian detection as an example. The criticality of this task would vary depending on the specific driving scenario. For instance, when performed on a crosswalk with a high density of pedestrians, accurately detecting pedestrians becomes of utmost importance. On the other hand, in a highway setting where pedestrians are generally absent, the criticality of pedestrian detection would be significantly lower. Similarly, when evaluating a suspension system designed to reduce body sway, the criticality of this task would be higher on a highway where even minor movements can evoke a sense of threat due to high speeds, compared to a crosswalk. This state-dependent approach to defining task criticality takes into consideration not only the geographical characteristics of the driving situation but also the physical environment. For instance, the criticality of windshield wipers may vary between sunny weather conditions and heavy rain conditions. By incorporating each task's specific state and context, rather than relying solely on

worst-case scenarios, this state-dependent method allows for a precise assignment of criticality levels to tasks, facilitating resource optimization.

	Pedestrian Detection	Rollover Protection	Body & Convenience	Wiper
Worst Case	ASIL D	ASIL D	ASIL A	ASIL B
Crosswalk	ASIL D	ASIL B	ASIL A	ASIL B
Highway	ASIL B	ASIL D	ASIL A	ASIL B
Downpour	ASIL D	ASIL B	ASIL A	ASIL C

Table 1: Various criticality with driving states

By considering the actual physical environment in which tasks are executed, we can optimize resource utilization while ensuring task safety. In the Table 1, where a specific task is used as an example, the criticality of tasks is ranked differently based on different states. By determining the minimum number of cores required for each state in this situation, we can identify the optimal number of cores needed to accommodate the varying criticality levels. This approach enables us to allocate resources efficiently, matching them to the specific requirements of each state. Consequently, by incorporating state-dependent criticality analysis into task scheduling, we can achieve resource savings while maintaining the necessary level of task safety and reliability.

3.2 Dynamic Core Execution Mode

To reduce the number of cores required for a single state, we propose the dynamically convertible lockstep/performance mode. The conventional approach

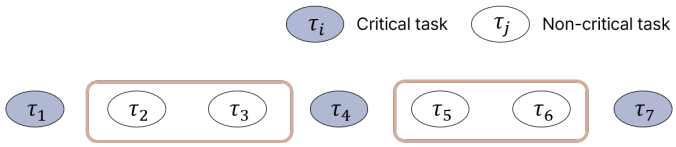

Figure 2: Example taskset

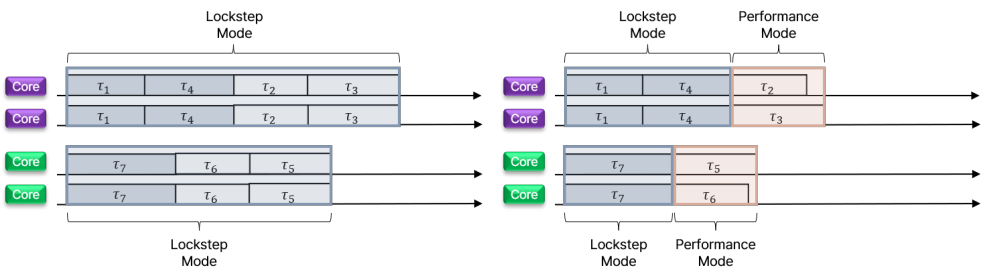

Figure 3: (a) Lockstep, (b) Dynamic core execution mode

employs redundant execution using lockstep for all tasks, regardless of their criticality. However, this method often leads to excessive resource utilization, potentially compromising real-time performance or resulting in unnecessary resource allocation. In our proposed method, we aim to minimize the number of cores needed by adapting the execution method of each core based on the criticality of the task.

The existing method in Figure 3 (a) executes all tasks in lockstep mode on the dedicated core, without distinguishing between non-critical and critical tasks. This approach is inefficient since it allocates twice as many resources to relatively unimportant non-critical tasks. Our proposed method, illustrated in Figure 3 (b), involves grouping non-critical tasks and executing them in performance mode, while critical tasks are executed in lockstep mode. It is much efficient since our method only runs critical tasks in lockstep mode. The

grouping of non-critical tasks and determining the number of groups are described in Section 3.3. To implement this scheduling approach, we leverage component-based scheduling, where the component-level scheduler handles the scheduling of critical and non-critical task groups, while the task-level scheduler focuses on scheduling the non-critical tasks within each group.

We ensure complete safety by employing the lockstep method for critical tasks, while utilizing the performance mode for non-critical tasks to enhance parallelism. As illustrated in Figure 2, τ_2, τ_3 and τ_5, τ_6 are grouped and executed in parallel on two cores in performance mode. Conversely, since tasks τ_1, τ_4, and τ_7 are critical, they are executed using the lockstep mode to ensure reliability. Through this mechanism, we can guarantee the reliability of critical tasks and improve overall schedulability by executing tasks efficiently. By dynamically adapting the execution mode based on task criticality, we can achieve a balance between safety and resource optimization, leading to improved performance and resource utilization in mixed-criticality systems.

3.3 Drop Non-critical Task with Fault

The lockstep method provides immediate detection of faulty situations, especially soft error. When a task fails to perform a normal computation due to a soft error, it is crucial to recompute the task within its deadline to ensure the successful behavior of task. Typically, fault-recovery strategies involve allocating an extra budget for re-run, as depicted in Figure 4 (a). However, reserving additional resources for low probability soft-errors (e.g., once every few hours) can significantly waste resources [3]. To mitigate the excessive allo-

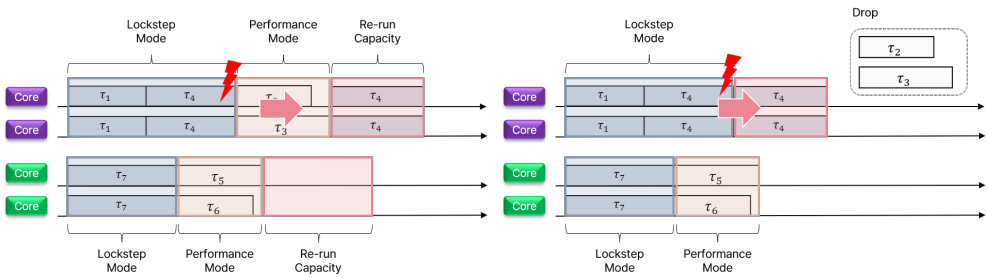

Figure 4: (a) Extra budget, (b) Drop non-critical task

cation of cores caused by resource waste, we propose an alternative approach that drop the execution of a group of non-critical tasks in the event of a fault and utilizae the available resources to re-run the faulted critical task.

By abandoning the execution of the non-critical task group, we temporally prioritize the re-run of the critical task that experienced the fault. While the discontinuation of non-critical tasks may lead to a temporary degradation in quality, it does not result in catastrophic consequences since these tasks are relatively less significant. Furthermore, these faults are transient and infrequent, allowing the subsequent job instances in the non-critical task group to resume normal execution without disruptions.

4 Real-Time Fault Tolerance Scheduling Framework

4.1 Task Model

In this study, we aim to determine the minimum number of cores required for a given task set to ensure safe execution in the presence of soft errors. Let's consider a task set, $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$, where each task τ_i is represented by (p_i, e_i, C_i). Here, p_i denotes the period of task τ, e_i represents the execution time, and C_i indicates the criticality of the task, defined as $C_i = [c_{i1}, c_{i2}, ..., c_{ij}]$. c_{ij} represents the criticality of task τ_i in state j. We assumed all tasks have an implicit deadline, which means the deadline of the task is the same as the period, p_i. And it is assumed that the system designer provides the types of states and their corresponding criticalities. To verify the schedulability of the system, we first aim to determine the schedulability of individual components by dividing the tasks into components.

As shown in Figure 5, the parent-level scheduler is responsible for scheduling multiple workloads, where each workload becomes a resource supply for scheduling tasks at the child level. Henceforth, we will refer to schedule at the parent level as component scheduling and at the child level as task scheduling. The utilization of a task set, denoted as u(task set), represents the sum of utilizations of all tasks within the set and can be calculated as $\sum(e_i/p_i)$.

Therefore, we aim to discuss the schedulability of the proposed scheduling framework by conducting component-level schedulability tests and task-level schedulability tests under fault-free conditions, and component-level schedulability tests under fault occurrence. There is no need to discuss the task-level

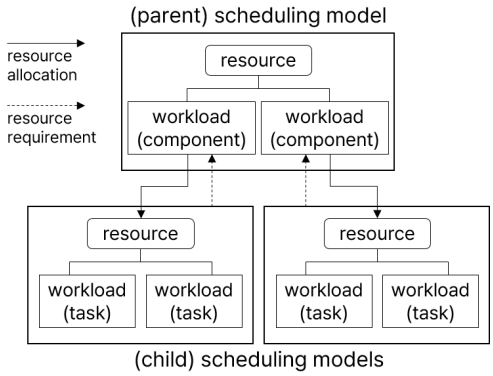

Figure 5: Hierarchical scheduling framework

schedulability test under fault occurrence. Our proposed idea 3, which involves dropping non-critical tasks in the presence of faults, eliminates the necessity of conducting a separate schedulability check for this group of tasks. Consequently, in the event of a fault, non-critical tasks are immediately dropped from execution. As a result, the execution of non-critical tasks does not need to undergo an additional schedulability assessment, as per our proposed approach.

4.2 EDF Utilization-based Schedulability Test

First, we consider the critical tasks as a single component running on two bound cores for the given task set. Through this process, we can group the critical tasks performed in lockstep on two cores and the non-critical tasks performed on a single core together at the component level. It is necessary to allocate tasks to the bound cores. This problem is commonly known as the bin-packing problem, which is an NP-hard problem. We use the well-

known heuristic worst-fit to allocate tasks. Once all critical tasks are allocated, we proceed with an utilization-based schedulability test based on the EDF (Earliest Deadline First) algorithm. If the sum of utilizations of all critical tasks allocated to the bound cores is less than 1.0, we can determine that it is schedulable. In this case, the utilization-based schedulability check is typically a single core based test method. However, since we execute the duplicate tasks of lockstep on the bound cores simultaneously, a single core based test method can be applied.

4.3 Time Demand Analysis

When there are a total of m cores, after allocating critical tasks to $\frac{m}{2}$ bound cores, we can allocate non-critical tasks to the remaining utilization space. This becomes a bin-packing problem of assigning non-critical tasks to $\frac{m}{2}$ bins. The difference from the problem in section 4.2 lies in the methodology of schedulability testing. Allocating non-critical tasks to $\frac{m}{2}$ bins and determining if each task is schedulable within its respective bin, or more precisely, within the supply of its corresponding PRM, can be resolved using time demand analysis.

Regarding supply, we modified the supply function proposed in [15]. The PRM $\Gamma = (\Theta, \Pi)$ can provide resources for Π units of time every Θ units of time. Our PRM model, being executed on the bound dual cores rather than a single core, effectively provides 2Π units of resources. Figure 6 shows an example when supply is given as the worst case. The supply bound function in the worst-case scenario and the linear supply bound function for convenience are as follows.

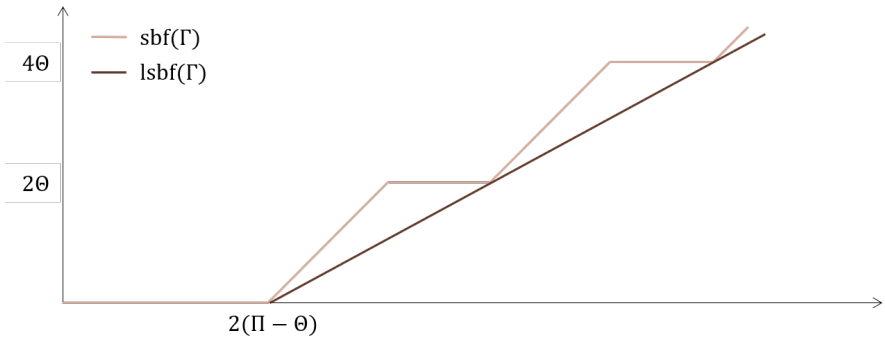

Figure 6: Supply example on worst case

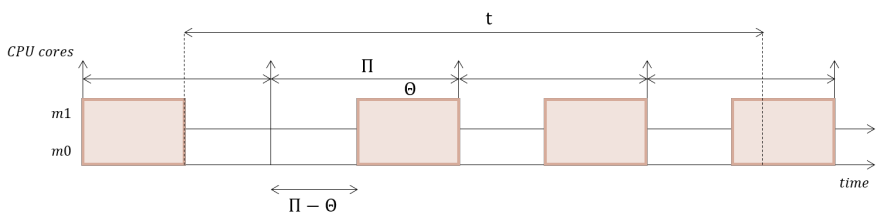

Figure 7: Supply graph on worst case

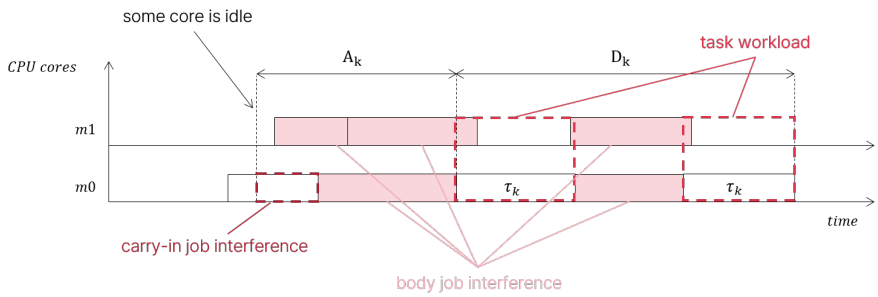

Figure 8: Demand example

The linear supply bound function is not exact, but has the advantage of being more computationally convenient.

Therefore, in EDF, sbf(supply bound function) and lsbf(linear supply bound function) are given by equation 1, 2, which is equivalent to multiplying the sbf and lsbf by a factor of 2. The graph of these functions is shown in Figure 7.

$$\mathrm{sbf}_\Gamma(t) = 2 \cdot (\lfloor \frac{t - (\Pi - \Theta)}{\Pi} \rfloor \cdot \Theta + \epsilon_s), \tag{1}$$

where $\epsilon_s = \max(t - 2 \cdot (\Pi - \Theta) - \Pi \lfloor \frac{t-(\Pi-\Theta)}{\Pi} \rfloor, 0)$

$$\mathrm{lsbf}_\Gamma(t) = 2 \cdot \frac{\Theta}{\Pi}(t - 2(\Pi - \Theta)) \tag{2}$$

We used the demand from [16]. Figure 8 is simple explanation of demand function. Let $W_i(t)$ denote workload bound for task τ_i at time t and let $CI_i(t)$ denote the carry-in demand. Then workload bound $W_i(t)$ defined as

$$W_i(t) = N_i(t)e_i + CI_i(t) \tag{3}$$

, where $N_i(t) = \lfloor \frac{t}{p_i} \rfloor$ and $CI_i(t) = \min\{e_i, \max\{0, t - N_i(t)p_i\}\}$. Please note that we assume an implicit deadline, so some notation may differ from [16, 17].

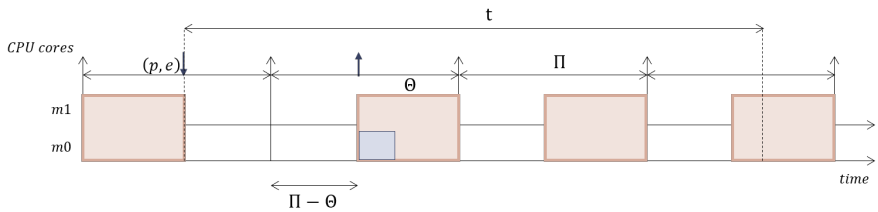

Figure 9: Minimum theta value

Then, interference can be defined as

$$\begin{aligned} \hat{I}_{i,2} &= \min\{W_i(A_k + p_k) - CI_i(A_k + p_k), A_k + p_k - e_k\} \text{ for all } i \neq k, \\ \hat{I}_{k,2} &= \min\{W_i(A_k + p_k) - e_k - CI_i(A_k + p_k), A_k\} \end{aligned} \tag{4}$$

Then the demand of τ_k is

$$\text{dem}(A_k + p_k) = \sum_{i=1}^{n} \hat{I}_{i,2} + \max(\overline{I}_{i,2} - \hat{I}_{i,2}) + e_k \tag{5}$$

for all critical tasks $\tau_k \in W_c$ and all $A_k \geq 0$.

Using these supply and demand values, we determined the parameters Π and Θ of the PRM. First, the period Π of the PRM component was set as $\min_{\tau_i} \frac{p_i - e_i}{2}$. As shown in Figure 9, for the internal tasks to execute without starvation, even in the worst-case scenario where PRM supply is provided at its worst case, the condition $2(\Pi - \Theta) + e \leq p$ must hold for the tasks. At this point, Θ was excluded and left as an undetermined positive value. The following approach was taken to find the sub-optimal theta.

Theorem 1. Let non-critical tasks in the core denote as W_n, and critical tasks as W_c, then PRM utilization of the bound is

$$\frac{\Theta^+}{\Pi}, \text{ where } \Theta^+ = \max_{0<t<2\text{LCM}_{W_n}} \left(\frac{\sqrt{(t - 2\Pi)^2 + 4\Pi \text{dbf}_{W_n}(t)} - (t - 2\Pi)}{4} \right) \tag{6}$$

Proof. To make tasks in W_n schedulable,

$$\mathrm{dbf}_{W_n}(t) \leq \mathrm{lsbf}_{\Gamma}(t) = 2 \cdot \frac{\Theta}{\Pi}(t - 2(\Pi - \Theta)) <= \mathrm{sbf}_{\Gamma}(t) \tag{7}$$

From Eq. 7, we can derive the below equation.

$$\Theta \geq \frac{\sqrt{(t - 2\Pi)^2 + 4\Pi \mathrm{dbf}_{W_n}(t)} - (t - 2\Pi)}{4} \tag{8}$$

□

From theorem 1 for $\frac{m}{2}$ bins, we can consider the non-critical tasks as schedulable within their respective groups. This determination is independent of the schedulability of the PRM component itself and assumes that the PRM component executes properly as a result of being schedulable. Thus, we need to test whether the PRM component can execute even at the component level, or in other words, on the bound cores.

Theorem 2**.** Assume critical task set W_c and PRM $\Gamma = (\Pi, \Theta)$ has allocated on bound core. Component-level of core is schedulable when

$$\sum_{\tau_i \in W_c} \frac{e_i}{p_i} + \frac{\Theta}{\Pi} \leq 1.0 \tag{9}$$

Proof. Despite the physical composition of the bound cores as dual cores, the simultaneous execution of all components using both cores enables us to consider them as a single core for our analysis. Within the EDF scheduling framework, tasks are considered schedulable if the sum of their utilizations is less than 1.0. Therefore, based on equation Eq. 9, we can conclude that both the critical tasks and the PRM component are schedulable. □

The PRM supplies resources to internal tasks while acting as a single task when regarded from a higher level. Therefore, we applied the utilization-based schedulability test as in section a. Since we originally derived Π and Θ within the remaining utilization bound on a single core, the PRM component itself is also schedulable.

4.4 Response Time Analysis

If a fault is detected to have actually occurred, the execution of non-critical tasks must be dropped immediately, and the critical task should be re-executed. Therefore, it is necessary to verify that the re-execution in the event of a fault can be completed within the deadline. For this purpose, EDF single-core response time analysis can be utilized. Unlike the well-known rate monotonic (RM) scheduling, EDF does not have a predetermined critical instant. Therefore, it is not possible to calculate the response time with the condition that all tasks are released simultaneously, as in RM. To determine the critical instant, Spuri [18] employs the sliding window method, which involves considering each task individually to identify the point at which the interference from other tasks reaches its maximum, indicating the occurrence of the critical instant. If a soft error occurs, the affected critical task must be re-executed within the deadline. Non-critical tasks are dropped during this process, so they need not be considered. However, other critical tasks that did not experience errors must still adhere to their timing constraints, regardless of re-execution. Therefore, we considered re-execution as a form of interference. In the iterative response time formula, we added the fixed interference amount that the

task can receive during that time period.

Let $s_i(a)$ denote the time in which the first instance of τ_i arrived. Also, then the interference from higher priority workload $HW_i(a,t)$ is

$$HW_i(a,t) = \sum_{j \neq i \ \& \ p_j \leq a+p_i} \min\left\{ \left\lceil \frac{t}{p_j} \right\rceil, 1 + \left\lfloor \frac{a+p_i-p_j}{p_j} \right\rfloor \right\} e_j + \delta_i(a,t) e_i \quad (10)$$

where $s_i = a - \left\lfloor \frac{a}{p_i} \right\rfloor p_i$ and

$$\delta_i(a,t) = \begin{cases} \min\left\{ \left\lceil \frac{t-s_i(a)}{p_i} \right\rceil, 1 + \left\lceil \frac{a}{p_i} \right\rceil \right\} & \text{if } t > s_i(a) \\ 0 & \text{otherwise} \end{cases} \quad (11)$$

From Eq. 10, the busy period $L_i(a)$ could be defined.

$$\begin{cases} L_i^{(0)}(a) & = \sum_{j \neq i \ \& \ p_j \leq a+p_i} e_j + I_{\{s_i(a)=0\}} \cdot e_i \\ L_i^{(m+1)}(a) & = HW_i\left(a, L_i^{(m)}(a)\right) + \text{rerun capacity} \end{cases} \quad (12)$$

where

$$I_{\{s_i(a)=0\}} = \begin{cases} 1 & \text{if } s_i(a) = 0, \\ 0 & \text{otherwise} \end{cases}$$

Finally, worst-case response time of τ_i is

$$r_i = \max_{a \geq 0} \left\{ \max\{e_i, L_i(a) - a\} \right\} \quad (13)$$

In the context of single core EDF response time analysis (RTA), the precise critical instant is not predetermined. However, it is known that the critical instant occurs when all other tasks are simultaneously released. To determine

the critical instant, we employ the sliding window method, which involves considering each task individually to identify the point at which the interference from other tasks reaches its maximum, indicating the occurrence of the critical instant.

5 Experiments

In the experimental section of our research, we evaluate three key contributions.

- Adaptive dynamic criticality level decisions
- Dynamic core execution mode
- Drop non-critical tasks with fault

Specifically, we focus on evaluating two aspects: The impacts of criticality per state and the scheduling mechanism's impacts. These two contributions, the "Dynamic core execution mode" and the "Drop non-critical task with fault" techniques, are evaluated together to assess their effectiveness compared to existing approaches.

5.1 Impacts of Dynamic Core Execution Mode and Drop Non-critical Task with Fault

We investigated the impact of Idea 2 and Idea 3 on the required number of cores. The detailed parameters are as follows.

- Total iteration number: 1,000
- The number of tasks in task set: 10
- Period: log-uniform on [10, 500]
- Max utilization of task: 0.3

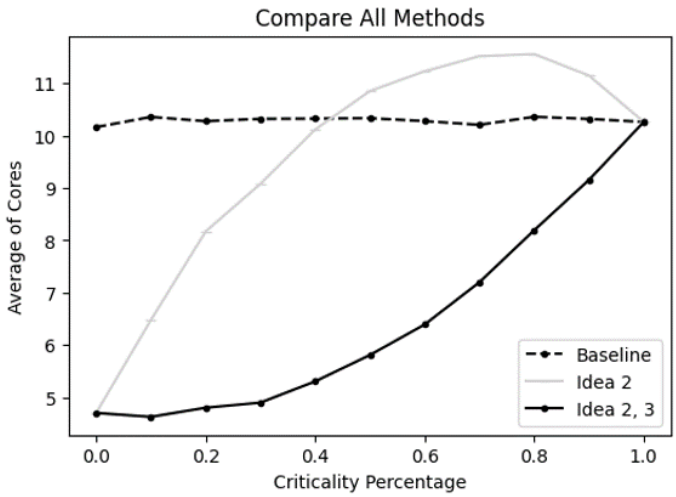

Figure 10: Compare method experiment

Similar to the previous experiment, we compared the baseline with the application of only idea 2, and the application of both idea 2 and idea 3. Except for the baseline, we observed that as the percentage of critical tasks in the task set increased, the required number of cores seam to be increased. However, we noticed that the baseline consistently required many cores regardless of the criticality percentage. In the case of applying only idea 2, we observed that it required more cores when the critical task ratio was moderately higher compared to the scenario where the task set consisted entirely of critical tasks. This can be attributed to the overhead of grouping non-critical tasks into PRMs. If there are not enough non-critical tasks assigned to the PRM to take advantage of parallelism within the PRM, the loss due to grouping outweighs the benefits of parallelization. However, in general, where the ratio of critical tasks is lower than that of non-critical tasks, both methods incorporating our

ideas required fewer cores than the baseline.

This finding demonstrates that our proposed ideas can effectively reduce the required number of cores, especially when the critical task ratio is lower than that of non-critical tasks. Nonetheless, it is important to carefully consider the overhead introduced while grouping non-critical tasks and the available parallelism within PRMs when determining the optimal resource allocation strategy. Further analysis and optimization of these factors can lead to more efficient resource utilization in mixed-criticality systems.

5.2 Impacts of Criticality Per State

In the second experiment, we experimented to investigate the impact of assigning criticality based on states on the required number of cores. The experiment was conducted to determine the required number of cores based on the ratio of critical tasks within the task set and the number of states. These parameters were tested using the given task set to analyze the results.

- Total iteration number: 1,000
- The number of tasks in task set: 40
- Period: uniform on [10, 100]
- Max utilization of task: 0.1

As shown in Figure 11 (a), when applying idea 1, it exhibited a more attenuated growth trend as the ratio of critical tasks within the task set increased compared to the baseline. Additionally, in all cases, it was observed that idea 1

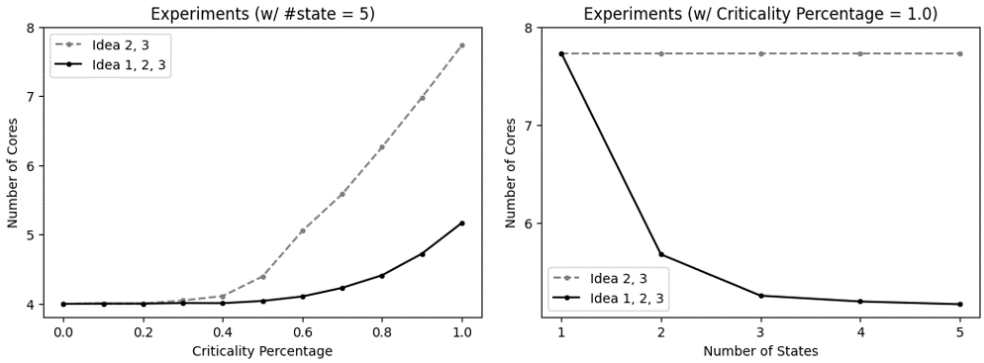

Figure 11: (a) criticality percentage experiment, (b) state experiment

required fewer states compared to the baseline. In Figure 11 (b), the minimum required number of cores, which varies depending on the number of states set, can be observed. Without idea1 required the same number of cores regardless of the number of states. At the same time, ours showed a reduction in the required number of cores as the number of states increased, indicating a more refined criticality differentiation.

6 Conclusion

In this paper, we have presented a comprehensive approach for resource optimization and reliability in mixed-criticality systems. Our contributions include the consideration of adaptive dynamic criticality level decisions, dynamic core execution mode, and dropping non-critical tasks with faults. Through experimental evaluation, we have demonstrated the effectiveness of our approach in reducing the required number of cores while maintaining task safety and reliability.

By incorporating state-dependent criticality analysis, we have shown that the criticality of tasks can vary based on the specific physical environment in which they are executed. This approach enables us to allocate resources efficiently, matching them to the particular requirements of each state. Moreover, our dynamic core execution mode allows for adapting execution methods based on task criticality, improving resource utilization and performance. Additionally, by dropping non-critical tasks in the event of a fault and prioritizing the rerun of critical tasks, we have mitigated resource waste caused by excessive allocation for low probability soft errors. This fault-handling strategy ensures the reliability of critical tasks while minimizing disruptions to non-critical tasks.

Furthermore, our scheduling framework can be improved by applying various error detection methods, not just simple lockstep. Considering these different error detection methods can better reflect real-world situations. For example, you can more appropriately meet the error detection rates required by the four ASIL levels.

In conclusion, our research has contributed to resource optimization in mixed-criticality systems. By incorporating our proposed ideas, we have demonstrated the potential for reducing the required number of cores while maintaining task safety and reliability. Integrating various error detection methods and exploring adaptive approaches are promising avenues for future research, enabling further enhancements in reliability, resource utilization, and schedulability in mixed-criticality systems.

References

[1] Phil Koopman. A case study of toyota unintended acceleration and software safety. Presentation. Sept, 2014.

[2] James F Ziegler and William A Lanford. Effect of cosmic rays on computer memories. Science, 206(4420):776–788, 1979.

[3] Premkishore Shivakumar, Michael Kistler, Stephen W Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In Proceedings International Conference on Dependable Systems and Networks, pages 389–398. IEEE, 2002.

[4] Robert Baumann. Soft errors in advanced computer systems. IEEE design & test of computers, 22(3):258–266, 2005.

[5] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–12, 2017.

[6] Bowen Zheng, Yue Gao, Qi Zhu, and Sandeep Gupta. Analysis and optimization of soft error tolerance strategies for real-time systems. In 2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS), pages 55–64. IEEE, 2015.

[7] Christian El Salloum, Andreas Steininger, Peter Tummeltshammer, and Werner Harter. Recovery mechanisms for dual core architectures. In 2006 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pages 380–388. IEEE, 2006.

[8] Emmanuel Touloupis, James A Flint, Vassilios A Chouliaras, and David D Ward. A fault-tolerant processor core architecture for safety-critical automotive applications. SAE transactions, pages 1–6, 2005.

[9] Abhishek Rhisheekesan, Reiley Jeyapaul, and Aviral Shrivastava. Control flow checking or not?(for soft errors). ACM Transactions on Embedded Computing Systems (TECS), 18(1):1–25, 2019.

[10] Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pages 237–246, 2011.

[11] Moslem Didehban, Aviral Shrivastava, and Sai Ram Dheeraj Lokam. Nemesis: A software approach for computing in presence of soft errors. In 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pages 297–304. IEEE, 2017.

[12] Philip Koopman and Michael Wagner. Challenges in autonomous vehicle testing and validation. SAE International Journal of Transportation Safety, 4(1):15–24, 2016.

[13] Risat Mahmud Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. Real-Time Systems, 50:509–547, 2014.

[14] Zaid Al-bayati, Jonah Caplan, Brett H Meyer, and Haibo Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 97–102. IEEE, 2016.

[15] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003, pages 2–13. IEEE, 2003.

[16] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In 2008 Euromicro Conference on Real-Time Systems, pages 181–190. IEEE, 2008.

[17] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In 28th IEEE International Real-Time Systems Symposium (RTSS 2007), pages 119–128. IEEE, 2007.

[18] Marco Spuri. Analysis of deadline scheduled real-time systems. PhD thesis, Inria, 1996.

요약(국문초록)

자율주행 분야에서 ECU는 안전에 중요한 기능부터 중요하지 않은 기능까지 광범위한 작업을 실행하게 된다. 그러나 소프트 에러로 인한 비트 플립 현상은 전체 시스템 신뢰성과 안전성을 위협할 수 있다. 따라서 이러한 오류의 영향을 없애기 위해 높은 정확도의 에러 감지율을 가진 방법론의 사용과 함께 효율적인 리소스 활용이 필요하다. 본 논문에서는 이러한 문제를 해결하기 위한 세 가지 핵심 아이디어, 즉 상태 별 중요도, 동적 코어 실행 모드, 오류 발생 시 중요하지 않은 태스크 실행 삭제를 제안한다. 첫째, 작업이 실행되는 특정 물리적 환경을 고려하여 다양한 상황에 따라 작업의 중요도를 할당하는 아이디어를 제안한다. 둘째, 태스크의 중요도에 따라 각 코어의 실행 방식을 전환하는 동적 코어 실행 모드를 제안한다. 마지막으로, 장애 발생 시 중요하지 않은 태스크의 실행을 중단하고 중요 태스크의 재실행을 데드라인 내에 완료하는 장애 처리 전략을 제시한다. 실험 결과를 통해 이러한 아이디어를 적용했을 때 작업 안전성을 유지하면서 필요한 코어 수를 크게 줄일 수 있음을 확인했다.

주요어 : 실시간 시스템, 장애 허용 시스템, 동적 전환
학 번 : 2021-20935