UNIVERSITY _of York_

This is a repository copy of *LEFT-RS: A Lock-Free Fault-Tolerant Resource Sharing Protocol for Multicore Real-Time Systems*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/232452/

Version: Accepted Version

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

LEFT-RS: A Lock-Free Fault-Tolerant Resource Sharing Protocol for Multicore Real-Time Systems

Nan Chen*, Xiaotian Dai*, Tong Cheng†, Alan Burns*, Iain Bate*, Shuai Zhao†

*University of York, UK †Sun Yat-sen University, China

Abstract—**Emerging real-time applications have driven the transition to multicore embedded systems, where tasks must share resources due to functional demands and limited availability. These resources, whether local or global, are protected within critical sections to prevent race conditions, with locking protocols ensuring both exclusive access and timing requirements. However, transient faults occurring within critical sections can disrupt execution and propagate errors across multiple tasks. Conventional locking protocols fail to address such faults, and integrating traditional fault tolerance techniques often increases blocking. Recent approaches improve fault recovery through parallel replica execution; however, challenges remain due to sequential accessing, coordination overhead, and susceptibility to common-mode faults. In this paper, we propose a Lock-frEe Fault-Tolerant Resource Sharing (LEFT-RS) protocol for multicore real-time systems. LEFT-RS allows tasks to concurrently access and read global resources while entering their critical sections in parallel. Each task can complete its access earlier upon successful execution if other tasks experience faults, thereby improving the efficiency of resource usage. Our design also limits the overhead and enhances fault resilience. We present a comprehensive worst-case response time analysis to ensure timing guarantees. Extensive evaluation results demonstrate that our method significantly outperforms existing approaches, achieving up to an 84.5% improvement in schedulability on average.**

I. INTRODUCTION

The growing demands of emerging real-time applications have driven the shift from single-core to multicore embedded systems, where tasks frequently access shared resources due to functional needs and inherent resource constraints. These shared resources include both local and global resources. Local shared resources reside within a single core but may be accessed by multiple threads on that core, such as per-core queues or synchronisation flags, whereas global shared resources span multiple cores and include memory-mapped I/O devices and system-wide data structures. To avoid race conditions, such resources are typically protected within critical sections, which are code segments that require mutual exclusion. Resource-locking protocols are designed to guarantee mutually exclusive access for tasks while maintaining the timing requirements of the systems [1].

Embedded systems in critical domains must continue to function correctly in the presence of faults. Transient faults, which are temporary errors that occur during execution due to hardware fluctuations, timing anomalies, or other short-lived conditions, are of particular concern [2]. They are often tolerated by temporal redundancy (e.g., re-execution

or checkpointing, where checkpointing enables small-scope re-execution), or spatial redundancy (e.g., replication) [3]. Transient faults during critical sections must be handled to prevent error propagation across tasks. Conventional resource-locking protocols do not explicitly address fault tolerance [1]. Directly integrating checkpointing into resource-sharing protocols can significantly exacerbate global resource contention [4], primarily because fault recovery may substantially prolong the execution of critical sections, leading to locks being held for extended periods and thus increasing blocking times for other tasks in the system.

To address this challenge, MSRP-FT [4] combines checkpointing and replication, allowing spinning tasks to execute replicas in parallel and thereby reducing time lost to transient faults during critical-section execution. However, resource requests are still served sequentially in FIFO order, which can cause prolonged blocking under frequent faults. Moreover, coordinating replica execution introduces additional overhead, and the reliance on identical replicas makes the system vulnerable to correlated or timing-sensitive faults. These limitations are examined in detail in Section III-C.

In this paper, we propose a Lock-frEe Fault-Tolerant Resource Sharing protocol for multicore real-time systems, referred to as LEFT-RS. We record resource request orders in a FIFO queue. Instead of managing resource access sequentially as in MSRP-FT, LEFT-RS adopts a lock-free design that enables tasks to concurrently read global resources and initiate local execution. Tasks re-execute their critical sections either when they incur faults or when the shared resource has been updated, to ensure fault-free execution and avoid race conditions. Each task can complete its access upon successful execution when preceding tasks in the FIFO queue encounter faults. LEFT-RS facilitates resource access efficiency and ensures that remote blocking (blocking caused by tasks on remote cores due to resource contention) does not worsen as fault occurrences increase. Moreover, its design avoids the heavy coordination overhead and enhances fault resilience. To guarantee timing predictability, we provide a comprehensive worst-case response time analysis. Extensive experimental evaluations demonstrate that our method outperforms the state-of-the-art by up to 84.5% on average in terms of schedulability.

The following sections are structured as follows: Section II presents the system model. Section III reviews related work and provides an in-depth analysis of the limitations of state-of-the-art approaches. Section IV introduces the proposed protocol, while Section V offers a theoretical comparison with

the state-of-the-art approach. Section VI presents the worst-case response time analysis. Section VII shows the evaluation results and Section VIII concludes the paper.

II. SYSTEM MODEL

We consider systems composed of a set of identical cores, denoted as Λ, and a set of sporadic tasks, Γ, scheduled using the FP-FPS (Fully-Partitioned Fixed-Priority Scheduling) scheme. Each core in Λ is represented as λ_k, where k identifies the index of the core. Each task τ_i (the ith task in Γ) is characterized by the tuple $\tau_i = \{C_i, T_i, D_i, P_i\}$. Here, C_i represents the pure Worst-Case Execution Time (WCET) without accessing shared resources, T_i denotes the period (or minimum inter-arrival time), D_i is the constrained deadline satisfying $D_i \leq T_i$, and P_i indicates the priority of the task. Each task is assigned a unique priority, with higher values of P_i corresponding to higher priorities. For a given task τ_i, we denote tasks on the same core with higher and lower priorities as τ_h and τ_l, respectively. Tasks assigned to other cores are denoted by τ_j.

The system also includes a set of shared resources, denoted as $\mathcal{R}$, where the xth shared resource is represented as r^x. Each resource r^x is characterised by two parameters: c^x and N_i^x. The parameter c^x represents the computation length associated with r^x, while N_i^x indicates the number of requests made by task τ_i to r^x during a single release. This work does not consider nested resource sharing, meaning that a task can hold only one resource at a time. However, group locks [5] can be directly supported in the system.

We address transient faults in this study, which can be mitigated through redundancy techniques such as checkpointing or replication. Each transient fault is assumed to affect only one task at a time. Fault detection is performed at designated checkpoints using an acceptance test, such as result validation or consistency checking, to verify the correctness of task outputs [6]. The frequency of occurrence of transient faults can be modelled through various ways [7], which is out of the scope of this paper. Similar to [4], we assume each task can incur a maximum of f_i faults during any single release. The number of faults a task can incur during its access to r^x is denoted as $f_i^x \leq f_i$. We use n_i^x to denote the total execution number of a request of τ_i to r^x, which contains f_i^x failures and one successful execution ($n_i^x = f_i^x + 1$). All the notations are summarised in Table I.

III. RELATED WORK AND BACKGROUND

In this section, we first introduce the literature on resource sharing protocols and then review the state-of-the-art in fault-aware resource sharing.

A. Resource sharing protocols

Lock-based protocols have been widely studied for efficient resource sharing in multicore systems. The Multiprocessor Priority Ceiling Protocol (MPCP) extends traditional priority ceilings to reduce remote blocking [8]. Multiprocessor Stack Resource Protocol (MSRP) uses non-preemptive spinning to

TABLE I: List of Notations Used in the System Model

Notations	Descriptions
Λ	Set of identical cores in the system
λ_k	Core with index k, $\lambda_k \in \Lambda$
Γ	Set of sporadic tasks
τ_i	Task with index i, $\tau_i \in \Gamma$
τ_h, τ_l, τ_j	Tasks related to τ_i: higher/lower priority on the same core, or on a different core
C_i	Pure worst-case execution time (WCET) of τ_i
T_i, D_i	Period (minimum inter-arrival time) and constrained deadline ($D_i \leq T_i$) of τ_i
P_i	Priority of τ_i (a larger P_i means a higher priority)
$\mathcal{R}$	Set of shared resources in the system
r^x	Shared resource with index x
c^x	Computation length of r^x
N_i^x	Number of requests for r^x by τ_i per release
f_i, f_i^x, n_i^x	Number of faults incurred by τ_i per release. Fault and total execution number during access to r^x

avoid preemption delay [8]. In contrast, the Multiprocessor Resource Sharing Protocol (MrsP) employs a global priority ceiling to bound blocking, introduces a helping mechanism, and allows cross-core migration [9]. The Flexible Multiprocessor Locking Protocol (FMLP) distinguishes between short and long critical sections, combining spinning and suspension to balance blocking and concurrency [10]. The $O(m)$ Locking Protocol (OMLP) further refines priority-based mechanisms for improved scalability [11], and the Flexible Resource Accessing Protocol (FRAP) introduces dynamic spinning priorities and advanced blocking analysis for better performance [12]. Each protocol targets specific challenges, and their effectiveness depends on system architecture and application requirements.

In contrast, lock-free approaches [13] rely on atomic operations, such as Compare-and-Swap (CAS) or Load-Link/Store-Conditional (LL/SC), to update shared resources without using explicit locks. Instead of enforcing mutual exclusion through serialized access, these algorithms detect conflicts by monitoring changes to shared data and retry operations when necessary. This non-blocking design ensures system-wide progress while preventing race conditions and data corruption.

B. Fault-aware resource sharing

The work in [14] addresses transient faults in critical sections but adopts simple checkpointing, which tolerates faults through repeated sequential re-execution of resource requests. Building upon the traditional MSRP [15], MSRP-FT [4] efficiently tolerates transient faults within the critical sections of tasks in Mixed-Criticality Systems (MCS). Since the fault tolerance approach of MSRP-FT is independent of MCS, we will discuss this approach within a standard system setup to ease the presentation.

As shown in Figure 1, MSRP-FT establishes checkpoints not only at the beginning and end of each task but also around critical sections within a task's execution. These checkpoints divide execution into normal sections (no shared resources) and critical sections, where shared resources are accessed. Critical sections include local (intra-core) and global (inter-core) resource access. Each checkpoint detects faults for the
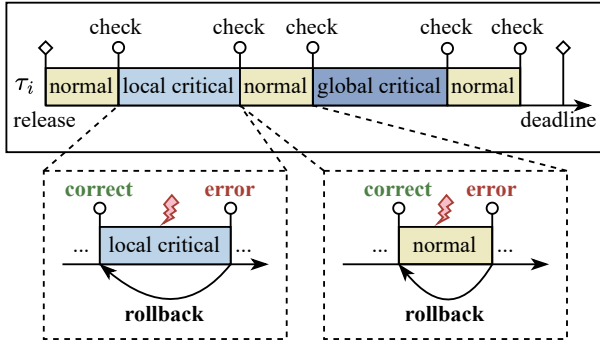
Fig. 1: Fault tolerance of normal and local critical sections

previous segment and saves the states for the next segment (if applicable). The setup enables faults to be tolerated in time after the execution of a critical section, preventing transitive errors from affecting other tasks and avoiding unnecessary large-scale re-executions.

If a fault is detected in a normal section, the system rolls back to the most recent checkpoint and re-executes the faulty segment only. When managing local shared resources, the same ceiling protocol is adopted from MSRP. Each local resource r^x is assigned a ceiling priority equal to the highest priority of any task that accesses it. When a task locks a local resource, its priority is temporarily elevated to the resource's ceiling priority. As shown in Figure 1, the fault-tolerance approach applied is the same as a normal section where the system initiates rollback and re-execution.
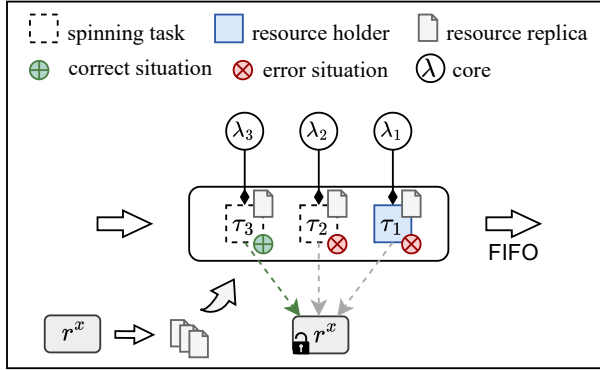

Fig. 2: Fault tolerance of global critical sections

Additionally, MSRP-FT incorporates a novel fault-tolerance mechanism specifically designed for global resources. As shown in Figure 2, when a task requests a global resource, it enters a FIFO queue, where the resource is allocated in FIFO order. The task at the head of the queue (i.e., τ_1) becomes the resource holder, while other tasks remain in a busy-wait state, spinning. The resource holder reads the shared resource and duplicates its execution into replicas, enabling all spinning tasks (i.e., τ_2 and τ_3) to assist in executing its critical sections in parallel. Replicas are executed locally, once a fault is detected, the replica is re-executed on the same core. A resource update is performed as an atomic action and only fault-free executions (i.e., τ_3) are eligible for updating
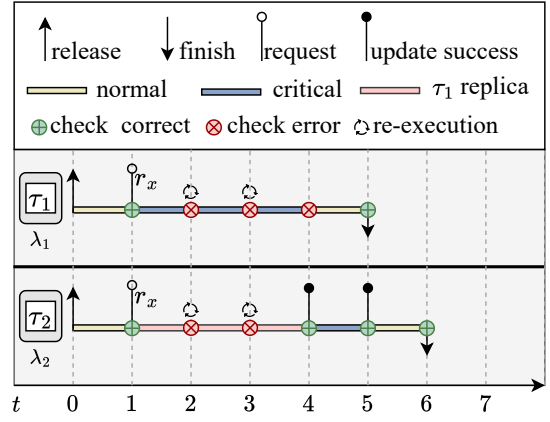

Fig. 3: Worst-case resource accessing of τ_2 under MSPR-FT

the resource. Once the resource is updated, all replicas are terminated, the head task exits the queue, and the next task in line (i.e., τ_2) repeats the process.

The amount of time that a task needs to execute all its n_j^x executions is calculated as $\lceil n_j^x/k_j \rceil \cdot c^x$ [4], where k_j represents the number of execution units available (including the task itself and helper tasks positioned behind it in the FIFO queue). To intuitively illustrate the performance of MSRP-FT, we demonstrate the following example which indicates the worst-case resource accessing time of τ_2.

Example 1. *As shown in Figure 3, τ_1 with $n_1^x = 6$ executes on λ_1, and τ_2 with $n_2^x = 1$ executes on λ_2. At time $t = 1$, both tasks simultaneously request access to r^x. The critical section length is $c^x = 1$. τ_1 becomes the head of the FIFO queue, with τ_2 following. Both begin executing their critical sections concurrently, with τ_2 assisting by executing a replica of τ_1's critical section. After repeated failures and retries, τ_2 successfully executes the 6th execution at $t = 4$, updates the resource, and helps τ_1 complete, allowing τ_1 to leave the FIFO queue at $t = 4$ and finally finishes at $t = 5$. At $t = 4$, τ_2 then executes its own critical section. As no other tasks are queued, it executes alone and, with $n_2^x = 1$, completes successfully and updates at $t = 5$. τ_2 finally finishes at $t = 6$.*

Under MSRP-FT, without considering coordination overhead, the worst-case resource-access time for a given task τ_i (e.g., τ_2 in Example 1) is determined by the following two conditions. First, τ_i suffers the maximum possible remote blocking (blocking from remote cores due to global resource contention) by being placed at the end of the FIFO queue, with no external helpers available which means it must execute its n_i^x replicas sequentially on its own core. Second, the maximum number of remote requests (denoted as m) ahead of τ_i in the FIFO queue are arranged such that tasks with higher execution counts n_j^x appear closer to the end of the queue with fewer helpers. Specifically, the first remote request (with the largest n_j^x) is assisted by only $k_1 = 2$ cores (including itself and τ_i), the second by $k_2 = 3$, and so on. The worst-case access time of request τ_i is expressed in Equation (1) [4]. All terms are measured in units of c^x, the execution length

of a single critical section of resource r^x. Specifically, n_i^x is the number of re-executions caused by faults, and S_i^x is the blocking delay due to remote requests. We further observe that each remote request contributes at least one unit of c^x to the blocking, as $\lceil n_j^x/k_j \rceil \geq 1$. Hence, in the worst case, $S_i^x \geq m$.

$$E_i^x = (n_i^x + S_i^x) \cdot c^x, \quad S_i^x \geq m \quad (1)$$

Based on the characteristics of MSRP-FT we address the following corollary which will be useful later.

Corollary 1. *Under MSRP-FT, the remote blocking units S_i^x suffered by a given task τ_i accumulates with the increase of the n_j^x of the remote requests. More specifically, if more than one remote request has $\lceil n_j^x/k_j \rceil > 1$, then $S_i^x > m + 1$. For example, if a task with $k_j = 2$ then incurs 2 faults (i.e. $n_j^x = 3$), it satisfies $\lceil 3/2 \rceil = 2 > 1$, thus contributing more than one unit of remote blocking.*

C. Limitations of state-of-the-art

The MSRP-FT protocol improves the fault tolerance efficiency by leveraging the parallel execution of replicas. While this approach improves fault recovery efficiency, it introduces new challenges as summarized below.

Limitation 1. *With a high number of faults, managing resource accesses sequentially, even with the helping mechanism, can still result in substantial delays.*

Although MSRP-FT accelerates fault tolerance by allowing spinning tasks to execute replicas in parallel, each task must still wait for the resource access in order. According to Corollary 1, when a remote task experiences frequent faults (i.e., a high n_j^x), it can significantly delay the execution of tasks behind it in the FIFO queue, i.e., increase the remote blocking ($S_i^x \cdot c^x$).

Limitation 2. *Utilising spinning tasks to assist requires complicated scheduling and may incur substantial overhead.*

MSRP-FT introduces unavoidable coordination overhead due to its helping mechanism. When a task is at the head of the FIFO queue, it must construct a shared operation descriptor, we call it *wrap*, which includes a function pointer for the critical section logic, a reference to the shared resource, and any relevant execution context [16], [17]. This structure is written to a globally visible memory location such as DRAM or the last level cache, and a readiness flag is set using a memory barrier to ensure cross-core visibility [18]. This step contributes to the per-head metadata setup cost, denoted O_{wrap}.

Each task behind the head in the queue must monitor the shared readiness flag and, once available, retrieve the wrap, copy the shared resource locally, and execute the head task's critical section. This cooperative execution introduces the per-replica overhead O_{replica}, which includes coordination, local preparation, and control transfer costs [16], [17]. When the task later becomes the head itself, it must construct its own wrap, incurring a one-time setup cost denoted as $O_{\text{self\_wrap}}$. Therefore, if a task is preceded by m other tasks in the FIFO

queue, the total overhead it incurs for a single resource request is given by:

$$O_{\text{total}}(m) = m \cdot (O_{\text{wrap}} + O_{\text{replica}}) + O_{\text{self\_wrap}} \quad (2)$$

Limitation 3. *Replica-based parallel execution, which relies on identical execution logic and synchronized data at the same time point, may be ineffective against deterministic or common-mode faults.*

In MSRP-FT, all replicas execute identical operations derived from the same wrap descriptor, leading to highly synchronized control flow and memory access patterns. Consequently, faults triggered by specific execution sequences or timing interference may simultaneously affect all replicas, causing coordinated failure.

IV. LEFT-RS PROTOCOL

In this section, we propose LEFT-RS for multicore real-time systems. The LEFT-RS introduces novel mechanisms to handle faults in accessing global resources more efficiently, aiming to improve the system's schedulability. We first introduce the basic setup which includes the part without global resource sharing. Then, we show how LEFT-RS manages global resource sharing with faults and its design rationale.

A. Normal and local critical sections

To avoid transitive errors and prevent unnecessary resource contention caused by re-executing the entire task, we continue to adopt the checkpointing mechanism from MSRP-FT, as illustrated in Section III-B. Checkpoints are placed not only at the beginning and end of each task, but also around critical sections within the task's execution. These checkpoints divide execution into normal sections (without shared resources) and critical sections (global and local). Each checkpoint, if applicable, detects faults in the preceding segment and saves the state for the next segment. In cases where a task has few or no critical sections, resulting in large normal sections, checkpoints can be introduced to subdivide them and reduce re-execution range. However, since our focus is on fault tolerance in critical sections, we do not further explore checkpoint placement within normal sections.

If a fault occurs in the normal or local critical section, the system simply applies rollback and re-execution. For the management of local resources, we assume a ceiling priority protocol, where a task accessing a resource has its priority raised to the ceiling priority, which is equal to the highest priority among local tasks that access the same resource.

B. Global resource sharing

In this subsection, we present how global resource sharing is managed by LEFT-RS through a series of rules. The presentation order of these rules does not follow the execution order of a task when accessing shared resources, instead, it follows the design rationale. In the end, Example 2 will show the overview of the entire execution sequence of tasks following the rules.

The core objective of the protocol is to improve resource access efficiency under fault-tolerant conditions. To achieve this, the protocol aims to allow tasks that complete without faults to update the global resources earlier when others encounter faults. This reduces unnecessary delays and enhances overall system throughput. Enabling concurrent execution of critical sections is key to achieving this goal. This is realized through a lock-free approach, as described in Rule 1, which is feasible as concurrent reads do not modify the shared state.

Rule 1. *When requesting a global resource, all tasks are allowed to read the resource simultaneously and proceed to their critical sections locally without acquiring a global lock.*

While concurrent execution improves efficiency, it introduces two key challenges that must be addressed: (1) concurrent updates to global resources, which may result in data races, and (2) the use of stale data, where tasks operate on outdated information due to limited update visibility.

The first concurrency issue is mitigated by Rules 2 and 3. In Rule 2, the FIFO queue records the order of task requests to regulate the update sequence, rather than enforcing resource accessing order as in MSRP-FT. Rule 3 ensures update priority in FIFO order to provide fairness and restricts resource updates to a single task at a time, thus avoiding race conditions.

Rule 2. *When a task accesses a global resource, it is added to a FIFO queue.*

Rule 3. *Upon successful (fault-free) execution, if multiple tasks request an update concurrently, the update is granted in FIFO order, and updates must be performed atomically.*

The second concurrency issue is resolved by Rule 4, which ensures that each update triggers the re-read and re-execution of other tasks in the FIFO queue to avoid outdated execution.

Rule 4. *When a task successfully updates the global resource, it leaves the FIFO queue. All other tasks operating on outdated copies of the resource are notified to abort their current executions, discard local copies, and restart their critical sections using the updated global resource.*

In addition, tasks also re-execute due to transient faults encountered during their critical sections as defined in Rule 5. This leads to two types of re-execution behaviours of critical sections as shown in Definitions 1 and 2.

Rule 5. *Upon unsuccessful (faulty) execution, the task re-executes the faulty critical section individually.*

Definition 1. *[Data-Induced Re-execution] Re-execution is triggered when another task updates the shared resource, invalidating the current task's local copy and requiring it to restart with the latest data.*

Definition 2. *[Fault-Induced Re-execution] Re-execution is triggered when a task detects a transient fault during its critical section, requiring a retry to ensure correctness.*

The main concurrency issues are addressed by regulating concurrent updates and data consistency enforcement. However, as resource accesses can start and finish at different times due to variations in request arrivals and execution paths, tasks may complete their executions out of order. Thus, regulating the timing of updates and re-executions remains critical to prevent unfairness and unpredictable delays. To address this issue, the protocol introduces a comprehensive synchronisation mechanism composed of three tightly integrated rules. First, Rule 6 ensures that tasks joining the FIFO at different times begin execution simultaneously, aligning their execution windows and reducing initial timing gaps.

Rule 6. *Upon joining the FIFO queue, if the head task in the queue is in the middle of executing a critical section, subsequent tasks enter a synchronisation period and wait for the head task to finish its current execution; otherwise, the synchronisation step is skipped.*

However, tasks may still finish their critical sections earlier than expected. For example, consider a scenario where τ_a, τ_b, and τ_c are placed in FIFO order from front to end. If τ_c completes earlier and updates the resource, this may trigger data-induced re-executions of τ_a and τ_b. Furthermore, later-joining tasks may starve τ_a and τ_b for the same reason. To address this issue, Rule 7 governs the update of a shared resource by tasks upon successful execution. Even if a task finishes earlier, it cannot update the global resource until all preceding tasks have either been completed or encountered faults. This prevents early completion from triggering unnecessary data-induced re-execution of preceding tasks and ensures update fairness, which completes Rule 3.

Rule 7. *Upon successful (fault-free) execution, the head task in the FIFO queue can directly attempt to update the shared resource. Other tasks (apart from the head) can attempt to update only after confirming that all preceding tasks have encountered faults during their current execution.*

In this scenario, suppose τ_c completes successfully. According to Rule 7, it must wait for τ_a and τ_b to finish before updating. If τ_a completes first but encounters faults and immediately re-executes, then when τ_b eventually finishes, τ_c must still wait for τ_a's new execution, which introduces additional delays. Therefore, Rule 8 complements Rule 5 by controlling fault-induced re-executions timing. This prevents re-executing tasks from interfering with later tasks' updates, avoiding serial delays from unsynchronised retries.

Rule 8. *Upon unsuccessful (faulty) execution, the task waits for all tasks in the FIFO queue to finish their current executions and then re-executes the faulty critical section by itself.*

These design elements work collectively to realize two core rationales that govern the overall protocol behaviour.
- *Rationale 1*: A task is re-executed due to the update of the global resource by preceding tasks in the FIFO queue.
- *Rationale 2*: A task is re-executed due to an update made by later-arriving tasks. However, such re-executions overlap with fault-induced re-executions.

Rationale 1 ensures fairness and bounded re-execution: tasks re-execute due to their relative positions in the FIFO queue. Under Rule 4, each update to the global resource triggers the departure of a task from the FIFO queue. Therefore, the number of re-executions is predictable. Rationale 2 is the most critical for achieving efficient resource access under fault tolerance. If a preceding task encounters faults, it will re-execute regardless. If this re-execution overlaps with the resource update made by a later-arriving task, it benefits the later-arriving task by reducing delay without adversely affecting the preceding task. Since the forced waiting time occurs only when tasks complete earlier than expected, the total of execution and waiting time still falls within the duration of a single critical section, which will be further proved in Lemma 4.

Finally, Rules 9 and 10 complete the protocol. Rule 9 can be implemented by raising its active priority to the system's maximum level, ensuring it is not preempted during global resource access. This behaviour is inherited from MSRP [15], chosen for its simplicity. While ceiling-based approaches, such as MrsP [9], may further reduce arrival blocking, they are beyond the scope of this study.

Rule 9. *When a task requests a global resource, it becomes non-preemptive with respect to local tasks and remains so until it completes the resource access.*

Rule 10 completes Rule 6 by eliminating unnecessary delays. We propose Lemma 1 to support Rule 10.

Rule 10. *The initial synchronisation period may be skipped if a task joins the FIFO queue with the assurance that all preceding tasks will not incur faults (i.e., $n_j^x = 1$, or they have already encountered the maximum number of allowable faults), or if the system's fault-tolerance mode is disabled.*

Lemma 1. *For a given task τ_i, if its preceding tasks in the FIFO queue will not incur faults, then the request of τ_i does not need to enter the synchronisation period.*

Proof. As the preceding tasks must start no later than τ_i, a resource update must be made within an execution period c^x from preceding tasks since they will not encounter faults. Even if τ_i finishes earlier, Rule 7 is sufficient to guarantee that the task can only update the resource when the preceding tasks finish, which aligns with Rationale 1 and 2. □

These rules are visualized by Example 2 which illustrates the complete execution sequence of a task request for a global resource under LEFT-RS.

Example 2. *As shown in Figure 4, τ_1 and τ_2 are executing on cores λ_1 and λ_2, respectively. At $t = 1$, τ_1 requests access to r^x with $c^x = 2$. It becomes locally non-preemptive and starts executing its critical section immediately (Rules 1 and 9) without a synchronisation period (Rule 6). At $t = 2$, τ_2 requests access to r^x and the current FIFO queue order becomes τ_1 and τ_2, from front to end. According to Rule 6, since τ_1 is in the middle of its critical section execution at*
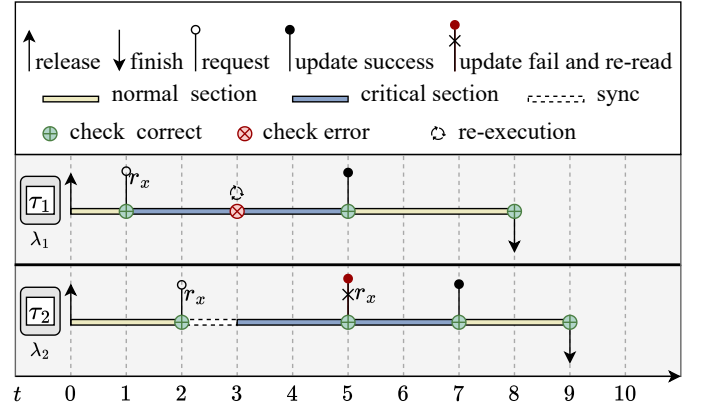

Fig. 4: Example of global resource management of LEFT-RS

$t = 2$, τ_2 enters a synchronisation period. At $t = 3$, as τ_1 fails its execution and all executions are synchronised, τ_1 re-executes (Rules 5 and 8) and τ_2 reads the shared resource and begins executing its critical section in parallel with τ_1 (Rule 1). At $t = 5$, the executions of τ_1 and τ_2 are correct and synchronised, τ_1 successfully updates r^x, in accordance with the update procedure (Rules 3 and 7). At $t = 5$, after the resource update, τ_1 leaves the FIFO queue (Rule 4) and completes its execution at $t = 8$. Also, at $t = 5$, τ_2 is signalled to re-read and re-execute the shared resource (Rule 4). At $t = 7$, τ_2 updates r^x successfully (Rules 3 and 7), leaves the FIFO queue (Rule 4), and finishes at $t = 9$.

Overall, LEFT-RS allows tasks to start resource accessing concurrently without being served in FIFO order. Each task can exit the FIFO queue earlier upon successful execution if the preceding tasks incur faults. Such a design can effectively address Limitation 1. Moreover, LEFT-RS avoids the coordination overhead described in Equation (2) by eliminating cooperative replica execution. Tasks do not construct or publish shared wrap descriptors, nor do they assist in executing other tasks' critical sections. Instead, each task executes its own critical section independently after retrieving the shared resource state which addresses Limitation 2. Other overheads, such as local fault detection and thread communications, are omitted from the discussion since they are either common to both protocols or have negligible performance impact. Since tasks under LEFT-RS operate on their own logic and execution paths, faults affecting one task are less likely to propagate to others. This design inherently reduces the risk of synchronised failure and improves fault isolation across concurrent executions which addresses Limitation 3. The effectiveness of the proposed rules and rationales will be further demonstrated in the worst-case analysis of resource access in Section V.

C. Implementation discussion

The lock-free approach is well-established and has a long history [13]. LEFT-RS is applicable to shared resources where tasks can read a consistent snapshot and perform local computation before a single atomic update. Typical examples include

status flags, control registers, configuration parameters, and shared counters. These structures are common in embedded systems and align with LEFT-RS's design, which favours safe concurrent reads and avoids complex coordination. Furthermore, both LEFT-RS and MSRP-FT [4] require atomic updates and execute critical sections concurrently and locally, meaning their theoretical applicability spectra are the same.

LEFT-RS requires each task to copy the shared resource into its private memory (e.g., L1/L2 cache) and execute independently, which is supported by most commercially off-the-shelf (COTS) architectures. From a software perspective, Rule 4 involves signalling other tasks, while Rules 6, 7, and 8 involve checking the execution status of other tasks. Such coordination can be achieved through lightweight communication mechanisms, such as shared status flags or conditional signalling.

V. THEORETICAL COMPARISON WITH MSRP-FT

The key distinction between the proposed approach and MSRP-FT lies in the management of global resource access. Before carrying out the full analysis, in this section, we analyse the advantages of the proposed approach by examining the worst-case resource accessing time for one single request of a given task τ_i to a global resource r^x.

According to Rule 5, each task under the proposed approach re-executes independently without assistance when it incurs faults. Therefore, in the worst-case scenario without accounting for the delay caused by remote tasks, a resource request from τ_i will execute n_i^x times its critical sections, consisting of f_i^x failed executions and one successful submission.

According to Rules 4 and 7, each task will perform data-induced re-execution caused by the resource updates from preceding tasks in the FIFO queue. Each resource update will lead to the leave of a task from the FIFO queue. Therefore, the amount of data-induced re-execution can be bounded by the following lemma.

Lemma 2. *For a given request from a task τ_i to r^x, if there are at most m ($\leq |\Lambda|$) remote requests that can request r^x at the same time, then the request of τ_i can incur at most m instances of data-induced re-execution.*

Proof. Data-induced re-execution occurs due to a successful resource update in the system. In the meantime, each update removes a task from the FIFO queue (Rule 4). There are at most m instances of preceding tasks in the queue, therefore, the request of τ_i can incur at most m data-induced re-executions from preceding tasks. According to Rule 7, later-arriving tasks can only cause data-induced re-execution to τ_i when it incurs faults. This is already accounted for in the f_i^x failed executions due to faults. $\square$

According to Rule 6, each task may enter the synchronization period when requesting a resource. We propose the following lemma to bound the synchronization period.

Lemma 3. *The maximum synchronization overhead for a given resource request is one unit of c^x.*

Proof. A resource request incurs synchronization overhead when it joins the FIFO queue and waits for the head task to complete its current execution. Since the maximum execution length is c^x and synchronization occurs at most once, the maximum synchronization overhead is c^x. $\square$

Lemma 4. *The synchronisation overhead is the only additional waiting time.*

Proof. According to Rule 7, a task may need to wait for the preceding tasks to finish when executed correctly. However, Rule 6 forces all tasks to start simultaneously; therefore, the waiting period only occurs when the task finishes earlier, and the sum of its execution and waiting time will not exceed c^x. The same reason also applies to Rule 8, when a task incurs faults, it waits for all tasks to finish before carrying out re-execution. $\square$

According to Lemmas 2, 3 and 4, we can have the following corollary. In Corollary 2, n_i^x is the fundamental worst-case execution of $\tau_i's$ request without accounting for the delay from remote cores. The m accounts for data-induced re-executions as demonstrated in Lemma 2. The 1 unit of c^x is the synchronization overhead as illustrated in Lemma 3.

Corollary 2. *For a given resource request to r^x from τ_i under LEFT-RS, its worst-case resource accessing time can be upper bounded as $E_i^x = (n_i^x + m + 1) \cdot c^x$.*

According to Lemma 1, the synchronization overhead may not happen when preceding tasks of τ_i in the FIFO queue do not encounter faults, and we can have the following corollary.

Corollary 3. *For a given resource request of τ_i to r^x, if preceding tasks in the FIFO queue incur no faults, the worst-case resource accessing time under the proposed approach will be $E_i^x = (n_i^x + m) \cdot c^x$.*

As shown in Equation (1), the worst-case resource accessing time for a given request from τ_i under MSRP-FT is $E_i^x \geq (n_i^x + m) \cdot c^x$, which indicates that it can be better (i.e., smaller) than the proposed bound $E_i^x = (n_i^x + m + 1) \cdot c^x$ in limited scenarios. More specifically, according to Corollary 1 and 2, we can have the following corollary.

Corollary 4. *For a request from τ_i to r^x, LEFT-RS outperforms MSRP-FT as the number of faults increases. More specifically, this holds when more than one remote request satisfies $\lceil n_j^x / k_j \rceil > 1$. Unlike MSRP-FT, LEFT-RS maintains a fixed number of remote delay units at m, ensuring that its remote blocking does not grow with the fault count.*

To visualize the effectiveness of the proposed approach, we illustrate the following example with the same setup of Example 1 which illustrates the worst-case resource accessing time of τ_2 as well.

Example 3. *As shown in Figure 5, τ_1 with $n_1^x = 6$ executes on λ_1, and τ_2 with $n_2^x = 1$ executes on λ_2. At time $t = 1$, both tasks simultaneously request access to r^x. Under LEFT-RS, both tasks begin executing their own critical sections*
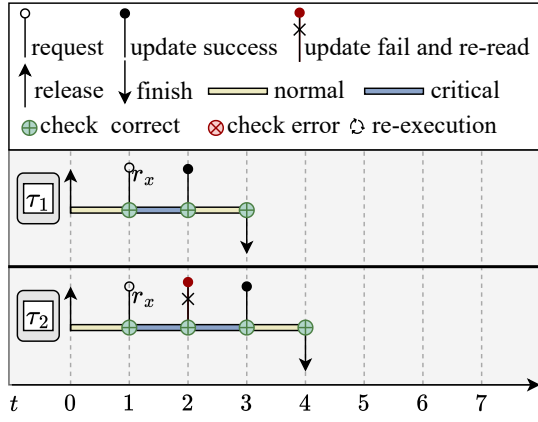
Fig. 5: Worst-case resource accessing of τ_2 with LEFT-RS

concurrently (Rule 1). They both complete successfully at $t = 2$, with τ_1 updates r^x, while τ_2 is blocked (Rule 3). This represents the worst case for τ_2, as it would have updated r^x if τ_1 had failed. At $t = 2$, τ_2 re-reads r^x and re-executes (Rule 4). It completes accessing r^x at $t = 3$ and finishes at $t = 4$, which is earlier than the $t = 6$ completion in Example 1.

In addition, the Corollary 4 and Example 3 are based on the theoretical worst-case response time analysis of MSRP-FT as introduced in [4]. The unavoidable overhead associated with MSRP-FT, as described in Equation (2), has not yet been taken into account. The overall performance comparisons will be demonstrated in Section VII.

VI. SCHEDULABILITY ANALYSIS

In this section, we derive the worst-case response time analysis of LEFT-RS. The notations specific to this analysis are listed in Table II, while general notations reused from earlier sections can be found in Table I.

As shown in Equation (3), R_i represents the worst-case response time of a given task τ_i. C_i represents the pure WCET of τ_i without accessing any shared resource. E_i is the total resource-accessing time of τ_i and local tasks with higher priority than τ_i ($lhp(i)$). B_i is the arrival blocking incurred by τ_i upon arrival. F_i denotes the amount of time τ_i spends tolerating faults. The term $\lceil R_i/T_h \rceil$ denotes the maximum number of times a local high-priority task $\tau_h \in lhp(i)$ can preempt τ_i during R_i. C_h and F_h are the pure WCET and fault-tolerance time of τ_h per release instance, respectively.

$$R_i = C_i + E_i + B_i + F_i + \sum_{\tau_h \in lhp(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot (C_h + F_h) \quad (3)$$

We first introduce how to bound E_i which includes the resource-accessing time of τ_i and tasks in $lhp(i)$, as in the worst-case tasks in $lhp(i)$ preempt τ_i and finish their resource accessing during preemption which transitively delays τ_i [5]. As shown in Equation (4), E_i iterates over all the shared resources in the system ($\mathcal{R}$). For a given resource r^x, $N_{i,local}^x$ represents the total number of requests from τ_i and tasks in $lhp(i)$ to r^x and they are called local requests. We made an assumption here that every local request has an execution $n_i^x = 1$

TABLE II: Table of Notation for Schedulability Analysis

Notations	Descriptions
R_i	Worst-case response time of τ_i
E_i	Total resource-accessing time of τ_i and $lhp(i)$
B_i	Arrival blocking incurred by τ_i
F_i, F_h	Fault-tolerance time for τ_i and $\tau_h \in lhp(i)$
$lhp(i), llp(i)$	Set of higher/lower-priority tasks relative to τ_i on the same core
$\lambda(\tau_i)$	Core where τ_i is allocated
$\Gamma(\lambda_k)$	Tasks allocated to core λ_k
$N_{i,local}^x$	Number of requests issued by τ_i and $lhp(i)$ to r^x
$\eta_i^x(R_i, R_j)$	List of τ_j's requests to r^x over time periods R_i and R_j
ξ_{i,λ_k}^x	Remote requests (n_j^x) that can block τ_i and $lhp(i)$ from λ_k
$\mathcal{N}_{i,\lambda_k}^x$	Maximum number of remote requests blocking τ_i and $lhp(i)$ from λ_k
$\mathcal{E}_i^x$	Total list of n_j^x of remote requests that can block τ_i and $lhp(i)$
Syn_i^x	Synchronisation overhead incurred by local requests to r^x
α_i^x	Largest n_j^x from $llp(i)$ for r^x
β_i^x	The list of remote n_j^x that can block α_i^x
$\mathbb{I}(\cdot)$	An indicator function that returns 1 if the condition inside holds true and 0 otherwise
$F^A(i)$	Resources imposing arrival blocking on τ_i
$F(\tau_i)$	Resources accessed by τ_i

because the worst-case fault-tolerance time is accounted in F_i and F_h. $N_{i,local}^x$ is further expanded in Equation (5). $|\mathcal{E}_i^x|$ gives the total number of remote blocking requests incurred by local requests which is illustrated through steps in Equations (6), (7) and (8). Syn_i^x is the synchronisation overhead experienced by local requests as explained in Equation (9).

$$E_i = \sum_{r^x \in \mathcal{R}} (N_{i,local}^x + |\mathcal{E}_i^x| + Syn_i^x) \cdot c^x \quad (4)$$

As shown in Equation (5), $N_{i,local}^x$ includes the total number of requests from τ_i to r^x (N_i^x). During the release time of τ_i (denoted by R_i), τ_i may be preempted by tasks in $lhp(i)$, it additionally accounts for the requests from all $\tau_h \in lhp(i)$ by including the corresponding N_h^x of each τ_h.

$$N_{i,local}^x = N_i^x + \sum_{\tau_h \in lhp(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot N_h^x \quad (5)$$

$|\mathcal{E}_i^x|$ denotes the size of $\mathcal{E}_i^x$ which is worst-case list of remote requests in terms of n_j^x that can block local requests. In Equation (6), $\lambda(\tau_i)$ denotes the core where τ_i is allocated. $\mathcal{E}_i^x$ iterates over all the remote cores of τ_i (i.e., $\lambda_k \in \Lambda, \lambda_k \neq \lambda(\tau_i)$) and takes the first $\mathcal{N}_{i,\lambda_k}^x$ requests from ξ_{i,λ_k}^x.

$$\mathcal{E}_i^x = \bigcup_{\lambda_k \in \Lambda, \lambda_k \neq \lambda(\tau_i)} \bigcup_{p=1}^{\mathcal{N}_{i,\lambda_k}^x} \xi_{i,\lambda_k}^x(p) \quad (6)$$

ξ_{i,λ_k}^x as shown in Equation (7) represents a non-increasing list of total requests in terms of n_j^x from a given remote core λ_k. It iterates over tasks on λ_k (denoted as $\Gamma(\lambda_k)$) and takes n_j^x of each request to r^x. The notation $\eta_j^x(R_i, R_j)$ denotes the

list of n_j^x of requests from a task τ_j executing on a remote core λ_k during the period R_i and R_j, which accounts for the back-to-back hit [5]. It follows that $|\eta_j^x(R_i, R_j)| = \lceil(R_i + R_j)/T_j\rceil \cdot N_j^x$ where $|\cdot|$ denotes the size of the list (i.e., the number of requests), $\lceil(R_i + R_j)/T_j\rceil$ represents the number of times τ_j is released, and N_j^x is the number of times τ_j accesses r^x in a single release. For instance, $\eta_j^x(R_i, R_j) = \{2, 2, 2\}$ indicates that there are three requests with $N_j^x = 3$ and all requests have execution numbers $n_j^x = 2$.

$$\xi_{i,\lambda_k}^x = \bigcup_{\tau_j \in \Gamma(\lambda_k), \lambda_k \neq \lambda(\tau_i)} \eta_j^x(R_i, R_j) \tag{7}$$

As each core can manage a request at a time, each local request can have at most one request from a remote core to place ahead of it in the FIFO queue [5]. Therefore the maximum of requests from λ_k that can block local requests is denoted as $\mathcal{N}_{i,\lambda_k}^x$ as shown in Equation (8) which equals to the minimum number between local requests and the number of remote requests on a given remote core λ_k. As ξ_{i,λ_k}^x is a non-increasing list, the first $\mathcal{N}_{i,\lambda_k}^x$ items are the biggest n_j^x, which can result in the worst-case synchronisation overhead according to Lemma 1.

$$\mathcal{N}_{i,\lambda_k}^x = Min\{N_{i,local}^x, |\xi_{i,\lambda_k}^x|\} \tag{8}$$

According to Lemma 1, a request does not incur synchronisation overhead if all preceding requests have $n_j^x = 1$. The synchronisation overhead of local requests that must be accounted for is summarized in Equation (9), where $\mathbb{I}(\cdot)$ is an indicator function that returns 1 if the condition inside holds true and 0 otherwise. The first term in the equation represents the total number of preceding requests with $\mathcal{E}_i^x(p) > 1$. The second term, $N_{i,local}^x$, denotes the maximum possible synchronisation overhead as each local request can suffer at most one synchronisation overhead according to Lemma 3. The overall synchronisation overhead is then determined as the minimum of these two quantities.

$$Syn_i^x = Min\left\{\sum_{p=1}^{|\mathcal{E}_i^x|} \mathbb{I}(\mathcal{E}_i^x(p) > 1), N_{i,local}^x\right\} \tag{9}$$

The arrival blocking is denoted as B_i as shown in Equation (10). As the arrival blocking can only occur once upon the arrival of τ_i due to a local low-priority task $\tau_l \in llp(i)$ accessing a shared resource with a ceiling priority higher than the priority of τ_i or it is a global resource (non-preemptive). The set of resources that can impose arrival blocking to τ_i is denoted as $F^A(i)$. Then B_i iterates over all resources in $F^A(i)$ and finds out the maximum arrival blocking.

$$B_i = \max_{r^x \in F^A(i)} \left\{(\alpha_i^x + |\beta_i^x| + \mathbb{I}(\exists n_j^x \in \beta_i^x, \ n_j^x > 1)) \cdot c^x\right\} \tag{10}$$

The first term α_i^x in Equation (10) is expanded in Equation (11), which denotes the largest n_l^x of a request among requests of local low-priority tasks ($\tau_l \in llp(i)$) to a given resource $r^x \in F^A(i)$ (expressed as $N_l^x > 0$).

$$\alpha_i^x = \max\{n_l^x | \tau_l \in llp(i) \wedge N_l^x > 0\} \tag{11}$$

The second term indicates the size of β_i^x, which denotes the list of remote blocking requests incurred by α_i^x shown in Equation 12. β_i^x iterates over all the remote cores ($\lambda_k \in \Lambda, \lambda_k \neq \lambda(\tau_i)$) and takes the $\mathcal{N}_{i,\lambda_k}^x + 1$ request from ξ_{i,λ_k}^x if the length is satisfied ($\mathcal{N}_{i,\lambda_k}^x + 1 \leq |\xi_{i,\lambda_k}^x|$). This is because the first $\mathcal{N}_{i,\lambda_k}^x$ is already taken by E_i through Equation (6).

$$\beta_i^x = \bigcup_{\lambda_k \in \Lambda, \lambda_k \neq \lambda(\tau_i)} \left\{\xi_{i,\lambda_k}^x(\mathcal{N}_{i,\lambda_k}^x + 1) \mid \mathcal{N}_{i,\lambda_k}^x + 1 \leq |\xi_{i,\lambda_k}^x|\right\} \tag{12}$$

Finally, $\mathbb{I}(\exists n_j^x \in \beta_i^x, \ n_j^x > 1)$ checks whether any request in β_i^x has $n_j^x > 1$. If so, the function returns 1, indicating that an additional synchronisation overhead must be added as previously described; otherwise, it returns 0.

As the checkpoints divide tasks into segments and each task is assumed to incur at most f_i faults during the execution of the single release, the worst-case fault-tolerance scenario therefore is when all f_i faults happen on the longest segment of a task repeatedly [19]. Therefore, the worst-case fault-tolerance time for τ_i and $lhp(i)$ is denoted in Equation (13). It identifies the longest segment by taking the maximum value between the total execution time of normal sections C_i and c^x, which is the maximum critical section among the resources accessed by τ_i denoted as $F(\tau_i)$. The worst-case fault-tolerance time is calculated by f_i times the largest segment as each task re-executes by itself when encounters faults under LEFT-RS.

$$F_i = f_i \cdot \max\{C_i, \max\{c^x | r^x \in F(\tau_i)\}\} \tag{13}$$

VII. EVALUATION

In this section, we evaluate the effectiveness of the proposed approach by comparing the schedulability of systems under LEFT-RS, MSRP-FT, MSRP-FT-OF (Overhead Free), and Checkpointing. All methods adopt the same checkpoint placement strategy, ensuring a fair comparison without considering checkpointing overhead. Their difference lies solely in how they manage global resource sharing in the presence of faults. MSRP-FT and MSRP-FT-OF both employ the helping mechanism introduced in Section III-B. Specifically, MSRP-FT-OF does not account for protocol overhead, while MSRP-FT includes the overhead as defined in Equation (2). By contrast, Checkpointing requires tasks to simply re-execute sequentially when a fault occurs while holding a lock.

We implemented the evaluations by generating synthetic tests. We consider multicore platforms with a set of cores $M = [2, 4, 6, 8, 10, 12, 14, 16]$. For each core, we vary the number of tasks per core in the range of $N = [2, 3, 4, 5, 6, 7, 8, 9]$, which enables a proper range of schedulable systems for analysis and comparison. The utilisation of each task U_i is generated using the UUnifast algorithm [20], with a total system utilisation bound $U_i = 0.04 \cdot M \cdot N$. Task periods

(a) Schedulability with varied M

(b) Schedulability with varied $N \cdot M$

(c) Schedulability with varied rsf

(d) Schedulability with varied L

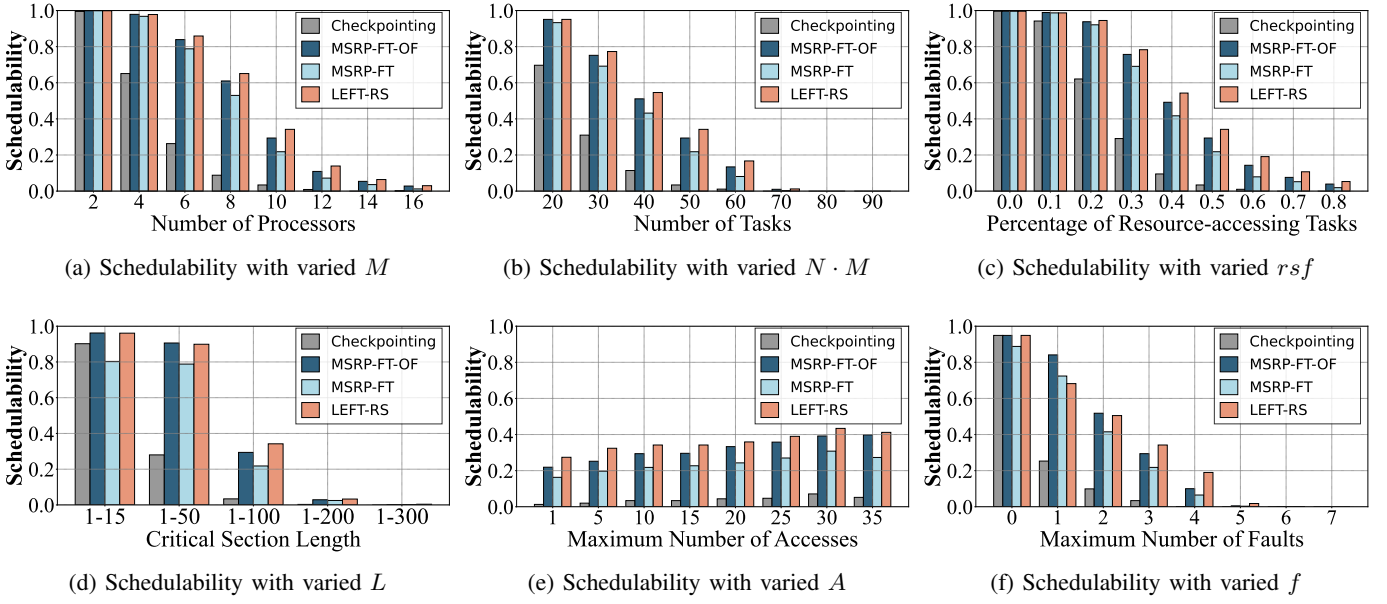(e) Schedulability with varied A

(f) Schedulability with varied f

Fig. 6: System schedulability with $N = 5$, $M = 10$, $A = 10$, $L = [1, 100]$, $rsf = 0.5$, $f = 3$ and $K = M$ resources

are randomly selected between [1ms,1000ms] in a log-uniform distribution. The deadlines for the tasks are equal to their periods. For a task τ_i, the total worst-case execution (including critical sections) is given by $\widehat{C}_i = U_i \times T_i$.

Each system contains K number of shared resources equal to the number of cores (i.e., $K = M$). Among the generated tasks, a ratio of tasks are selected to request a random number of resources (up to K). The ratio is denoted by resource-sharing factor $rsf \in [0, 0.8]$. The maximum number of accesses to a resource from each task is set to $A = [1, 5, 10, 15, 20, 25, 30, 35]$ for a single release. The worst-case computation time of a shared resource is generated randomly in the range $L \in [1\mu s, 300\mu s]$. We denote the total length of critical sections of τ_i as C_i^r, then we enforce $C_i = \widehat{C}_i - C_i^r \geq 0$ to obtain C_i. We set the maximum number of faults a task can incur during a single release to be $f \in [0, 7]$. For example, if $f = 5$, a task in one release can incur randomly from 0 up to 5 times faults. This setup follows the same fault model used in [4] to ensure a fair comparison. It is well-justified, as radiation and EMI studies confirm that a single energetic particle or disturbance can induce multi-bit or multi-node upsets, making multiple transient faults per task a realistic and critical scenario in safety-critical embedded systems [21]. In addition, tasks are allocated by the Worst-Fit heuristic and are scheduled by FP-FPS. Deadline Monotonic Priority Ordering is applied.

To parametrize the structural coordination overheads introduced by MSRP-FT, as discussed in Limitation 2, we rely on empirical measurements from real-time operating systems deployed on multicore platforms. The wrap construction overhead, denoted as O_{wrap} and $O_{\text{self\_wrap}}$ in Equation (2), arises from the cost of publishing a shared descriptor that contains function pointers, resource references, and synchronisation flags. The underlying operations include memory setup, global

visibility enforcement, and memory fencing. Shi et al. [16] report that enabling the helping mechanism in LITMUS$^{\text{RT}}$ incurs a wrap setup cost consistently below $1\,\mu s$. Similarly, a Linux-based fault-tolerant messaging prototype reports cross-core message publication latency of approximately $0.55\,\mu s$ [18]. Based on this evidence, we conservatively assign $O_{\text{wrap}} = O_{\text{self\_wrap}} = 1\,\mu s$ in our evaluation.

The per-replica execution overhead, O_{replica}, encompasses the time for a helper task to poll for wrap readiness, retrieve the operation descriptor, copy resource state locally, and execute the head task's critical section. Zhao et al. [17] measured this cooperative execution latency under MrsP on LITMUS$^{\text{RT}}$ at approximately $8.4\,\mu s$. Shi et al. [16] further decomposed this into $5.6\,\mu s$ for the migration itself and $1.5\,\mu s$ for context re-entry on the helper core. Even commercial RTOS platforms show comparable orders of magnitude, e.g., $11\,\mu s$ for Vx-Works and $13.4\,\mu s$ for RTLinux [22]. Based on these results, we assign $O_{\text{replica}} = 6\,\mu s$, which reflects a realistic and typical cost of helper-driven coordination and execution in multicore embedded systems, so as to avoid overstating the overhead in comparison with LEFT-RS.

In the following evaluations, for each combination of system settings, 1000 systems are generated, and the percentage of schedulable systems of the considered models is presented. In Figure 6, the y-axis is the rate of schedulable systems over 1000 generated systems (denoted as schedulability).

Observation 1: *As shown in Figure 6, LEFT-RS consistently outperforms the Checkpointing approach across all settings.*

Since the two approaches differ only in their fault-tolerance mechanisms for global resources, this performance gap validates that the direct integration of the conventional fault-tolerance technique will lead to unmanageable global resource contention. Although checkpointing reduces re-execution costs

by dividing tasks into segments, it does not mitigate contention in fault-tolerant global resource access. When faults occur, prolonged lock-holding from sequential re-executions can significantly delay other tasks, leading to system inefficiency.

Observation 2: *LEFT-RS approach consistently outperforms MSRP-FT in Figures 6a, 6b, 6c, 6d and 6e.*

As shown in Figure 6a, increasing the number of cores from 2 to 16 while keeping the per-core task count fixed at $N = 5$ reduces schedulability for all methods. This decline is mainly due to heightened contention for global resources as more cores and tasks compete for access.

The performance gap between LEFT-RS and MSRP-FT widens as the number of cores increases. For instance, with 4 cores, LEFT-RS successfully schedules 10 more systems than MSRP-FT, whereas with 6 cores, this difference grows to 71 systems. This suggests that as global resource contention intensifies, LEFT-RS more effectively manages both resource access and fault tolerance. Although MSRP-FT employs a fault-tolerance mechanism by utilising spinning tasks to execute critical sections in parallel and accelerate recovery, it still has some fundamental limitations. As discussed in Limitations 1 and 2 (Section III-C), tasks in MSRP-FT must manage resource accesses sequentially. If a task encounters a fault, it delays all subsequent tasks in the FIFO queue. Moreover, the unique overheads associated with MSRP-FT can be directly affected by the number of requests from different cores as illustrated in Equation (2). In contrast, LEFT-RS allows tasks to execute their critical sections independently without following FIFO order, which not only enables them to exit the FIFO queue upon successful update but also exempts them from the overheads associated with MSRP-FT. Overall, as shown in Figure 6a, LEFT-RS outperforms MSRP-FT by an average of 51.3%.

A similar trend is observed in Figures 6b, 6c, and 6d, where variations in system parameters, such as the total number of tasks, the proportion of tasks that share resources, and the length of critical sections, lead to increased global resource contention. In all these scenarios, LEFT-RS consistently outperforms MSRP-FT. Specifically, it achieves average improvements of 62.9%, 58.8%, and 84.5% in Figures 6b, 6c, and 6d, respectively. In addition, when $L = [1\mu s, 15\mu s]$ in Figure 6d, the Checkpointing outperforms MSRP-FT, indicating that when the critical section is small, re-executing sequentially is more cost-effective than applying a helping mechanism.

As shown in Figure 6e, an increase in the maximum number of resource accesses does not necessarily degrade schedulability. This is because, under system utilisation constraints, an increase in accesses can sometimes shorten the critical section length of shared resources generated. Similar effects have been observed in [4], [12]. Nevertheless, LEFT-RS continues to outperform MSRP-FT by an average of 53%.

Observation 3: *LEFT-RS loses its advantage over MSRP-FT when the number of faults in the system is very small.*

As shown in Figure 6f, when $f = 0$, Checkpointing performs exactly like traditional MSRP by definition, as no faults are considered in the system. LEFT-RS performs identically

TABLE III: Number of Systems Schedulable Only by MSRP-FT (left) or Only by LEFT-RS (right)

A	MSRP-FT ✓ LEFT-RS ✗	LEFT-RS ✓ MSRP-FT ✗	f	MSRP-FT ✓ LEFT-RS ✗	LEFT-RS ✓ MSRP-FT ✗
1	0	111	0	0	61
5	0	127	1	52	10
10	0	124	2	0	90
15	0	115	3	0	124
20	0	116	4	0	125
25	1	121	5	0	15
30	0	126	6	0	0
35	0	139	7	0	0

to Checkpointing because the synchronisation overhead is avoided, according to Rule 10 in Section IV. The additional overheads associated with MSRP-FT cause it to perform worse than the other protocols. When the maximum number of faults per task is randomly set to at most 1, where most resource accesses remain fault-free, LEFT-RS loses some of its advantages to MSRP-FT by an average of 6.2%. This behaviour matches the prediction in Corollary 4, where each remote request does not hold the resource-holder position for long due to rare faults, and the overhead of MSRP-FT is manageable compared to the critical section length in this scenario.

However, as f increases, the benefits of LEFT-RS become evident. In scenarios where LEFT-RS has an advantage (apart from $f = 1$), it outperforms MSRP-FT by an average of 110%. Under MSRP-FT, resource access follows a one-serve-at-a-time semantics, where the head-of-queue task occupies the exclusive service slot. If this task suffers frequent faults during its critical section, it can thereby extends the blocking time of subsequent tasks in the FIFO queue. In contrast, LEFT-RS limits the maximum resource access time units of each request to $n_i^x + m + 1$, as illustrated in Corollaries 2 and 4, regardless of the number of faults a request encounters. This prevents excessive faults from degrading system performance.

Table III compares the number of systems that can be scheduled exclusively by either MSRP-FT or LEFT-RS. On the left half of the table, the parameter A is varied, corresponding to the setting in Figure 6e. The first column under this section (labelled "MSRP-FT ✓ / LEFT-RS ✗") shows the number of systems that are schedulable only by MSRP-FT but not by LEFT-RS. The second column (labelled "LEFT-RS ✓ / MSRP-FT ✗") represents the opposite. Similarly, the right half of the table varies the parameter f, as shown in Figure 6f. This table highlights the complementary strengths of the two methods under different parameter configurations.

Observation 4: *LEFT-RS not only outperforms MSRP-FT in terms of overall schedulability, but also in scenarios where MSRP-FT struggles.*

When varying A, as shown in Table III, LEFT-RS consistently demonstrates a one-way dominance over MSRP-FT. Specifically, it can schedule almost all systems deemed schedulable under MSRP-FT, and additionally handle a large number of systems that cannot be scheduled under MSRP-FT. When $A = 25$, there is only one system that is uniquely schedulable by MSRP-FT. This aligns with Corollary 4, which

states that we do not theoretically dominate MSRP-FT in every scenario. In this case, the setting of $f = 3$ (the maximum number of faults) allows tasks generated to encounter between 0 to 3 faults. The rare advantage for MSRP-FT arises when tasks experience few faults, and the associated overhead remains manageable relative to critical section lengths.

A similar observation can be made from the right part of the table when varying the fault tolerance level f. When $f = 1$, MSRP-FT successfully schedules more systems that cannot be scheduled by LEFT-RS. This is expected, as many tasks in these configurations are fault-free or experience only one fault, which limits the fault-tolerance design of LEFT-RS. Even then, only 52 out of 1000 systems are missed by LEFT-RS. When f increases to 2, this number immediately drops to 0. As f increases further, the dominance of the LEFT-RS becomes clear. These findings further reinforce the method's reliability and effectiveness under increasingly fault-tolerant scenarios.

Observation 6: *LEFT-RS maintains a clear advantage over MSRP-FT-OF.*

The comparison with MSRP-FT-OF is not practically meaningful, since MSRP-FT is inherently associated with external overheads absent in LEFT-RS. Nevertheless, the results remain informative. As shown in Figure 6, the overall schedulability trends follow the same pattern as with MSRP-FT: the relative differences between bars are preserved, but the performance gap compared to LEFT-RS is smaller. Although MSRP-FT-OF remains inferior to LEFT-RS in the majority of cases, at lower fault numbers (see Figure 6f) one additional instance ($f = 2$) appears where MSRP-FT-OF achieves slightly higher schedulability than LEFT-RS. This behavior is consistent with Observation 3. Finally, the presentation of MSRP-FT-OF confirms that the advantage of LEFT-RS stems from its protocol design rather than solely from the exclusion of overheads.

VIII. Conclusion

In this paper, we propose a Lock-frEe Fault-Tolerant Resource Sharing (LEFT-RS) protocol for multicore real-time systems. Unlike MSRP-FT, where concurrency only serves the head request and access remains serialized, LEFT-RS enables true concurrency by allowing each request to execute locally after concurrent reading. Tasks can complete earlier upon successful execution, significantly reducing blocking in the presence of frequent transient faults. The independent execution avoids heavy coordination overhead and enables fault resilience. A comprehensive worst-case response time analysis is developed to support timing predictability. Evaluation results demonstrate that our method consistently outperforms state-of-the-art protocols, achieving up to an 84.5% improvement in schedulability on average. Future extensions will move beyond non-preemptive execution and adopt more advanced scheduling strategies, such as ceiling-based protocols.

IX. Acknowledgment

References

[1] B. B. Brandenburg, "Multiprocessor real-time locking protocols," in *Handbook of Real-Time Computing*. Springer, 2022, pp. 347–446.

[2] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 87–98, 2002.

[3] L. Osinski, T. Langer, and J. Mottok, "A survey of fault tolerance approaches on different architecture levels," in *30th International Conference on Architecture of Computing Systems (ARCS)*, 2017, pp. 1–9.

[4] N. Chen, S. Zhao, I. Gray, A. Burns, S. Ji, and W. Chang, "MSRP-FT: Reliable resource sharing on multiprocessor mixed-criticality systems," in *IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 201–213.

[5] S. Zhao, J. Garrido, A. Burns, and A. Wellings, "New schedulability analysis for MrsP," in *IEEE 23rd International Conference on Embedded and Real-time Computing Systems and Applications (RTCSA)*, 2017, pp. 1–10.

[6] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems," in *IEEE Design, Automation and Test in Europe (DATE)*, 2005, pp. 864–869.

[7] N. Miskov-Zivanov, K.-C. Wu, and D. Marculescu, "Process variability-aware transient fault modelling and analysis," in *IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 685–690.

[8] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca, "A comparison of MPCP and MSRP when sharing resources in the janus multiple-processor on a chip platform," in *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2003, pp. 189–198.

[9] A. Burns and A. J. Wellings, "A schedulability compatible multiprocessor resource sharing protocol–MrsP," in *IEEE 25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013, pp. 282–291.

[10] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *13th IEEE International Conference on Embedded and Real-time Computing Systems and Applications (RTCSA)*, 2007, pp. 47–56.

[11] B. B. Brandenburg and J. H. Anderson, "The OMLP family of optimal multiprocessor real-time locking protocols," *Design Automation for Embedded Systems*, vol. 17, no. 2, pp. 277–342, 2013.

[12] S. Zhao, H. Xu, N. Chen, R. Su, and W. Chang, "FRAP: A flexible resource accessing protocol for multiprocessor real-time systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2024, pp. 349–361.

[13] J. H. Anderson, S. Ramamurthy, and K. Jeffay, "Real-time computing with lock-free shared objects," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 2, pp. 134–165, 1997.

[14] S. S. Nabavi and H. Farbeh, "A fault-tolerant resource locking protocol for multiprocessor real-time systems," *Microelectronics Journal*, vol. 137, p. 105809, 2023.

[15] P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *22nd IEEE Real-Time Systems Symposium (RTSS)*, 2001, pp. 73–83.

[16] J. Shi, K.-H. Chen, S. Zhao, W.-H. Huang, J.-J. Chen, and A. Wellings, "Implementation and evaluation of multiprocessor resource synchronization protocol (MrsP) on LITMUSˆRT," in *13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2017.

[17] S. Zhao, J. Garrido, R. Wei, A. Burns, A. Wellings, and J. A. de la Puente, "A complete run-time overhead-aware schedulability analysis for MrsP under nested resources," *Journal of Systems and Software*, vol. 159, p. 110449, 2020.

[18] G. Losa, A. Barbalace, Y. Wen, H.-R. Chuang, B. Ravindran, and M. Sadini, "Transparent fault-tolerance using intra-machine full-software-stack replication on commodity multicore hardware," in *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 1521–1531.

[19] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Systems*, vol. 20, pp. 83–102, 2001.

[20] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.

[21] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[22] B. Ip, "Performance analysis of VxWorks and RTLinux," *Languages of Embedded Systems Department of Computer Science*, 2001.