# Replica-Aware Co-Scheduling for Mixed-Criticality[*]

**Eberle A. Rambo[1] and Rolf Ernst[2]**

1 **Technische Universität Braunschweig, Braunschweig, Germany**
`rambo@ida.ing.tu-bs.de`
2 **Technische Universität Braunschweig, Braunschweig, Germany**
`ernst@ida.ing-tu-bs.de`

## Abstract

Cross-layer fault-tolerance solutions are the key to effectively and efficiently increase the reliability in future safety-critical real-time systems. Replicated software execution with hardware support for error detection is a cross-layer approach that exploits future many-core platforms to increase reliability without resorting to redundancy in hardware. The performance of such systems, however, strongly depends on the scheduler. Standard schedulers, such as Partitioned Strict Priority Preemptive (SPP) and Time-Division Multiplexing (TDM)-based ones, although widely employed, provide poor performance in face of replicated execution. In this paper, we propose the replica-aware co-scheduling for mixed-critical systems. Experimental results show schedulability improvements of more than 1.5x when compared to TDM and 6.9x when compared to SPP.

**1998 ACM Subject Classification** C.4 Performance of Systems

**Keywords and phrases** replicated execution, scheduling, fault tolerance, real-time systems

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2017.20

## 1 Introduction

Technology downscaling has increased the hardware's overall susceptibility to errors to the point where they became non-negligible [15]. Hence, current and future computing systems must be appropriately designed to cope with errors in order to provide a reliable service and correct functionality. Specially in the real-time mixed-criticality domain, where applications with different requirements and criticalities co-exist in the system, which must provide *sufficient independence* and prevent error propagation (e.g. timing, data corruption) between criticalities [17, 28]. Recent examples are complex applications such as Flight Management Systems (FMS) and Advanced Driver Assistance Systems (ADAS) in the avionics and automotive domains, respectively [17, 28]. In this paper, we address the timing aspect of software execution protected from soft errors.

Soft errors, more specifically Single Event Effects (SEEs), are transient faults abstracted as *bit-flips* in hardware and can be caused by alpha particles, energetic neutrons from cosmic radiation and process variability [11, 15]. Depending on where and when they occur, their impact on software execution range from masked (no observable effect) to a complete system crash [5, 8, 9]. To handle such errors, the approaches can vary from completely software-based to completely hardware-based. The former are able to cover only part of the errors [9, 8] and the latter result in costly redundant hardware [15], as currently seen in lock-step

[*] This work was supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 – `http://spp1500.itec.kit.edu`).

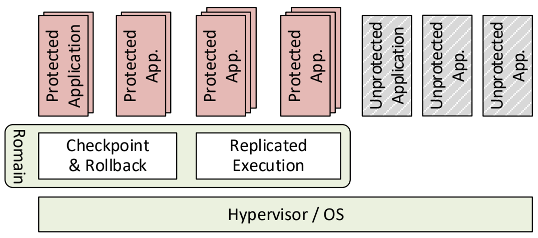


**Figure 1** ASTEROID's fault-tolerance architecture: the software side.

dual-core execution [21]. We focus on a more effective and efficient cross-layer approach, which distributes the tasks of detecting errors and recovering from them in different layers of software and hardware [15, 9, 8].

A cross-layer fault-tolerance solution for mixed-criticality has been developed in the ASTEROID project. It increases the reliability at a higher level of abstraction without resorting to hardware redundancy [3, 8]. ASTEROID's architecture is illustrated in Fig. 1. The reliable software execution is realized by the operating system service Romain [8]. Mixed-critical applications may co-exist in the system and are translated into protected and unprotected applications. Romain replicates the protected applications and manage their execution. Error detection is realized by a set o mechanisms whose main feature is the hardware assisted state comparison, which compares the replicas' state at certain points in time [3, 8]. Error recovery strategies can vary depending on whether the application is running in Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR) [3, 5].

The performance of replicated execution has been analyzed in [4] and revised in [2]. The work supports Partitioned Strict Priority Preemptive (SPP) scheduling, where tasks are mapped to arbitrary cores, and assumes a single error model. The authors found that SPP, although widely employed in real-time systems, provides very pessimistic response time bounds for replicated tasks. Depending on the interfering workload, replicated tasks executing serially (on the same core) present much better performance than when executing in parallel (on distinct cores). That occurs due to the long time that replicated tasks potentially have to wait on each core to synchronize and compare states before resuming execution. That leads to very low resource utilization and prevents the use of replicated execution in practice.

In this paper, we explore co-scheduling to provide small response times for replicated tasks without hindering the remaining unprotected tasks. Co-scheduling is a technique that schedules interacting tasks/threads to execute simultaneously on different cores [22]. It allows tasks/threads to communicate more efficiently by reducing the time they are blocked during synchronization. In contrast to SPP [4, 2], our approach drastically minimizes delays due to the implicit synchronization found in state comparisons. In contrast to gang scheduling [10], it rules out starvation and distributes the execution of replicas in time to achieve small response times of unprotected tasks. Finally, our approach differs from standard Time-Division Multiplexing (TDM) and TDM with background partition [18] in that all tasks have formal guarantees.

The major **contribution** of this paper is the replica-aware co-scheduling for mixed-critical systems. A formal Worst-Case Response Time (WCRT) analysis under a single error assumption is included. In contrast to related work, it supports different recovery strategies and accounts for the Network-on-Chip (NoC) communication delay and overheads due to replica management and state comparison. Experimental results with benchmark applications show an improvement on taskset schedulability of up to 6.9x when compared to SPP [2], and 1.5x when compared to a TDM-based scheduler.

## 2 Related Work

L4/Romain [8] is a cross-layer fault-tolerance approach that provides reliable software execution under soft errors. Romain provides protection at the application-level by replicating and managing the applications' executions as an operating system service. The error detection is realized by a set of mechanisms [3, 8, 9] whose main feature is the hardware assisted state comparison, which allows an effective and efficient comparison of the replicas' states. Pipeline fingerprinting [3] provides a checksum of the retired instructions and the pipeline's data path in every processor, detecting errors in the execution flow and data. The state comparison, reduced to comparing checksums instead of data structures, is carried out at certain points in time. It must occur at least when the application is about to externalize its state e.g. in a *syscall* [8]. The replica generated *syscalls* are intercepted by Romain, have their integrity checked and their replicas' states compared before being allowed to externalize the state [8].

Mixed-criticality, in the context of this paper, is supported with different levels of protection for applications with different criticalities and requirements (unprotected, protected with DMR[1]or TMR) and by ensuring that timing constraints are met even in case of errors. For instance, Romain provides different error recovery strategies [3, 5]:

- *DMR with checkpoint and rollback*: to recover, the replicas rollback to their last valid state and re-execute;
- *TMR with state copy*: to recover, the state of the faulty replica is replaced with the state of one of the healthy replicas.

In this work, we focus on the system-level timing aspect of errors affecting the applications. We assume thereby the absence of failures in critical components [9, 24], such as the Operating System (OS), the replica manager/voter (e.g. Romain) and interconnect (e.g. NoC), which can be protected as in [16, 26].

The WCRT of replicated execution has been analyzed in [4], where replicas are modeled as fork-join tasks in a system implementing Partitioned SPP. The work was later revised in [2] due to optimism in the original approach. The revised approach is used in this work. In that approach, with deadline monotonic priority assignment, where the priority of tasks decrease as their deadlines increase, replicated tasks perform worse when mapped in parallel than when mapped to the same core. This is due to the state comparisons during execution, which involves implicit synchronization between cores. With partitioned scheduling, in the worst-case, the synchronization ends up accumulating the interference from all cores to which the replicated task is mapped, resulting in poor performance in higher loads. On the other hand, mapping replicated tasks to the highest priorities results in long response times for lower priority tasks and rules out deadline monotonicity. The latter causes the unschedulability of all tasksets with at least one regular task whose deadline is shorter than the execution time of a replicated task.

Gang scheduling [10] is a co-scheduling variant that schedules groups of interacting tasks/threads simultaneously. It increases performance by reducing the inter-thread communication latency. The authors in [19] present an integration between gang scheduling and Global Earliest Deadline First (EDF), called the Gang EDF. They provide a schedulability analysis derived from the Global EDF's based on the sporadic task model. In another work, [12] shows that SPP Gang schedulers in general are not predictable, for instance, due to

[1] DMR *per se* can be used for system integrity only. However, DMR augmented with checkpointing and rollback enables recovery and can be used to achieve integrity and availability (state rollback followed by re-execution in both replicas) [3, 5].

priority inversions and slack utilization. In the context of real-time systems, gang scheduling has not received much attention.

TDM-based scheduling [18] is widely employed to achieve predictability and ensure temporal-isolation. Tasks are allocated to partitions, which are scheduled to execute in time slots. Partitions can span across several (or all) cores and can be executed at the same time. The downside of TDM is that it is not work-conserving and underutilizes system resources. A TDM variant with background partition [18] tackles this issue by allowing low priority tasks to execute in other partitions whenever no higher priority workload is executing. Yet, in addition to the high cost to switch between partitions, no guarantees can be given to tasks in the background partition.

In this work, we exploit co-scheduling with SPP to improve the performance of the system. Our work differs from [4] in that replicas are treated as gangs and are mapped with highest priorities, and are hence activated simultaneously on different cores. In contrast to gang-scheduling [10, 12] and to [4], the execution of replicas is distributed in time with offsets to compensate for the lack of deadline monotonicity thus allowing the schedulability of tasks with short deadlines. We further provide for the worst-case performance of lower priority tasks by allowing them to execute whenever no higher priority workload is executing. However, in contrast to [18], all tasks have WCRT guarantees. Moreover, we also model the state comparison and the on-Chip communication overheads, and although we use Romain as example, the model can also be applied to other approaches.

## 3 Preliminaries

In this work, we use the Compositional Performance Analysis (CPA) [14] to provide formal response time bounds. Let us introduce the system, task and error models.

### 3.1 System Model

The system consists of a standard NoC-based many-core composed of processing elements, simply referred to as cores.

There are two types of tasks in our system, as in [2]:

- *independent* tasks $\tau_i$: regular, unprotected tasks; and
- *fork-join* tasks $\Gamma_i$: replicated, protected tasks.

The system implements partitioned scheduling, where the operating system manages tasks statically mapped to cores. The mapping is assumed to be given as input. The scheduling policy is a combination of SPP and gang scheduling. When executing only independent tasks, the system's behavior is identical to Partitioned SPP, where tasks are scheduled independently on each core according to SPP. It differs from SPP, when scheduling fork-join tasks.

Fork-join tasks are mapped with highest priorities, hence do not suffer interference from independent tasks, and execute simultaneously on different cores, as in gang scheduling. Note that deadline monotonicity is therefore only partially possible. To limit the interference to independent tasks, the execution of a fork-join task is divided in smaller intervals called stages, whose executions are distributed in time. At the end of each stage, the states of the replicas are compared. In case of an error, i.e. states differ, recovery is triggered.

Fork-join stages are executed with static offsets [23] in execution slots. One stage is executed per slot. On a core with $n$ fork-join tasks, there are $n + 1$ execution slots: one slot for each fork-join task $\Gamma_i$ and one slot for recovery. The slots are cyclically scheduled in a cycle $\Phi$. The slot for $\Gamma_i$ starts at offset $\phi(\Gamma_i)$ relative to the start of $\Phi$ and ends after $\varphi(\Gamma_i)$,

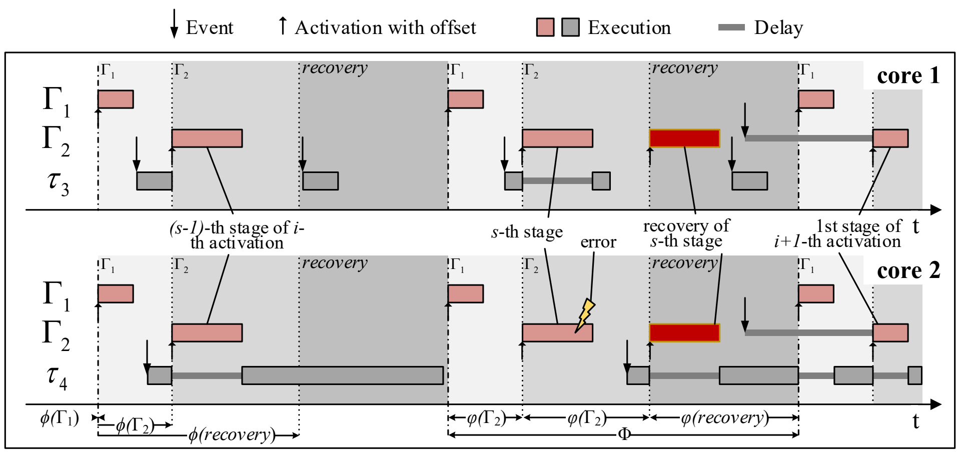


**Figure 2** Execution example with two fork-join and two independent tasks on two cores.

the slot length. The recovery slot is shared by all fork-join tasks on that core and is where error recovery may take place under a single error assumption (details in Sec. 3.3 and 4.3). The recovery slot has an offset $\phi(recovery)$ relative to $\Phi$ and length $\varphi(recovery)$. Lower priority independent tasks are allowed to execute whenever no higher priority workload is executing.

An example is shown in Fig. 2, where two fork-join tasks $\Gamma_1$ and $\Gamma_2$ and two independent tasks $\tau_3$ and $\tau_4$ are mapped to two cores. $\Gamma_1$ and $\Gamma_2$ execute in their respective slots simultaneously in both cores. When an error occurs, the recovery of $\Gamma_2$ is scheduled and the recovery of the error-affected stage occurs in the recovery slot. The use of offsets enables the schedulability of independent tasks with short periods and deadlines, such as $\tau_3$ and $\tau_4$. Note that, without the offsets, $\Gamma_1$ and $\Gamma_2$ would execute back-to-back leading to the unschedulability of $\tau_3$ and $\tau_4$.

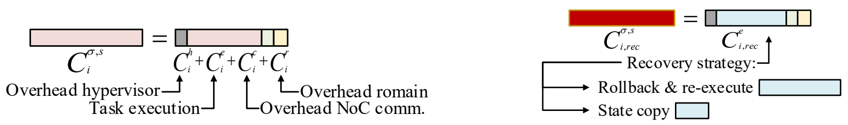
## 3.2 Task Model

An independent task $\tau_i$ is mapped to core $\sigma$ with a priority $p$. Once activated, it executes for at most $C_i$, its Worst-Case Execution Time (WCET). The activations of a task are modeled with arbitrary *event models*. Task activations in an event model are given by arrival curves $\eta^-(\Delta t)$ and $\eta^+(\Delta t)$, which return the minimum and maximum number of events arriving in any time interval $\Delta t$. Their pseudo-inverse counterparts $\delta^+(q)$ and $\delta^-(q)$ return the maximum and minimum time interval between the first and last events in any sequence of $q$ event arrivals. Conversion is provided in [27]. Periodic events with jitter, sporadic events and others can be modeled with the minimum distance function $\delta_i^-(q)$ as follows [27]:

$$\delta_i^-(q) = \max((q-1)\cdot d^{min}, (q-1)\cdot \mathcal{P} - \mathcal{J}) \tag{1}$$

where $\mathcal{P}$ is the period, $\mathcal{J}$ is the jitter, $d^{min}$ is the minimum distance between any two events, and the subscript $i$ indicates the association with a task $\tau_i$ or $\Gamma_i$.

Fork-join tasks are rigid parallel tasks, i.e. the number of processors required by a fork-join task is fixed and specified externally to the scheduler [12], and consist of multiple stages with data dependencies, as in [2, 1]. A fork-join task $\Gamma_i$ is a Directed Acyclic Graph (DAG) $G(V, E)$, where vertices in $V$ are subtasks and edges in $E$ are precedence dependencies [2]. In the graph, tasks are partitioned in *segments* and *stages*, as illustrated in Fig. 4a. A subtask $\tau_i^{\sigma,s}$ is the $s$-th stage of the $\sigma$-th segment and is annotated with its WCET $C_i^{\sigma,s}$. The WCET



**(a)** WCET of a fork-join subtask **(b)** WCET of recovery

**Figure 3** The composition of WCET of fork-join subtasks.

of a stage is equal across all segments, i.e. $\forall x, y$: $C_i^{x,s} = C_i^{y,s}$. Each segment $\sigma$ of $\Gamma_i$ is mapped to a distinct core. A fork-join task $\Gamma_i$ is annotated with the *static offset* $\phi(\Gamma_i)$, which marks the start of its execution slot in $\Phi$. The offset also admits a small positive jitter $j_\phi$, to account for a slight desynchronization between cores and context switch overhead.

The activations of a fork-join task are modeled with *event models*. Once $\Gamma_i$ is activated, its stages are successively activated by the completion of all segments of the previous stage, as in [2, 1]. Our approach differs from them in that it restricts the scheduling of at most one stage of $\Gamma_i$ in a cycle $\Phi$, and the stage receives service at the offset $\phi(\Gamma_i)$. Note that the event arrival at a fork-join task is not synchronized with its offset. The events at a fork-join task are queued at the first stage and only one event at a time is processed (FIFO) [2]. A queued event is admitted when the previous event leaves the last stage.

The interaction with Romain (the voter) is modeled in the analysis as part of the WCET $C_i^{\sigma,s}$, as depicted in Fig. 3a. The WCET includes the on-Chip communication latency and state comparison overheads, as the Romain instance may be mapped to an arbitrary core. Those can be obtained e.g. with [25] along with task mapping and scheduler properties to avoid over-conservative interference estimation and obtain tighter bounds.

## 3.3 Error Model

Our model assumes a single error scenario caused by SEEs (cf. Sec. 1). We assume that all errors affecting fork-join tasks can be detected and contained, ensuring integrity. The overhead of error detection mechanisms are modeled as part of the WCET (cf. Fig. 3a). Regarding independent tasks, we assume that an error immediately leads to a task failure and assume also that its failure will not violate the WCRT guarantees of the remaining tasks. Those assumptions are met e.g. by Romain[2]. Moreover, we assume the absence of failures in critical components [9, 24], such as the OS, the replica manager/voter Romain and the interconnect (e.g. the NoC), which can be protected as in [16, 26].

Our model provides recovery[2] for fork-join tasks, ensuring their availability. With a recovery slot in every cycle $\Phi$, our approach is able to handle up to one error per cycle $\Phi$. However, the analysis in Sec. 4.3 assumes at most one error per busy window for the sake of a simpler analysis (the concept will be introduced in Sec. 4). The assumption is reasonable since the probability of a multiple error scenario is very low and can be considered as an acceptable risk [17]. A multiple error scenario occurs only if an error affects more than one replica at a time or if more than one error occurs within the same busy window.

[2] Romain is able to detect and recover from all soft errors affecting user-level applications. For details on the different error impacts and detection strategies, the interested reader can refer to [3, 8].

## 3.4 Offsets

The execution of fork-join tasks in our approach is based on static offsets, which are assumed to be provided as input to the scheduler. The offsets form execution slots whose size do not vary during runtime, as seen in Fig. 2. Varying the slots sizes would substantially increase the timing analysis complexity without a justifiable performance gain. The offsets must satisfy two constraints:

▶ **Constraint 1.** *A slot for a fork-join task* $\Gamma_i$ *must be large enough to fit the largest stage of* $\Gamma_i$*. That is,* $\forall s, \sigma$: $\varphi(\Gamma_i) \geq C_i^{\sigma,s} + j_\phi$.

▶ **Constraint 2.** *The recovery slot must be large enough to fit the recovery of the largest stage of any fork-join task mapped to that core. That is,* $\forall i, s, \sigma$: $\varphi(recovery) \geq C_{i,rec}^{\sigma,s} + j_\phi$.

where a one error scenario per cycle is assumed and $C_{i,rec}^{\sigma,s}$ is the recovery WCET of subtask $\tau_i^{\sigma,s}$ (cf. Sec. 4.3).

We provide basic offsets that satisfy Constraints 1 and 2. The calculation must consider only overlapping fork-join tasks, i.e. fork-join tasks mapped to at least one core in common. Offsets for non-overlapping fork-join tasks are computed separately as they do not interfere directly with each other. The indirect interference, e.g. in the NoC, are accounted for in the WCETs. First we determine the smallest slots that satisfy Constraint 1:

$$\forall \Gamma_i : \varphi(\Gamma_i) = \max_{\forall \sigma,s} \{C_i^{\sigma,s}\} + j_\phi \tag{2}$$

and the smallest recovery slot that satisfies Constraint 2:

$$\varphi(recovery) = \max_{\forall \Gamma_i, \tau_j^{\sigma,s} \in \Gamma_i} \{C_{i,rec}^{\sigma,s}\} + j_\phi \,. \tag{3}$$

The cycle $\Phi$ is then the sum of all slots:

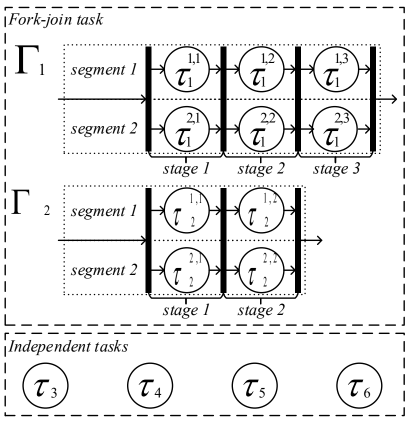
$$\Phi = \sum_{\forall \Gamma_i} \{\varphi(\Gamma_i)\} + \varphi(recovery) \,. \tag{4}$$

The offsets then depend on the order in which the slots are placed inside $\Phi$. Assuming that the slots $\phi(\Gamma_i)$ are sorted in ascending order on $i$ and that the recovery slot is the last one, the offsets are obtained by:

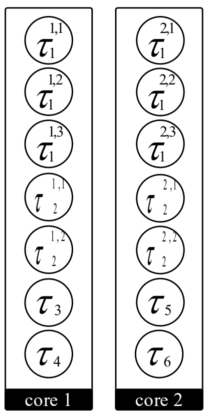
$$\phi(x) = \begin{cases} 0 & \text{if } x = \Gamma_1 \\ \phi(\Gamma_{i-1}) + \varphi(\Gamma_{i-1}) & \text{if } x = \Gamma_i \text{ and } i > 1 \\ \Phi - \varphi(recovery) & \text{if } x = recovery \end{cases} \tag{5}$$

## 4 Response-Time Analysis

The analysis is based on CPA and inspired by [2, 23]. In CPA, the WCRT is calculated with the busy window approach [29]. The response time of an event of a task $\tau_i$ (resp. $\Gamma_i$) is the time interval between the event arrival and the completion of its execution. In the busy window approach [29], the event with the WCRT can be found inside the busy window. The busy window $w_i$ of a task $\tau_i$ (resp. $\Gamma_i$) is the time interval where all response times of the task depend on the execution of at least one previous event in the same busy window, except for the task's first event. The busy window starts at a critical instant corresponding to the worst-case scheduling scenario. Since the worst-case scheduling scenario depends on the type of task, it will be derived individually in the sequel.


Fork-join task
$\Gamma_1$
segment 1
segment 2
$\tau_1^{1,1}$ $\tau_1^{1,2}$ $\tau_1^{1,3}$
$\tau_1^{2,1}$ $\tau_1^{2,2}$ $\tau_1^{2,3}$
stage 1
stage 2
stage 3
$\Gamma_2$
segment 1
segment 2
$\tau_2^{1,1}$ $\tau_2^{1,2}$
$\tau_2^{2,1}$ $\tau_2^{2,2}$
stage 1
stage 2
Independent tasks
$\tau_3$ $\tau_4$ $\tau_5$ $\tau_6$

**(a)** Taskset


$\tau_1^{1,1}$ $\tau_1^{1,2}$ $\tau_1^{1,3}$ $\tau_2^{1,1}$ $\tau_2^{1,2}$ $\tau_3$ $\tau_4$
core 1
$\tau_1^{2,1}$ $\tau_1^{2,2}$ $\tau_1^{2,3}$ $\tau_2^{2,1}$ $\tau_2^{2,2}$ $\tau_5$ $\tau_6$
core 2

**(b)** Mapping

**Figure 4** A taskset with 4 independent tasks and 2 fork-join tasks, and its mapping to 2 cores. Highest priority at the top, lowest at the bottom.

Before we derive the analysis for fork-join and for independent tasks, let us introduce the example in Fig. 4 used throughout the section. The taskset consists of 4 independent tasks and 2 fork-join tasks, mapped to two cores. The task priority on each core decreases from top to bottom (e.g. $\tau_1^{1,1}$ has the highest priority and $\tau_4$ the lowest).

## 4.1 Fork-Join Tasks

We now derive the WCRT for an arbitrary fork-join task $\Gamma_i$. Therefor, we need to identify the critical instant leading to the worst-case scheduling scenario. In case of SPP, the critical instant is when all tasks are activated at the same time and the tasks' subsequent events arrive as early as possible [29]. In our case, the critical instant must also account for the use of static offsets [23].

The worst-case scheduling scenario for $\Gamma_2$ on core 1 is illustrated in Fig. 5. $\Gamma_2$ is activated and executed at the same time on cores 1 and 2 (omitted). Note that, by design, fork-join tasks do not dynamically interfere with each other. The *critical instant* occurs when the first event of $\Gamma_2$ arrives just after missing $\Gamma_2$'s offset. The event has to wait until the next cycle to be served, which takes time $\Phi + j_\phi$ when the activation with offset is delayed by a jitter $j_\phi$. Notice that the WCETs of fork-join tasks already account for the inter-core communication and synchronization overhead (cf. Fig. 3a).

▶ **Lemma 1.** *The critical instant leading to the worst-case scheduling scenario of a fork-join task $\Gamma_i$ is when the first event of $\Gamma_i$ arrives just after missing $\Gamma_i$'s offset $\phi(\Gamma_i)$.*

**Proof.** A fork-join task $\Gamma_i$ does not suffer interference from independent tasks or other fork-join tasks. The former holds since independent tasks always have lower priority. The latter holds due to three reasons: an arbitrary fork-join task $\Gamma_j$ always receives service in its slot $\phi(\Gamma_j)$; the slot $\phi(\Gamma_j)$ is large enough to fit $\Gamma_j$'s largest subtask (Constraint 1); and the slots in a cycle $\Phi$ are disjoint. Thus, the critical instant can only be influenced by $\Gamma_i$ itself.

We prove by contradiction. Suppose that there is another scenario worse than Lemma 1. That means that the first event can arrive at a time that causes a delay to $\Gamma_i$ larger than $\Phi + j_\phi$. However, if the delay is larger than $\Phi + j_\phi$, then the event arrived before a previous slot $\phi(\Gamma_i)$ and $\Gamma_i$ did not receive service. Since that can only happen if there is a pending

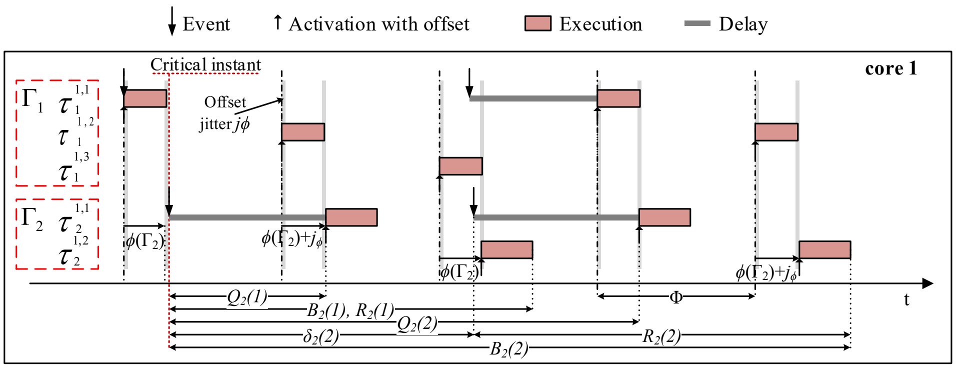


**Figure 5** Worst-case schedule for fork-join gang $\Gamma_2$ on core 1 (cf. Fig. 4).

activation of $\Gamma_i$ and thus violates the definition of a busy window, the hypothesis must be rejected. ◀

Let us now derive the Multiple-Event Queueing Delay $Q_i(q)$ and Multiple-Event Busy Time $B_i(q)$ on which the busy window relies. $Q_i(q)$ is the longest time interval between the arrival of $\Gamma_i$'s first activation and the first time its $q$-th activation receives service, considering that all events belong to the same busy window [2, 20]. For $\Gamma_i$, the $q$-th activation can receive service at the next cycle $\Phi$ after the execution of $q-1$ activations of $\Gamma_i$ lasting $s_i \cdot \Phi$ each, a delay $\Phi$ (cf. Lemma 1) and a jitter $j_\phi$. This is given by:

$$Q_i(q) = (q-1) \cdot s_i \cdot \Phi + \Phi + j_\phi \tag{6}$$

where $s_i$ is the number of stages of $\Gamma_i$ and $\Phi$ is the cycle.

▶ **Lemma 2.** *The Multiple-Event Queueing Delay $Q_i(q)$ given by Eq. 6 is an upper bound.*

**Proof.** The proof is by induction. When $q=1$, $\Gamma_i$ has to wait for service at most until the next cycle $\Phi$ plus an offset jitter $j_\phi$ to get service for its first stage, considering that the event arrives just after its offset (Lemma 1). In a subsequent $q+1$-th activation in the same busy window, Eq. 6 must also consider $q$ entire executions of $\Gamma_i$. Since $\Gamma_i$ has $s_i$ stages and only one stage can be activated and executed per cycle $\Phi$, it takes additional $s_i \cdot \Phi$ for each activation of $\Gamma_i$, resulting in Eq. 6. ◀

The Multiple-Event Busy Time $B_i(q)$ is the longest time interval between the arrival of $\Gamma_i$'s first activation and the completion of its $q$-th activation, considering that all events belong to the same busy window [2, 20]. The $q$-th activation of $\Gamma_i$ completes after a delay $\Phi$ (cf. Lemma 1), a jitter $j_\phi$ and the execution of $q$ activations of $\Gamma_i$. This is given by:

$$B_i(q) = q \cdot s_i \cdot \Phi + j_\phi + C_i^{\sigma,s} \tag{7}$$

where $C_i^{\sigma,s}$ is the WCET of $\Gamma_i$'s last stage.

▶ **Lemma 3.** *The Multiple-Event Busy Time $B_i(q)$ given by Eq. 7 is an upper bound.*

**Proof.** The proof is by induction. When $q=1$, $\Gamma_i$ has to wait for service at most until the next cycle $\Phi$ plus an offset jitter $j_\phi$ to get service for its first stage (Lemma 1), plus the completion of the last stage of the activation lasting $(s_i-1) \cdot \Phi + C_i^{\sigma,s}$. This is given by:

$$\begin{aligned} B_i(1) &= (s_i - 1) \cdot \Phi + \Phi + j_\phi + C_i^{\sigma,s} \\ &= s_i \cdot \Phi + j_\phi + C_i^{\sigma,s} \end{aligned} \tag{8}$$

In a subsequent $q+1$-th activation in the same busy window, Eq. 7 must consider $q$ additional executions of $\Gamma_i$. Since $\Gamma_i$ has $s_i$ stages and only one stage can be activated and executed per cycle $\Phi$, it takes additional $s_i \cdot \Phi$ for each activation of $\Gamma_i$. Thus, Eq. 7. ◀

Now we can calculate the busy window and WCRT of $\Gamma_i$. The busy window $w_i$ of a fork-join task $\Gamma_i$ is given by:

$$w_i = \max_{q \geq 1,\, q \in \mathbb{N}} \{B_i(q) \,|\, Q_i(q+1) \geq \delta_i^-(q+1)\}\,. \tag{9}$$

▶ **Lemma 4.** *The busy window is upper bounded by Eq. 9.*

**Proof.** The proof is by contradiction. Suppose there is a busy window $\breve{w}_i$ longer than $w_i$. In that case, $\breve{w}_i$ must contain at least one activation more than $w_i$, i.e. $\breve{q} \geq q+1$. From Eq. 9, we have that $Q_i(\breve{q}) < \delta_i^-(\breve{q})$, i.e. $\breve{q}$ is not delayed by the previous activation. Since that violates the definition of a busy window, the hypothesis must be rejected. ◀

The response time $R_i(q)$ of the $q$-th activation of $\Gamma_i$ in the busy window is given by:

$$R_i(q) = B_i(q) - \delta_i^-(q)\,. \tag{10}$$

The worst-case response time $R_i^+$ is the longest response time of any activation of $\Gamma_i$ observed in the busy window.

$$R_i^+ = \max_{1 \leq q \leq \eta_i^+(w_i)} R_i(q)\,. \tag{11}$$

▶ **Theorem 5.** *$R_i^+$ (Eq. 11) provides an upper bound on the worst-case response time of an arbitrary fork-join task $\Gamma_i$.*

**Proof.** The WCRT of a fork-join task $\Gamma_i$ is obtained with the busy window approach [29]. It remains to prove that the critical instant leads to the worst-case scheduling scenario, that the interference captured in Eqs. 6 and 7 are upper bounds, and that the busy window is correctly captured by Eq. 9. These are proved in Lemmas 1, 2, 3, and 4, respectively. ◀

## 4.2 Independent Tasks

We now derive the WCRT analysis of an arbitrary independent task $\tau_i$. Two types of interference affect independent tasks: interference caused by higher priority independent tasks and by fork-join tasks. Let us first identify the critical instant leading to the worst-case scheduling scenario where $\tau_i$ suffers the most interference.

▶ **Lemma 6.** *The critical instant of $\tau_i$ is when the first event of higher priority independent tasks arrive simultaneously with $\tau_i$'s event at the offset of a fork-join task.*

**Proof.** The worst-case interference caused by a higher priority (independent) task $\tau_j$ under SPP is when its first event arrives simultaneously with $\tau_i$'s and continue arriving as early as possible [29].

The interference caused by a fork-join task $\Gamma_j$ on $\tau_i$ depends on $\Gamma_j$'s offset $\phi(\Gamma_j)$ and subtasks $\tau_j^{\sigma,s}$, whose execution times vary for different stages $s$. Assume a critical instant that occurs at a time other than at the offset $\phi(\Gamma_j)$. Since a task $\Gamma_j$ starts receiving service at its offset, an event of $\tau_i$ arriving at time $t > \phi(\Gamma_j)$ can only suffer less interference from $\Gamma_j$'s subtask than when arriving at $t = 0$. ◀

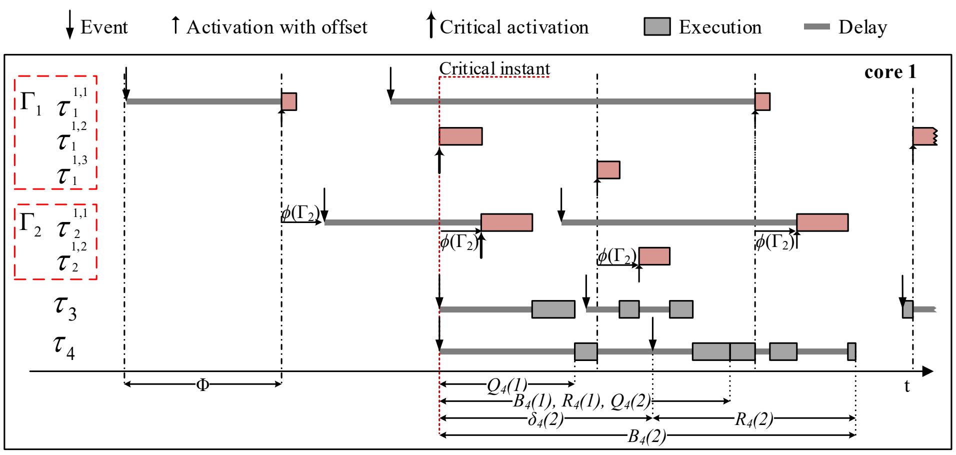


**Figure 6** The worst-case schedule for independent task $\tau_4$ on core 1 (cf. Fig. 4).

Fork-join subtasks have different execution times for different stages, which leads to a number of scheduling scenarios that must be evaluated [23]. Each scenario is defined by the fork-join subtasks that will receive service in the cycle $\Phi$ and the offset at which the critical instant supposedly occurs. The scenario is called a critical instant candidate $S$. Since independent tasks participate in all critical instant candidates, they are omitted in $S$ for the sake of simplicity.

▶ **Definition 7.** Critical Instant Candidate $S$: the critical instant candidate $S$ is an ordered pair $(a, b)$ where $a$ is a critical offset and $b$ is a tuple containing one subtask $\tau_j^{\sigma,s}$ of every interfering fork-join task $\Gamma_j$.

Let us also define the set of candidates that must be evaluated.

▶ **Definition 8.** Critical Instant Candidate Set $\mathcal{S}$: the set containing all possible different critical instant candidates $S$.

The worst-case schedule of the independent task $\tau_4$ from the example in Fig. 4 is illustrated in Fig. 6. In fact, the critical instant leading to $\tau_4$'s WCRT is at $\phi(\Gamma_1)$ when $\tau_1^{1,2}$ and $\tau_2^{1,1}$ receive service at the same cycle $\Phi$, i.e. $S = (\phi(\Gamma_1), (\tau_1^{1,2}, \tau_2^{1,1}))$. Events of the independent task $\tau_3$ start arriving at the critical instant and continue arriving as early as possible.

Let us now bound the interference $I_i^I(\Delta t)$ caused by equal or higher priority independent tasks in any time interval $\Delta t$. The interference $I_i^I(\Delta t)$ can be upper bounded as follows [20]:

$$I_i^I(\Delta t) = \sum_{\forall \tau_j \in hp_I(i)} \eta_j^+(\Delta t) \cdot C_j \tag{12}$$

where $hp_I(i)$ is the set of equal or higher priority independent tasks mapped to the same core as $\tau_i$.

To derive the interference caused by fork-join tasks we need to define the Critical Instant Event Model. The critical instant event model $\check{\eta}_i^{\sigma,s}(\Delta t, S)$ of a subtask $\tau_i^{\sigma,s} \in \Gamma_i$ returns the maximum number of activations observable in any time interval $\Delta t$, assuming the critical instant $S$. It can be derived from $\Gamma_i$'s input event model $\eta_i^+(\Delta t)$ as follows:

$$\check{\eta}_i^{\sigma,s}(\Delta t, S) = \min\left\{\eta_i^+\big(\Delta t_S + \Phi - \phi(\Gamma_i)\big), \psi\right\} - gt\big(s^S, s, \phi^S, \phi(\Gamma_i)\big) \tag{13}$$

$$\psi = \left\lfloor \frac{\Delta t_S}{\Phi \cdot s_i} \right\rfloor + ge\big(\Delta t_S \bmod (\Phi \cdot s_i)\,,\, \Phi \cdot (s-1)\big) \tag{14}$$

$$\Delta t_S = \Delta t + \underbrace{\Phi \cdot (s^S - 1)}_{\substack{\text{critical instant}\\ \text{stage}}} + \underbrace{\phi^S}_{\substack{\text{critical instant}\\ \text{offset}}} \tag{15}$$

where $s$ is the stage of subtask $\tau_i^{\sigma,s}$; $s_i$ is the number of stages in $\Gamma_i$; $\phi^S$ is the offset in $S$; $s^S$ is the stage of $\Gamma_i$ in $S$; $gt(a,b,c,d)$ is a function that returns 1 when $(a > b) \vee (a = b \wedge c > d)$, 0 otherwise; and $ge(a,b)$ is a function that returns 1 when $a \geq b$, 0 otherwise.

▶ **Lemma 9.** *$\check{\eta}_i^{\sigma,s}(\Delta t, S)$ (Eq. 13) provides a valid upper bound on the number of activations of $\tau_i^{\sigma,s}$ observable in any time interval $\Delta t$, assuming the critical instant $S$.*

For the sake of readability, the proof is presented in Appendix A.

The interference $I_i^{FJ}(\Delta t, S)$ caused by fork-join tasks on the same core in any time interval $\Delta t$, assuming a critical instant candidate $S$, can then be upper bounded as follows:

$$I_i^{FJ}(\Delta t, S) = \sum_{\forall \tau_j^{\sigma,s} \in hp_{FJ}(i)} \check{\eta}_j^{\sigma,s}(\Delta t, S) \cdot C_j^{\sigma,s} \tag{16}$$

where $hp_{FJ}(i)$ is the set of fork-join subtasks mapped to the same core as $\tau_i$.

The Multiple-Event Queueing Delay $Q_i(q,S)$ and Multiple-Event Busy Time $B_i(q,S)$ for an independent task $\tau_i$, assuming a critical instant candidate $S$, can be derived as follows.

$$Q_i(q,S) = (q-1) \cdot C_i + I_i^I(Q_i(q,S)) + I_i^{FJ}(Q_i(q,S), S) \tag{17}$$

$$B_i(q,S) = q \cdot C_i + I_i^I(B_i(q,S)) + I_i^{FJ}(B_i(q,S), S) \tag{18}$$

where $q \cdot C_i$ is the time required to execute $q$ activations of task $\tau_i$.

Eqs. 17 and 18 result in fixed-point problems, similar to the well known busy window equation (Eq. 9). They can be solved iteratively, starting with a very small, positive $\epsilon$.

▶ **Lemma 10.** *The Multiple-Event Queueing Delay $Q_i(q,S)$ given by Eq. 17 is an upper bound, assuming the critical instant $S$.*

**Proof.** The proof is by induction. When $q=1$, $\tau_i$ has to wait for service until the interfering workload is served. The interfering workload is given by Eqs. 12 and 16. Since $\eta_j^+(\Delta t)$ and $C_j$ are upper bounds by definition, Eq. 12 is also an upper bound. Similarly, since $\check{\eta}_j^{\sigma,s}(\Delta t, S)$ is an upper bound (cf. Lemma 9) and $C_j^{\sigma,s}$ is an upper bound by definition, 16 is an upper bound for a given $S$. Therefore, $Q_i(1,S)$ is also an upper bound, for a given $S$.

In a subsequent $q+1$-th activation in the same busy window, $Q_i(q,S)$ also must consider $q$ executions of $\tau_i$. This is captured in Eq. 17 by the first term, which is, by definition, an upper bound on the execution time. From that, Lemma 10 follows. ◀

▶ **Lemma 11.** *The Multiple-Event Busy Time $B_i(q,S)$ given by Eq. 18 is an upper bound, assuming the critical instant $S$.*

**Proof.** The proof is similar to Lemma 10, except that $B_i(q, S)$ in Eq. 18 also captures the completion of the $q$-th activation. It takes additional $C_i$, which is an upper bound by definition. Thus Eq. 18 is an upper bound, for a given $S$. ◀

The busy window $w_i(q, S)$ of an independent task $\tau_i$ is given by:

$$w_i(S) = \max_{q \geq 1,\, q \in \mathbb{N}} \{B_i(q, S) \mid Q_i(q+1, S) \geq \delta_i^-(q+1)\} \tag{19}$$

▶ **Lemma 12.** *The busy window is upper bounded by Eq. 19.*

**Proof.** The proof is by contradiction. Suppose there is a busy window $\breve{w}_i(S)$ longer than $w_i(S)$. In that case, $\breve{w}_i(S)$ must contain at least one activation more than $w_i(S)$, i.e. $\breve{q} \geq q+1$. From Eq. 19, we have that $Q_i(\breve{q}, S) < \delta_i^-(\breve{q})$, i.e. $\breve{q}$ is not delayed by the previous activation. Since that violates the definition of a busy window, the hypothesis must be rejected. ◀

The response time $R_i$ of the $q$-th activation of a task in a busy window is given by:

$$R_i(q, S) = B_i(q, S) - \delta_i^-(q) \tag{20}$$

Finally, the worst-case response time $R_i^+$ is found inside the busy window and must be evaluated for all possible critical instant candidates $S \in \mathcal{S}$. The worst-case response time $R_i^+$ is given by:

$$R_i^+ = \max_{S \in \mathcal{S}} \left\{ \max_{1 \leq q \leq \eta_i^+(w_i(S))} \{R_i(q, S)\} \right\} \tag{21}$$

where the set $\mathcal{S}$ is given by the following Cartesian products:

$$\mathcal{S} = \{\phi(\Gamma_j), \phi(\Gamma_k), \ldots\} \times \{\sigma_i(\Gamma_j) \times \sigma_i(\Gamma_k) \times \ldots\} \tag{22}$$

where $\Gamma_j, \Gamma_k, \ldots$ are all fork-join tasks mapped to the same core as $\tau_i$ and $\sigma_i(\Gamma_j)$ is the set of subtasks of $\Gamma_j$ that are mapped to that core. When no fork-join tasks interfere with $\tau_i$, $\mathcal{S} = \{(0, ())\}$.

▶ **Theorem 13.** *$R_i^+$ (Eq. 21) returns an upper bound on the worst-case response time of an independent task $\tau_i$.*

**Proof.** We must first prove that, for a given $S$, $R_i^+$ is an upper bound. $R_i^+$ is obtained with the busy window approach [29]. It returns the maximum response time $R_i(q, S)$ among all activations inside the busy window. From Lemmas 10 and 11 we have that Eqs. 17 and 18 are upper bounds for a given $S$. From Lemma 12 we have that the busy window is captured by Eq. 19. Since the first term of Eq. 20 is an upper bound and the second term is a lower bound by definition, $R_i(q, S)$ is an upper bound. Thus $R_i^+$ is an upper bound for a given $S$. Since Eq. 21 evaluates the maximum response time over all $S \in \mathcal{S}$, $R_i^+$ is an upper bound on the response time of $\tau_i$. ◀

## 4.3 Error Recovery

Designed for mixed-criticality, our approach supports different recovery strategies for different fork-join tasks (cf. Sec. 2). For instance, in DMR augmented with checkpointing and rollback, recovery consists in reverting the state and re-executing the error-affected stage in both replicas. In TMR, recovery consists in copying and replacing the state of the faulty replica with the state of a healthy one. The different strategies are captured in the analysis by the recovery execution time, which depends on the strategy and the stage to be recovered. The recovery WCET $C_{i,rec}^{\sigma,s}$ of a fork-join subtask $\tau_i^{\sigma,s}$ accounts for the adopted recovery strategy as illustrated in Fig. 3b. Once an error is detected, error recovery is triggered and executed in the recovery slot of the same cycle $\Phi$. Fig. 2 illustrates the recovery of the $s$-th stage of $\Gamma_2$'s $i$-th activation.

Let us incorporate the error recovery into the analysis. For a fork-join task $\Gamma_i$, we must only adapt the Multiple-Event Busy Time $B_i(q)$ (Eq. 7) to account for the execution of the recovery:

$$B_i^{rec}(q) = q \cdot s_i \cdot \Phi + j_\phi + \phi(recovery) - \phi(\Gamma_i) + C_{i,rec}^{\sigma,s} \tag{23}$$

where $C_{i,rec}^{\sigma,s}$ is the WCET of the recovery of last subtask of $\Gamma_i$. The recovery of another task $\Gamma_j$ does not interfere with $\Gamma_i$'s WCRT. Only the recovery of one of $\Gamma_i$'s subtasks can interfere with $\Gamma_i$'s WCRT. Moreover, since the recovery of a subtask occurs in the recovery slot of the same cycle $\Phi$ and does not interfere with the next subtask, only the recovery of the last stage of $\Gamma_i$ actually has an impact on its response time. This is captured by the three last terms of Eq. 23.

For an independent task $\tau_i$, the worst-case impact of recovery of a fork-join task $\Gamma_j$ is modelled as an additional fork-join task $\Gamma_{rec}$ with one subtask $\tau_{rec}^{\sigma,1}$ mapped to the same core as $\tau_i$ and that executes in the *recovery* slot. The WCET $C_{rec}^{\sigma,1}$ of $\tau_{rec}^{\sigma,1}$ is chosen as the maximum recovery time among the subtasks of all fork-join tasks mapped to that core:

$$C_{rec}^{\sigma,1} = \max_{\forall \tau_j^{\sigma,s} \in hp_{FJ}(i)} \{C_{i,rec}^{\sigma,s}\} \tag{24}$$

With $\Gamma_{rec}$ mapped, Eq. 21 finds the critical instant where the recovery $C_{rec}^{\sigma,1}$ has the worst impact on the response time of $\tau_i$.

## 5 Experimental Evaluation

In our experiments we evaluate our approach with real as well as synthetic workload, focusing on the performance of the scheduler. First we characterize MiBench applications [13] and evaluate them as fork-join (replicated) tasks in the system. Then we evaluate the performance of independent (regular) tasks. Finally we evaluate the approach with synthetic workload when varying parameters of fork-join tasks.

### 5.1 Evaluation with benchmark applications

#### 5.1.1 Characterization

First we extract execution times and number of stages from MiBench automotive and security applications [13]. They were executed with small input on an ARMv7@1GHz and a DDR3-1600 [7]. Table 1 summarizes the total WCET, *observed* number of stages and WCET of the longest stage (max). A stage is delimited by *syscalls* (cf. Sec. 2). We report the observed execution times as WCETs. As pointed out in [2], stages vary on number and execution time depending on the application and on the current activity in that stage (computation/IO). This is seen e.g. in *susan*, where 99% of the WCET is concentrated in one stage (computation) while the other stages perform mostly IO and are on average 3.34us long.

In our approach, the optimal is when all stages of a fork-join task have the same WCET. There are two possibilities to achieve that: to *split* very long stages in smaller ones or to *group* small subsequent stages together. We exploit the latter as it does not require changes

**Table 1** MiBench applications' profile.

| | **WCET** | **Observed stages** | | **Grouped stages** | |
|---|---|---|---|---|---|
| | [ms] | #stages | max WCET [ms] | #stages | max WCET [ms] |
| basicmath | 32.48 | 19738 | 0.02 | 5 | 6.50 |
| bitcount | 24.42 | 30 | 15.16 | 3 | 15.16 |
| susan | 9.63 | 12 | 9.59 | 1 | 9.63 |
| blowfish | 0.11 | 7 | 0.09 | 1 | 0.11 |
| rijndael | 13.17 | 93 | 0.37 | 3 | 5.91 |
| sha | 3.49 | 51 | 0.11 | 2 | 1.90 |

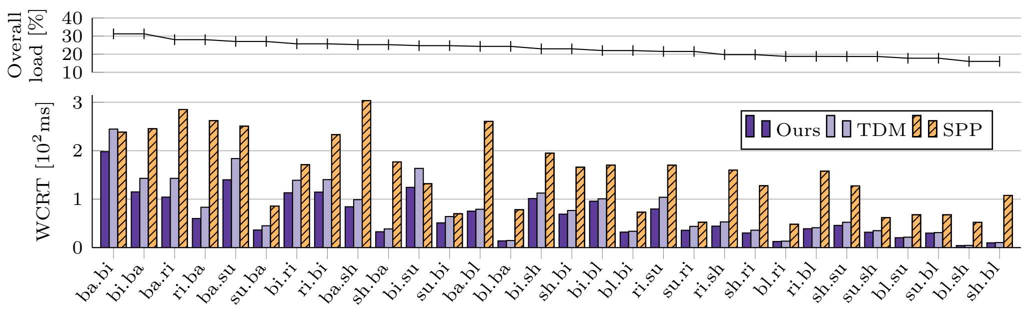


**Figure 7** WCRT of fork-join tasks with two segments derived from MiBench.

to the error detection mechanism or to our model. The results with *grouped* stages are shown on the right-hand side of Table 1. We have first grouped stages without increasing the maximum stage length. The largest improvement is seen in *bitcount*, where the number of stages reduces in one order of magnitude. In cases where all stages are very short, we increase the maximum stage length. When increasing the maximum stage length in two orders of magnitude, the number of stages of *basicmath* reduces in four orders of magnitude. We have manually chosen the maximum stage length. Alternatively the problem of finding the maximum stage length can be formulated as an optimization problem that e.g. minimizes the overall WCRT or maximizes the slack. Next, we map the applications as fork-join tasks and evaluate their WCRTs.

### 5.1.2 Evaluation of fork-join tasks

Two applications at a time are mapped as fork-join tasks with two segments (i.e. replicas in DMR) to two cores (cf. Fig. 4). On each core, 15% load is introduced by ten independent tasks generated with UUniFast [6]. We compare our approach with a TDM-based scheduler and Axer's Partitioned SPP [2]. In TDM, each fork-join task executes (and recovers) in its own slot. Independent tasks execute in a third slot, which replaces the recovery slot of our approach. The size of the slots are derived from our offsets. For all approaches, the priority assignment for independent tasks is deadline monotonic and considers that deadline equals period. In SPP, the deadline monotonic priority assignment also includes fork-join tasks.

The results are plotted in Fig. 7, where *ba.bi* gives the WCRT of *basicmath* when mapped together with *bitcount*. Despite the low system load, our approach also outperforms SPP in all cases, with bounds 58.2% lower, on average. Better results with SPP cannot be obtained unless the interfering workload is removed or highest priority is given to the fork-join tasks [2], which violates DM. Despite the similarity of how our approach handles fork-join tasks with TDM, the proposed approach outperforms TDM in all cases, achieving,

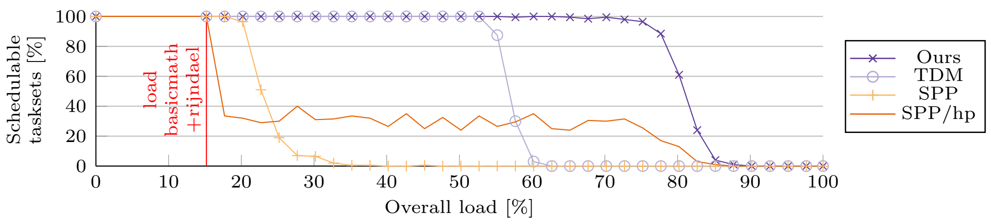
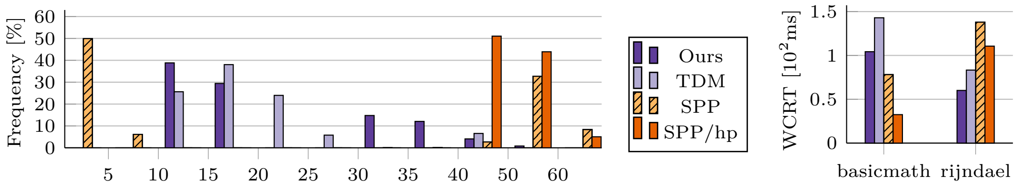


**Figure 8** Schedulability as a function of the load of the system. *Basicmath* and *rijndael* as fork-join tasks with two segments.



**(a)** WCRT of independent tasks [ms]

**(b)** WCRT of FJ tasks

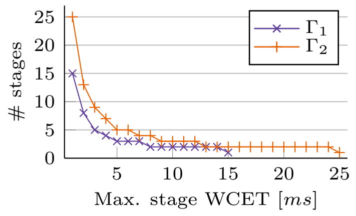
**Figure 9** *Basicmath* and *rijndael* as replicated tasks in DMR running on a dual-core configuration with 20.2% load (5% load from independent tasks).

on average, bounds 13.9% lower. This minor difference is because TDM slots must be slightly longer than our offsets to fit an eventual recovery. Nonetheless, not only our approach can guarantee small WCRT for replicated tasks but also provides for the worst-case performance of independent tasks.

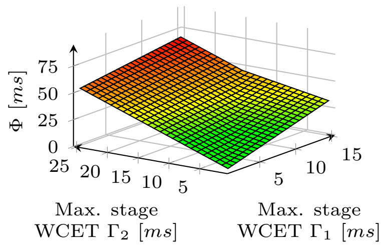
## 5.1.3 Evaluation of independent tasks

In a second experiment we fix *bitcount* and *rijndael* as fork-join tasks and vary the load on both cores. The generated task periods are in the range [20,500] ms, larger than the longest stage of the fork-join tasks. The schedulability of the system as the load increases is shown in Fig. 8. Our approach outperforms TDM and SPP in all cases, scheduling 1.55x and 6.96x more tasksets, respectively. Due to its non-work conserving characteristic, TDM's schedulability is limited to medium loads. SPP provides very small response times with lower loads but, as the load increases, the schedulability drops fast due to high interference (and thus high WCRT) suffered by fork-join tasks. For reference purposes, we also plot the schedulability of SPP when assigning the highest priorities to the fork-join tasks (SPP/hp). The schedulability in higher loads improves but losing deadline monotonicity guarantees renders the systems unusable in practice. Moreover, when increasing the jitter to 20% (relative to period), schedulability decreases 14.2% but shows the same trends for all schedulers.

Fig. 9 details the tasks' WCRTs when the system load is 20.2%. Indeed, when schedulable, SPP provides some of the smallest WCRTs for independent tasks, and SPP/hp improves the response times of fork-join tasks at the expense of the independent tasks'. Our approach provides a balanced trade-off between the performance of independent tasks and of fork-join tasks, and achieves high schedulability even in higher loads.

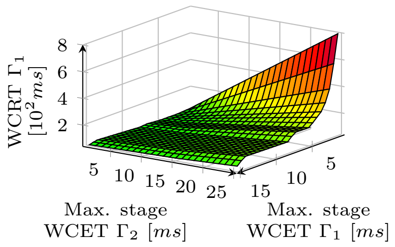


**(a)** Stages of $\Gamma_1$ and $\Gamma_2$

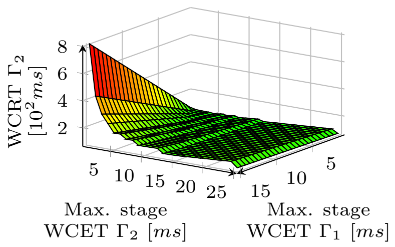


**(b)** Cycle $\Phi$

**Figure 10** Parameters of two fork-join tasks $\Gamma_1$ and $\Gamma_2$ with two segments running on a dual-core configuration.



**(a)** WCRT of $\Gamma_1$



**(b)** WCRT of $\Gamma_2$

**Figure 11** Performance of fork-join tasks $\Gamma_1$ and $\Gamma_2$ as a function of the maximum stage WCET.

## 5.2 Evaluation with synthetic workload

We now evaluate the performance of our approach when varying parameters such as stage length and cycle $\Phi$.

### 5.2.1 Evaluation of fork-join tasks

Two fork-join tasks $\Gamma_1$ and $\Gamma_2$ with two segments each (i.e. replicas in DMR) are mapped to two cores. The total WCETs[3] of $\Gamma_1$ and $\Gamma_2$ are 15 and 25ms, respectively. Both tasks are sporadic, with a minimum distance of 1s between activations. The number of stages of $\Gamma_1$ and $\Gamma_2$ is varied as a function of the maximum stage WCET, as depicted in Fig. 10a. The length of the cycle $\Phi$, depicted in Fig. 10b, varies with the maximum stage WCET since it is derived from them (cf. Sec. 3.4).

The system performance as the maximum stage lengths of $\Gamma_1$ and $\Gamma_2$ increase is reported in Fig. 11. The WCRT of $\Gamma_1$ increases with the stage length (Fig. 11a) as it depends on the number of stages and $\Phi$'s length. In fact, the WCRT of $\Gamma_1$ is longest when the stages of $\Gamma_1$ are the shortest and the stages of the interfering fork-join task ($\Gamma_2$) are the longest. Conversely, WCRT of $\Gamma_1$ is shortest when its stages are the longest and the stages of the interfering fork-join task are the shortest. The same occurs to $\Gamma_2$ in Fig. 11b. Thus, there is a trade-off between the response times of interfering fork-join tasks. This is plotted in Fig. 12 as the sum of the WCRTs of $\Gamma_1$ and $\Gamma_2$. As can be seen in Fig. 12, low response times can be obtained next and above to the line segment between the origin $(0, 0, 0)$ and the point $(15, 25, 0)$, the total WCETs[1] of $\Gamma_1$ and $\Gamma_2$ respectively.

[3] The sum of the WCET of all stages of a fork-join task.

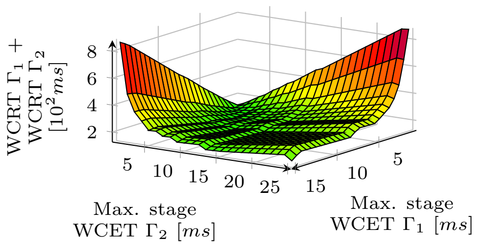
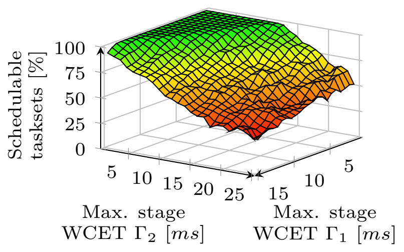
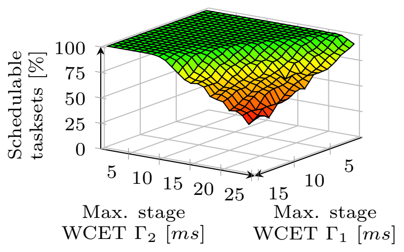


**Figure 12** WCRT trade-off between interfering fork-join tasks.



**(a)** Task period interval [15-500] ms



**(b)** Task period interval [25-500] ms

**Figure 13** Schedulable tasksets as a function of the maximum stage WCET of fork-join tasks $\Gamma_1$ and $\Gamma_2$ with 25% load from independent tasks.

### 5.2.2 Evaluation of independent tasks

To evaluate the impact of the parameters on independent tasks, we extend the previous scenario introducing 25% load on each core with ten independent tasks generated with UUniFast [6]. The task periods are within the interval [15,500] ms for the first experiment, and the interval [25,500] ms for the second. The priority assignment is deadline monotonic and considers that the deadline is equal to the period.

The schedulability as a function of the stage lengths is shown in Fig. 13. Sufficiently long stages cause the schedulability to decrease as independent tasks with short periods start missing their deadlines. This is seen in Fig. 13a when the stage length of either fork-join task reaches 15ms, the minimum period for the generated tasksets. Thus, when increasing the minimum period of generated tasks to 25ms, the number of schedulable tasksets also increases (Fig. 13b).

The maximum stage length of a fork-join task has direct impact on the response times and schedulability of the system. For the sake of performance, shorter stage lengths are preferred. However, that is not always possible because it would result in a large number of stages or because of the application, which restricts the minimum stage length (cf. Sec. 5.1.1). Nonetheless, fork-join tasks still are able to perform well with appropriate parameter choices. Additionally, one can formulate the problem of finding the stage lengths according to an objective function, such as minimize the overall response time or maximize the slack. The offsets can also be included in the formulation, as long as Constraints 1 and 2 are met.

## 6 Conclusion

In this paper, we presented the replica-aware co-scheduling for mixed-critical systems, where applications with different requirements and criticalities co-exist. The work includes a formal WCRT analysis supporting different recovery strategies and accounting for the NoC communication delay and overheads due to replica management and state comparison. Our approach provides for high worst-case performance of replicated software execution on many-core architectures without impairing the remaining tasks in the system. Experimental results with benchmark applications showed an improvement on taskset schedulability of up to 6.9x when compared to Partitioned SPP and 1.5x when compared to a TDM-based scheduler.

Naturally, there is always the possibility of critical components failing due to errors (replica manager or voter and the OS). In that case, either enough time for a reboot must be allocated or the critical components must be hardened.

**Acknowledgement.** We would like to thank Adam Lackorzynski and Tobias Stumpf for the helpful discussions.

## References

1 Björn Andersson and Dionisio de Niz. Analyzing Global-EDF for multiprocessor scheduling of parallel tasks. In *International Conference On Principles Of Distributed Systems*, pages 16–30. Springer, 2012.

2 Philip Axer. *Performance of Time-Critical Embedded Systems under the Influence of Errors and Error Handling Protocols*. PhD thesis, TU Braunschweig, 2015.

3 Philip Axer, Rolf Ernst, Björn Döbel, and Hermann Härtig. Designing an analyzable and resilient embedded operating system. In *Proc. on Software-Based Methods for Robust Embedded Systems*, Braunschweig, Germany, 2012.

4 Philip Axer, Sophie Quinton, Moritz Neukirchner, Rolf Ernst, Bjorn Dobel, and Hermann Hartig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proc. of ECRTS'13*, 2013.

5 Philip Axer, Maurice Sebastian, and Rolf Ernst. Reliability analysis for mpsocs with mixed-critical, hard real-time constraints. In *Proc. Intl. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011.

6 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

7 Nathan Binkert, Bradford Beckmann, Gabriel Black, et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2), August 2011.

8 Björn Döbel, Hermann Härtig, and Michael Engel. Operating system support for redundant multithreading. In *Proc. of EMSOFT'12*, 2012.

9 Michael Engel and Björn Döbel. The reliable computing base-a paradigm for software-based reliability. In *GI-Jahrestagung*, pages 480–493, 2012.

10 Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306 – 318, 1992. doi:10.1016/0743-7315(92)90014-E.

11 Rémi Gaillard. Single event effects: Mechanisms and classification. In Michael Nicolaidis, editor, *Soft Errors in Modern Electronic Systems*. Springer US, 2011.

12 Joël Goossens and Vandy Berten. Gang ftp scheduling of periodic and parallel rigid real-time tasks. *arXiv preprint arXiv:1006.2617*, 2010.

13 M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4. 2001*, Dec 2001.

14 R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis–the SymTA/S Approach. *IEE Proceedings-Computers and Digital Techniques*, 152, 2005.

15 Andreas Herkersdorf et al. Resilience Articulation Point (RAP): Cross-layer dependability modeling for nanometer system-on-chip resilience. *Microelectronics Reliability*, 54(6–7):1066–1074, 2014. doi:10.1016/j.microrel.2013.12.012.

16 M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann. dOSEK: the design and implementation of a dependability-oriented static embedded kernel. In *Proc. of RTAS'15*, pages 259–270, 2015. doi:10.1109/RTAS.2015.7108449.

17 International Standards Organization. *ISO 26262: Road Vehicles – Functional Safety*, 2011.

18 Robert Kaiser and Stephan Wagner. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, 2007.

19 Shinpei Kato and Yutaka Ishikawa. Gang EDF scheduling of parallel task systems. In *Proc. of RTSS'09*, 2009.

20 J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. of RTSS'90*, 1990.

21 NXP MPC577xK Ultra-Reliable MCU Family. [online]. Available: `http://www.nxp.com/assets/documents/data/en/fact-sheets/MPC577xKFS.pdf`, 2017.

22 John K. Ousterhout. Scheduling techniques for concurrent systems. In *ICDCS*, volume 82, pages 22–30, 1982.

23 J. C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of RTSS'98*, 1998. doi:10.1109/REAL.1998.739728.

24 Eberle A. Rambo and Rolf Ernst. Providing flexible and reliable on-chip network communication with real-time constraints. In *1st International Workshop on Resiliency in Embedded Electronic Systems (REES)*, 2015.

25 Eberle A. Rambo, Selma Saidi, and Rolf Ernst. Providing formal latency guarantees for ARQ-based protocols in networks-on-chip. In *Proc. of DATE'16*, 2016.

26 Eberle A. Rambo, Christoph Seitz, Selma Saidi, and Rolf Ernst. Designing networks-on-chip for high assurance real-time systems. In *Proc. of PRDC'17*, 2017.

27 K. Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, TU Braunschweig, 2005.

28 RTCA Incorporated. *DO-254: Design Assurance Guidance For Airborne Electronic Hardware*, 2000.

29 K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2), 1994.

## A Additional Proofs

For the sake of readability, the proof of Lemma 9 (Sec. 4.2) is presented here.

**Proof.** The proof is by induction, in two parts. First let us assume $s^S = 1$ and $\phi^S = 0$, neutral values resulting in $\Delta t_S = \Delta t$ and $gt(s^S, s, \phi^S, \phi(\Gamma_i)) = 0$. The maximum number of activations of $\tau_i^{\sigma,s}$ seen in the interval $\Delta t$ is limited by the maximum number of activations of the fork-join task $\Gamma_i$ because a subtask $\tau_i^{\sigma,s}$ is activated once per $\Gamma_i$'s activation, and limited by the maximum number of times that $\tau_i^{\sigma,s}$ can actually be scheduled and served in $\Delta t$. This is ensured in Eq. 13 by the minimum function and its first and second terms, respectively.

When $s^S > 1$ and/or $\phi^S > 0$, the time interval $[0, \Delta t)$ must be moved forward so that it starts at stage $s^S$ and offset $\phi^S$. This is captured by $\Delta t_S$ in Eq. 15 and by the last term of Eq. 13. The former extends the end of the time interval by the time it takes to reach the stage $s^S$ and the offset $\phi^S$, i.e. $[0, \Delta t_S)$. The latter pushes the start of the interval forward by subtracting an activation of $\tau_i^{\sigma,s}$ if it occurs before the stage $s^S$ and the offset $\phi^S$, resulting in the interval $[\Delta t_S - \Delta t, \Delta t_S)$. Thus Eq. 13. ◀