Article:
Chen, Nan, DAI, XIAOTIAN, Cheng, Tong et al. (2026) CAFT-RS: Fault-Tolerant Resource Sharing Protocols with Diverse Preemption Schemes. IEEE Transactions on Parallel and Distributed Systems. ISSN: 1558-2183

CAFT-RS: Fault-Tolerant Resource Sharing Protocols with Diverse Preemption Schemes

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 increasingly rely on multicore embedded systems, where tasks must coordinate access to shared local and global resources. Such accesses are protected by critical sections and managed by resource-sharing protocols to ensure mutual exclusion and timing predictability. However, transient faults occurring inside critical sections can corrupt execution and propagate errors across tasks, while directly combining conventional locking with fault-tolerance mechanisms can significantly increase blocking. Recent fault-tolerant resource-sharing approaches improve recovery through parallel replica execution, but still suffer from sequential global access and coordination overhead. In previous work, we proposed the Lock-frEe Fault-Tolerant Resource Sharing (LEFT-RS) protocol, which improves fault-tolerant global resource access by allowing concurrent critical-section execution. However, LEFT-RS enforces non-preemptive global resource access, which can cause excessive arrival blocking for high-priority tasks and limit schedulability. This paper introduces the CAFT-RS (Ceiling-based Access for Fault-Tolerant Resource Sharing) protocol, which applies a priority-ceiling mechanism to both local and global resource accesses. CAFT-RS allows higher-priority tasks to preempt ongoing global accesses while preserving correctness through dedicated post-preemption rules. We develop a worst-case response-time analysis that accounts for both the reduction in arrival blocking and the additional preemption overhead. Extensive evaluation results show that CAFT-RS improves schedulability by up to 188.5% on average over LEFT-RS.**

***Index Terms*—real-time systems, resource sharing protocols, transient faults**

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 synchronization flags, whereas global shared resources span multiple cores and include system-wide data structures, shared control variables, configuration parameters, shared counters, and memory-mapped status registers. 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 to ensure system reliability. 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 when there are 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 Sec. III-C.

In our previous work, we proposed Lock-frEe Fault-Tolerant Resource Sharing (LEFT-RS) [5], a lock-free resource-sharing protocol that enables efficient fault-tolerant global resource access by allowing tasks to execute critical sections in parallel and complete earlier via early update when preceding contending requests experience faults. However, LEFT-RS assumes non-preemptive global resource access, which can incur substantial arrival blocking for local high-priority tasks, i.e., a newly released high-priority task may be delayed by an ongoing non-preemptive global critical section executed by a lower-priority task on the same core; this effect can be further amplified under faults due to fault-induced re-executions, and thus limit schedulability.

Building upon LEFT-RS, this paper introduces the CAFT-RS (*Ceiling-based Access for Fault-Tolerant Resource Sharing*) protocol which uniformly applies a priority-ceiling mechanism to both local and global resource accesses. The key novelty of CAFT-RS is to enable preemptive global resource access in a fault-tolerant setting, allowing higher-priority tasks to preempt certain ongoing global accesses while controlling the resulting preemption cost through dedicated post-preemption rules. We further develop a comprehensive worst-

*Corresponding author: Shuai Zhao, Email: zhaosh56@mail.sysu.edu.cn.

case response-time analysis to provide timing guarantees. The main technical contributions of this paper are as follows:

- We propose CAFT-RS, a ceiling-based fault-tolerant resource-sharing protocol that supports preemptive global resource accesses, thereby mitigating the arrival blocking experienced by high-priority tasks in LEFT-RS, where lower-priority tasks execute global resource accesses non-preemptively.
- We develop a new schedulability analysis for CAFT-RS that explicitly accounts for preemptive global resource accesses, characterizing the associated preemption overhead and its trade-off with reduced arrival blocking.
- We extend the evaluation by comparing CAFT-RS with existing approaches in the synthetic schedulability study and by adding automotive-calibrated workloads and an overhead sensitivity analysis, showing that CAFT-RS improves schedulability by up to 188.5% on average over LEFT-RS.

The remainder of this paper is organized as follows. Sec. II presents the system model. Sec. III reviews related work and provides an in-depth analysis of the limitations of the state-of-the-art approach. Sec. IV introduces the LEFT-RS protocol. Sec. V provides a theoretical comparison with existing approaches. Sec. VI presents the worst-case response-time analysis. Sec. VII introduces the protocol design of CAFT-RS and its corresponding schedulability analysis. Sec. VIII reports the experimental evaluation results. Finally, Sec. IX 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, denoted as Γ, scheduled under a fully partitioned fixed-priority preemptive scheduling scheme (FP-FPPS). Under this scheme, each task is statically assigned to one core and does not migrate at runtime; on each core, ready tasks are scheduled according to unique fixed priorities, and a higher-priority task may preempt a lower-priority task except during protocol-defined non-preemptive intervals.

Each core in Λ is represented as λ_k, where k identifies the index of the core. Each task τ_i (the i^{th} 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 x^{th} shared resource is represented as r^x. Each resource r^x is characterized 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 assumes that each resource request is associated with a single resource, so that a task holds at most one resource at a time. Structured multi-resource accesses can still be represented within the model by treating the accessed resource set as a single composite resource, following the idea of group locks [1], [6].

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

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 [7]. The frequency of occurrence of transient faults can be modeled in various ways [8], which is outside 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 each request to r^x is denoted as $f_i^x \leq f_i$. We use n_i^x to denote the total number of executions 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 summarized in Table I.

III. Related Work and Background

A. Resource sharing protocols

In existing multiprocessor resource-sharing protocols, the handling of local resources is largely uniform. Since local resources are accessed only by tasks on the same processor, they reduce to a uniprocessor priority-inversion problem, and are typically regulated using ceiling-based uniprocessor mechanisms such as the Priority Ceiling Protocol (PCP) or the Stack Resource Policy (SRP) [1], [9]. Intuitively, these protocols assign each resource a ceiling priority (the highest priority among tasks that may access it) and enforce execution so as to prevent unbounded priority inversion and ensure deadlock freedom. Under fixed-priority scheduling, PCP and SRP provide equivalent blocking guarantees with bounded blocking and analyzability [10].

Fundamental design differences arise when resources are shared across processors. In this setting, contention spans multiple cores, and existing solutions broadly fall into

two categories, namely suspension-based and spin-based approaches [9]. Suspension-based protocols, such as the Multiprocessor Priority Ceiling Protocol (MPCP) [11] and the Flexible Multiprocessor Locking Protocol (FMLP) [6], suspend a task when a resource request cannot be immediately satisfied, allowing the processor to execute other ready tasks. While this avoids delaying local higher-priority tasks and can be beneficial when critical sections are long, suspensions incur frequent context switches and migrations, introduce nontrivial runtime overhead, and significantly complicate response-time analysis, often leading to pessimistic bounds [12].

In contrast, spin-based protocols, including the Multiprocessor Stack Resource Protocol (MSRP) [11], the $O(m)$ Locking Protocol (OMLP) [13], the Preemptive Waiting Locking Protocol (PWLP) [6], the Multiprocessor Resource Sharing Protocol (MrsP) [10], and more recent designs such as the Flexible Resource Accessing Protocol (FRAP) [14], require tasks to busy-wait while contending for global resources. Although spinning consumes processing capacity, it offers simpler implementation, more predictable execution behavior, and is generally more amenable to static analysis. Consequently, spin-based locking has been widely adopted in practice and remains the dominant approach in industrial real-time systems [15], [16].

Within the spin-based family, protocols further differ in whether spinning is preemptive. Non-preemptive spinning, as exemplified by MSRP, enforces that once a task requests a global resource, it cannot be preempted while either spinning or executing the critical section [11]. This design simplifies blocking analysis and protects the progress of the resource accessing, but it can impose substantial delays on local higher-priority tasks. Preemptive spinning protocols, such as PWLP, relax this restriction by allowing tasks to be preempted while spinning, thereby improving local responsiveness [6]. More recent protocols, such as FRAP, further extend this direction by allowing the spinning priority to vary within a bounded range, rather than enforcing a single fixed rule, enabling finer-grained control over blocking behavior under global contention [14].

Another common characteristic shared by most multiprocessor resource-sharing protocols is that critical sections are executed non-preemptively. This design choice addresses the lock-holder preemption problem. If a task holding a global resource is preempted, all other tasks waiting for that resource may be transitively blocked across processors, making worst-case blocking difficult to bound. Enforcing non-preemptive execution of critical sections eliminates preemption-induced queue re-ordering, thereby yielding more predictable blocking behavior [17]. Nevertheless, a small number of protocols explicitly deviate from this design. For example, Multiprocessor Bandwidth Inheritance (MBWI) does not enforce non-preemptive execution of critical sections; instead, it relies on bandwidth inheritance to limit interference, and thus does not explicitly address the lock-holder preemption issue [18]. In contrast, Spinning Processor Executes for Preempted Processor (SPEPP) [19] and MrsP permit preemption but introduce helping or migration mechanisms to preserve progress [10]. Although these mechanisms mitigate the resulting delays, they introduce additional runtime overhead, increase implementation complexity, and give rise to subtle corner cases.

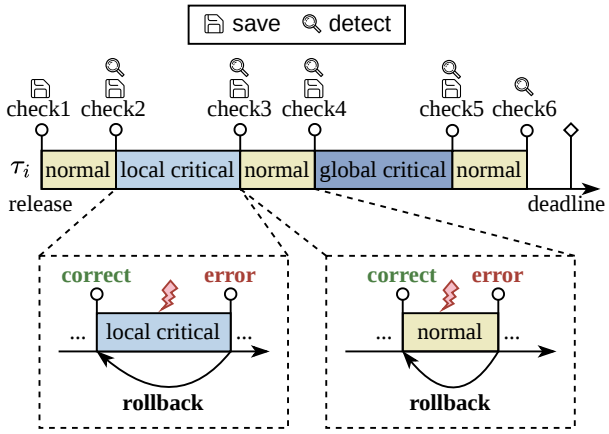


Fig. 1: Fault tolerance of normal and local critical sections.

These tradeoffs explain why most protocols avoid preemptible critical sections and why the few exceptions require carefully engineered mechanisms to remain analyzable.

In contrast to these lock-based approaches, lock-free approaches [20] rely on atomic operations such as Compare-and-Swap (CAS) or Load-Link and Store-Conditional (LL/SC) to update shared resources without explicit locks. Instead of enforcing mutual exclusion through serialized access, these algorithms detect conflicts by monitoring changes to shared data and retrying operations when necessary. This lock-free design provides system-wide progress guarantees while preventing race conditions and data corruption.

B. Fault-aware resource sharing

The work in [21] 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 [22], 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 Fig. 1, MSRP-FT places checkpoints at the beginning and end of each task, as well as around critical sections. These checkpoints divide execution into normal sections without shared-resource access and critical sections with local or global shared-resource access. In such a checkpointing scheme, checkpoints act as fault-tolerance boundaries, where validation, state saving, or both may be performed depending on their positions in the task. Therefore, the operations performed at different checkpoints are not identical. The initial checkpoint (e.g., Check 1 in Fig. 1) only saves the initial task state, such as the task's input values, local variables, and execution context before the first segment starts, as there is no preceding segment to validate. The intermediate checkpoints (e.g., Checks 2–5 in Fig. 1) validate the immediately preceding segment and, if it is correct, save the current task state as a valid recovery point. In contrast, the final checkpoint (e.g., Check 6 in Fig. 1) only validates the final segment before task completion and does not save a new recovery state,

since no subsequent segment needs to recover from that point. This structure detects faults after each key segment, especially critical sections, preventing errors from spreading to other tasks through resource sharing.

If a fault is detected in a normal section, the system rolls back to the most recent valid checkpoint and re-executes only the faulty segment. For local shared resources, MSRP-FT adopts the same ceiling protocol as MSRP: each local resource r^x is assigned a ceiling priority equal to the highest priority of any local task that accesses it, and a task locking the resource temporarily executes at that ceiling priority. As shown in Fig. 1, the fault-tolerance handling for a local critical section follows the same pattern as for a normal section: the system rolls back to the most recent valid checkpoint and re-executes only the affected segment.

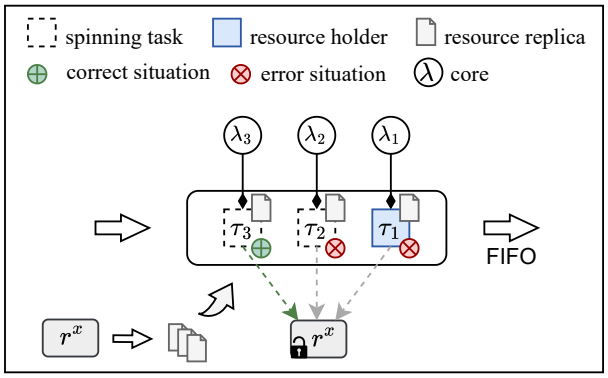


Fig. 2: Fault tolerance of global critical sections.

Additionally, MSRP-FT incorporates a fault-tolerance helping mechanism specifically designed for global resources. As shown in Fig. 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 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 with the following example which indicates the worst-case resource access time of τ_2.

Example 1. *As shown in Fig. 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 completes the 6th overall execution attempt (counting both τ_1 and τ_2) 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 $t = 6$.*

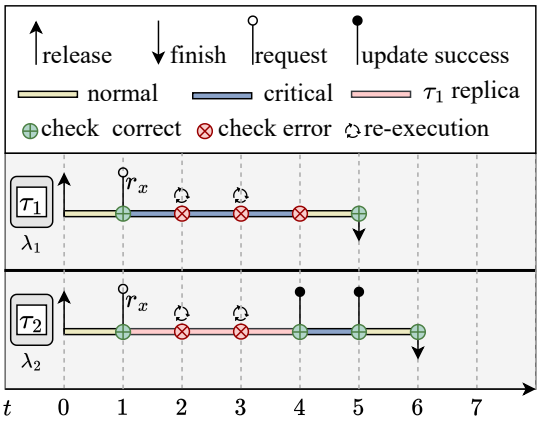


Fig. 3: Worst-case resource accessing of τ_2 under MSRP-FT.

Under MSRP-FT, ignoring coordination overhead, we consider a single access of τ_i to a global resource r^x (e.g., $\tau_i = \tau_2$ in Example 1). The worst-case resource-access time of τ_i then consists of two parts: (i) its own executions and (ii) the remote blocking caused by contending tasks on other cores. In the worst case, τ_i is placed at the end of the FIFO order. Since helpers can only come from requests queued behind it, τ_i receives no external help and must execute its n_i^x replicas sequentially on its own core, contributing $n_i^x \cdot c^x$. The second part arises from contending remote tasks ahead of τ_i in the FIFO order. For clarity, consider one representative remote task τ_j ahead of τ_i (e.g., in Example 1, $\tau_j = \tau_1$ is remote to $\tau_i = \tau_2$). Under MSRP-FT, τ_j's n_j^x executions (each of length c^x) are distributed over k_j participating cores (helping mechanism), which blocks τ_i for $\lceil n_j^x/k_j \rceil$ units of c^x. Combining the above two parts yields the following worst-case access time of τ_i:

$$E_i^x = \left(n_i^x + \left\lceil \frac{n_j^x}{k_j} \right\rceil \right) \cdot c^x \tag{1}$$

Based on this observation, we derive the following corollary, which will be used later.

Corollary 1. *Under MSRP-FT, the remote blocking contributed by a remote request from τ_j is $\lceil n_j^x/k_j \rceil \cdot c^x$. Hence, as fault occurrences increase (i.e., larger n_j^x), the remote blocking experienced by other tasks increases accordingly.*

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, enforcing sequential global resource access (even with helping) can lead to substantial accumulated delays.*

Although MSRP-FT accelerates fault recovery by allowing waiting (spinning) tasks to execute replicas in parallel, it still preserves sequential access semantics: each task must wait for preceding requests in the FIFO order. According to Corollary 1, if a remote request experiences frequent faults (i.e., a large n_j^x), its blocking contribution $\lceil n_j^x/k_j \rceil \cdot c^x$ increases, which in turn delays all tasks behind it in the FIFO queue and amplifies remote blocking.

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 [23], [24]. This structure is written to a globally visible memory location (e.g., DRAM or the last-level cache), and a readiness flag is set via a memory barrier to ensure cross-core visibility [25], contributing to the per-head metadata setup cost 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 [23], [24]. 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:

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

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 present LEFT-RS for multicore real-time systems [5]. 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 rationales.

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 Sec. 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 LEFT-RS manages global resource sharing through a set of protocol rules. The rules are organized according to their design roles rather than strictly following the runtime execution order. To provide an execution-oriented overview before introducing the rules, Fig. 4 illustrates the high-level workflow of a global resource request under LEFT-RS. Example 3 later gives a complete execution sequence involving multiple tasks.

As shown in Fig. 4, a task first joins the FIFO queue and performs initial synchronization when required. It then reads the shared resource state and executes the critical-section logic locally. Two events may force the task to restart its local execution. First, if a preceding request commits an update before the task updates, the task discards its stale local copy and restarts, which is later formalized as a *data-induced re-execution*. Second, if a transient fault is detected during the task's own execution, the task performs a *fault-induced re-execution*. If neither event occurs, the task waits for preceding requests when necessary, commits its update atomically, and leaves the queue.

For clarity, the rules are grouped according to their roles in the protocol. **General execution and re-execution rules (Rules 1–5)** define the lock-free execution model, the FIFO-based update discipline, and the two re-execution causes. **Synchronization and fairness rules (Rules 6–8)** coordinate concurrent executions, updates, and re-executions to preserve FIFO fairness and bound re-execution. **Supplementary execution rules (Rules 9–10)** specify the non-preemptive execution semantics during global resource access and eliminate

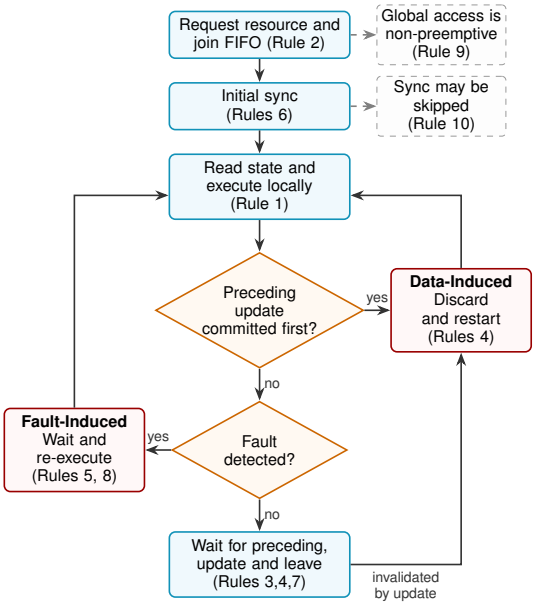


Fig. 4: High-level workflow of Rules 1–10 in LEFT-RS for managing global resource accesses.

unnecessary synchronization when fault-free execution is guaranteed.

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. *Upon accessing a global resource, a task registers its request by enqueueing it into a FIFO queue, which provides a bounded bookkeeping order for coordinating concurrent submissions.*

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 behaviors 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.*

Concurrency issues are addressed, but the current design is not sufficient to provide satisfactory worst-case timing behavior. Since tasks may enter and complete their critical sections at different times due to variations in request arrivals and execution paths, they may finish out of order. Such out-of-order completion can violate the intended FIFO update discipline, leading to unfairness, unpredictable delays, and unnecessary data-induced re-executions. An intuitive example is provided below.

Example 2. *Consider two tasks τ_a and τ_b recorded in FIFO order (front to end) for a global resource r^x. Due to different arrival times and execution paths, their critical-section executions may not overlap perfectly. Suppose τ_b completes its current execution earlier than τ_a. If τ_b were allowed to update r^x immediately, then τ_a, which is still executing based on an older version of r^x, would be forced to re-execute due to this update. Moreover, if additional tasks arrive later and update r^x in quick succession, these later-arriving tasks can repeatedly invalidate τ_a's ongoing execution, continuously deferring its progress and causing starvation and highly unpredictable waiting times, thereby violating the intended FIFO-based update discipline.*

To prevent such out-of-order updates from breaking the intended design, the LEFT-RS introduces a comprehensive synchronization mechanism composed of three tightly integrated rules. First, Rule 6 ensures that tasks joining the FIFO at different times begin execution in aligned windows, reducing initial timing gaps.

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

skipped.

Second, Rule 7 regulates update commitment to preserve FIFO fairness. In particular, early-finishing tasks are prevented from committing updates ahead of preceding tasks, which avoids unregulated data-induced re-executions.

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.*

Since early updates are regulated, early re-execution must be regulated as well. Otherwise, even with Rule 6, an early fault-induced re-execution may reintroduce update-order misalignment and repeatedly defer updates under Rule 7. In Example 2, suppose τ_b completes successfully. Due to Rule 7, τ_b cannot update r^x until the preceding task τ_a finishes its current execution. If τ_a completes early but incurs a fault and immediately starts a re-execution, the waiting condition in Rule 7 may be repeatedly deferred. As a result, τ_b may be forced to wait for successive re-executions of τ_a, leading to starvation and unbounded postponement of τ_b's update. Finally, Rule 8 regulates fault-induced re-executions to avoid serial delays caused by synchronized 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.*

The forced waiting time (Rules 7 and 8) 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. Overall, these design elements work collectively to realize two core rationales that govern the overall protocol behavior and preserve the intended FIFO discipline.

- *Rationale 1*: A task is re-executed due to the update of the global resource only 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 its fault-induced re-executions.

Rationale 1 ensures fairness and bounded re-execution. For any task, the set (and number) of tasks ahead of it in the FIFO queue is fixed once it is enqueued. A task is invalidated only by updates committed by these preceding tasks. Since Rule 4 enforces that each committed update removes exactly one task from the queue, the number of preceding updates is fixed and bounded, making the resulting re-executions 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.

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 behavior is inherited from MSRP [22], chosen for simplicity.

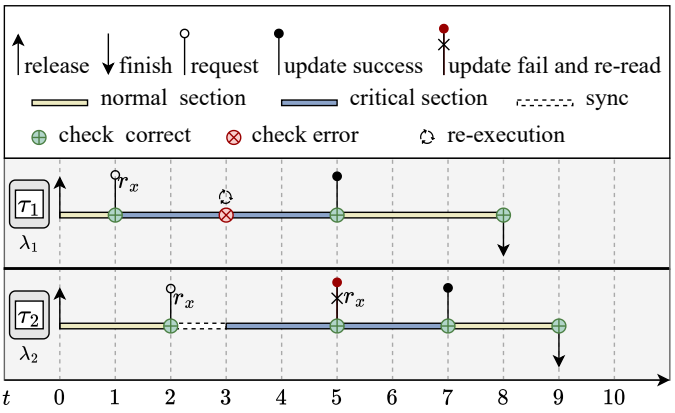


Fig. 5: Example of global resource management of LEFT-RS.

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 synchronization 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 synchronization 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 Rationales 1 and 2. □

These rules are visualized by Example 3 which illustrates the complete execution sequence of a task request for a global resource under LEFT-RS.

Example 3. *As shown in Fig. 5, 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 synchronization 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 executing its critical section at $t = 2$, τ_2 enters a synchronization period. At $t = 3$, as τ_1's execution fails and all executions are synchronized, 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 synchronized, 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 Eq. (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 communication, are shared by both protocols and are discussed in more detail in Sec. VII-C. 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 synchronized 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 Sec. V.

V. Theoretical Comparison with MSRP-FT

The key distinction between the proposed approach and MSRP-FT lies in the management of global resource accesses. Before presenting the full analysis, this section highlights the advantages of the proposed approach by examining the worst-case access time of a single request of a task τ_i to a global resource r^x. In particular, under LEFT-RS, this access time consists of three components: (i) τ_i's own (re-)executions, (ii) remote blocking induced by contending tasks on other cores, and (iii) the synchronization cost imposed by maintaining the FIFO-based submission discipline.

First, 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.

Second, under Rules 4 and 7, remote blocking experienced by τ_i is realized as data-induced re-execution. Specifically, a preceding task in the FIFO queue may commit an update to r^x, thereby invalidating the current execution attempt of τ_i and forcing it to re-execute. Since each committed update removes exactly one task from the FIFO queue (Rule 4), a preceding request can cause at most one such invalidation, even if it experiences multiple fault-induced re-executions before its successful update. Therefore, the total remote blocking can be bounded by counting the number of preceding updates, as formalized by the following lemma.

Lemma 2. *For a given request from a task τ_i to r^x, each simultaneously contending remote request contributes a* constant *remote-blocking delay of at most one unit of c^x, independent of any fault-induced re-executions. Therefore, if at most m ($\leq |\Lambda|$) such remote requests can contend for r^x concurrently, then the request of τ_i incurs a total remote-blocking delay of at most $m \cdot c^x$.*

Proof. A preceding contending task can cause remote blocking to τ_i only by committing an update to r^x, which may invalidate one execution attempt of τ_i and hence induce at most one data-induced re-execution of length c^x. Moreover, each committed update removes one task from the FIFO queue (Rule 4), and thus each preceding remote request can invalidate τ_i at most once. Hence, if at most m remote requests can contend simultaneously, then the request of τ_i incurs at most m instances of data-induced re-execution, i.e., a total remote blocking of at most $m \cdot c^x$. Finally, by Rule 7, later-arriving tasks can trigger such an invalidating update only when τ_i incurs faults, which is already captured by f_i^x failed executions. □

According to Rule 6, each task may enter a synchronization period when requesting a resource. We next bound the delay introduced by this synchronization mechanism. Lemma 3 bounds the one-time synchronization period incurred when a request joins the FIFO queue, while Lemma 4 shows that the waiting enforced by Rules 7 and 8 does not introduce additional delay beyond this bounded synchronization effect.

Lemma 3. *The maximum synchronization overhead for a given resource request is 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. □

Lemma 4. *Apart from the synchronization overhead, no additional waiting time is introduced by Synchronization Rules 6–8.*

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. □

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. The remote blocking incurred is m which 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 $E_i^x = (n_i^x + m) \cdot c^x$.*

Recall that Eq. (1) gives the worst-case access time under MSRP-FT. We next compare the worst-case access time of a single request to r^x from τ_i under MSRP-FT and LEFT-RS. In both methods, τ_i must execute its own request n_i^x times, contributing the same term $n_i^x \cdot c^x$. Hence, the key difference lies in the additional delay induced by *each* contending remote request.

Under MSRP-FT, each contending remote request (e.g., from τ_j) contributes a remote-blocking term of $\lceil n_j^x / k_j \rceil \cdot c^x$, which grows with its fault number (as $n_j^x = f_j^x + 1$). In contrast, under LEFT-RS, each contending remote request can invalidate τ_i at most once and thus contributes exactly one unit of remote delay, i.e., c^x, independent of n_j^x. Beyond these per-contender effects, LEFT-RS incurs an additional one-time synchronization cost of c^x. Therefore, once the fault-amplified remote blocking under MSRP-FT exceeds this one-time cost, LEFT-RS yields a smaller worst-case access time. These observations directly lead to the following corollary.

Corollary 4. *LEFT-RS yields a smaller worst-case access time than MSRP-FT whenever there exists a contending remote request τ_j such that $\lceil n_j^x / k_j \rceil > 2$. More generally, this also holds when the aggregate fault-amplified blocking from multiple remote contenders exceeds the one-time synchronization cost of LEFT-RS. These conditions become more likely under higher fault rates as $n_j^x = f_j^x + 1$ increases.*

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

Example 4. *As shown in Fig. 6, 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 concurrently (Rule 1). They both complete successfully at $t = 2$, with τ_1 updating 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, Corollary 4 and Example 4 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 Eq. (2), has not yet been taken into account. The overall performance comparisons will be demonstrated in Sec. VIII.

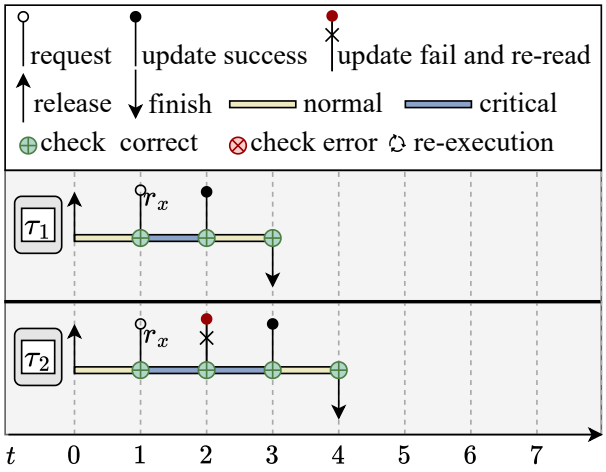


Fig. 6: Worst-case resource accessing of τ_2 with LEFT-RS.

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 Eq. (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 be released and 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.

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

The main challenge in Eq. (3) is to bound the resource-related delay terms under LEFT-RS. In the following, we derive safe upper bounds for E_i, B_i, and F_i in this order. 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 [26].

As shown in Eq. (4), E_i iterates over all shared resources in the system ($\mathcal{R}$). For each resource r^x, the following three components are considered. First, $N_{i,local}^x$ denotes the total number of local requests to r^x, issued by τ_i and tasks in $lhp(i)$. Here, we assume $n_i^x = 1$ for every local request, since the worst-case fault-recovery time is accounted for in F_i and F_h. $N_{i,local}^x$ is further expanded in Eq. (5). Second, $|\mathcal{E}_i^x|$ gives the total number of remote-blocking requests incurred by these local requests, as derived in Eqs. (6), (7), and (8). Third, Syn_i^x captures the synchronization overhead experienced by local requests, as defined in Eq. (9).

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

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
$N^x_{i,local}$	Number of requests issued by τ_i and $lhp(i)$ to r^x
$\eta^x_i(R_i, R_j)$	List of τ_j's requests to r^x over time periods R_i and R_j
ξ^x_{i,λ_k}	Remote requests (n^x_j) that can block τ_i and $lhp(i)$ from λ_k
$\mathcal{N}^x_{i,\lambda_k}$	Maximum number of remote requests blocking τ_i and $lhp(i)$ from λ_k
$\mathcal{E}^x_i$	Total list of n^x_j of remote requests that can block τ_i and $lhp(i)$
Syn^x_i	Synchronization overhead incurred by local requests to r^x
α^x_i	Largest n^x_j from $llp(i)$ for r^x
β^x_i	The list of remote n^x_j that can block α^x_i
$\mathbb{I}(\cdot)$	An indicator function that returns 1 if the condition inside holds true and 0 otherwise
$F^A(\tau_i)$	Resources imposing arrival blocking on τ_i
$F(\tau_i)$	Resources accessed by τ_i

As shown in Eq. (5), $N^x_{i,local}$ includes the total number of requests from τ_i to r^x (N^x_i). 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^x_h of each τ_h.

$$N^x_{i,local} = N^x_i + \sum_{\tau_h \in lhp(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot N^x_h \tag{5}$$

Under LEFT-RS, each simultaneously contending remote request contributes at most one unit of remote blocking, i.e., at most c^x. Therefore, we use $\mathcal{E}^x_i$ to denote the worst-case set of remote requests that can block the request of τ_i to r^x. Accordingly, $|\mathcal{E}^x_i|$ denotes the size of this set. In Eq. (6), $\lambda(\tau_i)$ denotes the core where τ_i is allocated. $\mathcal{E}^x_i$ 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 $\mathcal{N}^x_{i,\lambda_k}$ possible requests from ξ^x_{i,λ_k} which will be explained later.

$$\mathcal{E}^x_i = \bigcup_{\lambda_k \in \Lambda, \lambda_k \neq \lambda(\tau_i)} \bigcup_{p=1}^{\mathcal{N}^x_{i,\lambda_k}} \xi^x_{i,\lambda_k}(p) \tag{6}$$

ξ^x_{i,λ_k} as shown in Eq. (7) represents a non-increasing list of total requests in terms of n^x_j from a given remote core λ_k. It iterates over tasks on λ_k (denoted as $\Gamma(\lambda_k)$) and takes n^x_j of each request to r^x. The notation $\eta^x_j(R_i, R_j)$ denotes the list of n^x_j 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 [26]. For instance, $\eta^x_j(R_i, R_j) = \{2, 2, 2\}$ indicates that there are three requests with $N^x_j = 3$ and all requests have execution numbers $n^x_j = 2$. In general, the execution number can differ across requests, but we use this uniform example for clarity.

$$\xi^x_{i,\lambda_k} = \bigcup_{\tau_j \in \Gamma(\lambda_k), \lambda_k \neq \lambda(\tau_i)} \eta^x_j(R_i, R_j) \tag{7}$$

Since each core can manage at most one active request at a time, for any local request of τ_i, there can be at most one request from λ_k placed ahead of it in the FIFO queue [26]. Therefore, only a bounded number of requests from λ_k can contribute to the worst-case remote blocking to τ_i's local requests. We denote this maximum number by $\mathcal{N}^x_{i,\lambda_k}$, as given in Eq. (8), which equals the minimum of (i) the number of local requests issued by τ_i and $lhp(i)$ to r^x and (ii) the number of remote requests to r^x issued on core λ_k. Since ξ^x_{i,λ_k} is a non-increasing list, selecting the first $\mathcal{N}^x_{i,\lambda_k}$ items yields the largest n^x_j values and thus the worst-case synchronization overhead according to Lemma 1.

$$\mathcal{N}^x_{i,\lambda_k} = \min\{N^x_{i,local}, |\xi^x_{i,\lambda_k}|\} \tag{8}$$

According to Lemma 3, each local request can incur synchronization overhead at most once. Therefore, over all local requests of τ_i to r^x, the total synchronization overhead is upper bounded by $N^x_{i,\text{local}}$. On the other hand, by Lemma 1, a local request incurs no synchronization overhead if all preceding requests have $n^x_j = 1$. Hence, synchronization overhead can be triggered only if there exists at least one preceding request with $n^x_j > 1$. As a result, to reach the upper bound $N^x_{i,\text{local}}$, there must exist at least $N^x_{i,\text{local}}$ such preceding requests in the FIFO order; otherwise, the total overhead is limited by their count. Combining the above observations, the total synchronization overhead equals the smaller of (i) the number of preceding requests with $n^x_j > 1$ and (ii) the per-request upper bound $N^x_{i,\text{local}}$. This is captured in Eq. (9), where $\mathbb{I}(\cdot)$ is an indicator function that returns 1 if the condition holds and 0 otherwise.

$$Syn^x_i = \min\left\{ \sum_{p=1}^{|\mathcal{E}^x_i|} \mathbb{I}\left(\mathcal{E}^x_i(p) > 1\right), N^x_{i,local} \right\} \tag{9}$$

Arrival blocking can occur only once upon the release of τ_i. It is caused by a local lower-priority task $\tau_l \in llp(i)$ accessing either (i) a shared local resource with a ceiling priority higher than that of τ_i or (ii) a global resource executed non-preemptively. The arrival blocking is denoted as B_i, as shown in Eq. (10). The set of resources that can impose arrival blocking on τ_i is denoted as $F^A(\tau_i)$. Accordingly, B_i takes the maximum blocking over all resources in $F^A(\tau_i)$, where each candidate blocking term consists of the same per-request access-time components (i.e., own executions, remote blocking, and synchronisation overhead).

$$B_i = \max_{r^x \in F^A(\tau_i)} \left\{ (\alpha^x_i + |\beta^x_i| + \mathbb{I}(\exists n^x_j \in \beta^x_i,\ n^x_j > 1)) \cdot c^x \right\} \tag{10}$$

The first term α^x_i in Eq. (10) is expanded in Eq. (11), which denotes the largest n^x_l of a request among requests of local low-priority tasks ($\tau_l \in llp(i)$) to a given resource $r^x \in F^A(\tau_i)$ (expressed as $N^x_l > 0$).

$$\alpha^x_i = \max\{n^x_l | \tau_l \in llp(i) \wedge N^x_l > 0\} \tag{11}$$

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 Eq. (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 Eq. (6). This construction ensures that the remote requests counted in β_i^x are not double-counted in E_i.

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

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 synchronization 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 [27]. Therefore, the worst-case fault-tolerance time for τ_i and $lhp(i)$ is denoted in Eq. (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 it 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. CAFT-RS: A Uniform Ceiling Protocol

This section presents CAFT-RS, a ceiling-based protocol that uniformly applies a priority-ceiling mechanism to both local and global resource accesses. We first introduce the protocol design and then present the corresponding schedulability analysis.

A. Protocol design

As discussed in Sec. III-A, multiprocessor resource-sharing protocols differ in whether resource accesses are executed preemptively. Both non-preemptive and preemptive designs have been studied, trading off different forms of blocking and overhead [10], [22].

LEFT-RS assumes non-preemptive execution for global resource accesses according to Rule 9. Unlike local resources, whose contention reduces to a uniprocessor priority-inversion problem handled by PCP/SRP, allowing preemptions during global resource accesses can introduce cross-core transitive blocking and more complex execution interleavings. The non-preemptive assumption therefore simplifies both the protocol semantics and the corresponding schedulability analysis by avoiding such preemption-induced interactions. However, enforcing non-preemptive accesses may incur substantial arrival blocking for high-priority tasks (captured by B_i in Eq. (10)), since a newly released high-priority job (typically with an urgent deadline) may be delayed by an ongoing non-preemptive global access executed by a lower-priority task. This effect can be further amplified under faults due to fault-induced re-executions.

CAFT-RS builds upon the lock-free resource-access structure introduced in LEFT-RS, where tasks read the shared resource, execute locally, and then submit updates without holding a global lock during execution. It preserves the key behavioral guarantees of LEFT-RS, including bounded re-execution and FIFO-based update ordering. To mitigate arrival blocking, CAFT-RS enables preemptive global resource accesses under a uniform priority-ceiling discipline for both local and global resources. Specifically, upon accessing a resource, the executing task's priority is immediately raised to the resource ceiling, i.e., the highest priority among tasks allocated to the same core that may access the resource, which prevents interference from medium-priority tasks and restricts interruptions to only higher-priority tasks. While this capability reduces arrival blocking for high-priority tasks on the same core, introducing preemption in a fault-tolerant resource-sharing setting is non-trivial. In particular, a preempted access must preserve protocol correctness and limit additional overhead, which requires dedicated post-preemption rules and corresponding schedulability analysis. Therefore, CAFT-RS extends the LEFT-RS workflow with a post-preemption path: a preempted task discards its current local execution, temporarily leaves the FIFO queue, re-synchronizes after resumption, and rejoins the queue at its original position. Rules 11–13 formalize these steps.

First, unlike conventional lock-based protocols, by leveraging the lock-free semantics of CAFT-RS (inherited from LEFT-RS), when a task accessing a global resource is preempted, it does not stall other tasks from accessing the same global resource, since the FIFO discipline constrains only the update/progress order rather than enforcing exclusive lock ownership throughout the access. This contrasts with lock-based protocols (e.g., MrsP [10]), where preempting a lock holder would stall all contending tasks unless thread migration or helping is applied. Moreover, a preempted execution has not committed any update to the shared resource and therefore it will not leave partially updated states. Consequently, a preempted task in CAFT-RS can safely discard its ongoing execution, allowing preemption to proceed without thread migration or helping mechanisms. To preserve the FIFO-based execution discipline (Rules 6–8) under preemption, a preempted task is temporarily removed from the FIFO queue so that subsequent tasks are not forced to treat it as a FIFO predecessor during its preemption. It re-joins the queue upon resuming the resource access. We formalize this behavior as the following rule.

Rule 11. *When a uniform ceiling protocol is applied, a task that is preempted during a global resource access immediately discards its ongoing execution and is temporarily removed from the FIFO queue, without affecting the relative order of other tasks.*

Rule 11 ensures that enabling preemption does not delay the progress of other tasks accessing the same resource. The remaining issue is how a preempted task should re-enter the

protocol after preemption. If it were allowed to read the global resource again and immediately restart its execution, the resulting execution may violate the synchronization and submission discipline required by Rules 6–8, leading to excessive re-executions and potentially unbounded remote-blocking effects. To avoid this, a preempted task must first re-align with other active tasks before retrying, as formalized next.

Rule 12. *After being preempted, if there are active tasks in the queue, a task synchronizes with them and waits until the next update of the shared resource before restarting its critical section; otherwise, it restarts its critical section after resuming.*

When a preempted task re-enters the protocol after re-synchronization, the natural and safe behavior is to resume from its original FIFO position. Technically, this can be realized by associating each pending request with a logical FIFO index when it first enters the queue, which remains unchanged across preemption. This choice is safe for other tasks because it preserves the original relative order: requests that were originally behind the preempted task do not face any additional blocking beyond what is already captured by the FIFO-based worst-case accounting. For instance, Lemma 2 bounds remote blocking based on the set of simultaneously contending remote requests and their relative order in the queue. Allowing the preempted task to move behind other requests is also unnecessary. Such a delay does not reduce any other task's worst-case bound, but it can only worsen the preempted task's own outcome by delaying its update eligibility. Therefore, the following rule is proposed.

Rule 13. *When a preempted task resumes its global resource access, it rejoins the FIFO queue at its original position, without affecting the relative order of other requests.*

The discussion above shows that CAFT-RS enforces a *uniform priority-ceiling discipline* for both local and global resources, thereby enabling *preemptive* global resource accesses. In doing so, CAFT-RS preserves the lock-free submission semantics and the FIFO-based update discipline, while upgrading the execution semantics of global accesses from strictly non-preemptive to preemptive under a ceiling protocol. This capability can reduce arrival blocking by allowing higher-priority tasks to preempt ongoing global accesses, which can directly improve schedulability for tasks with tighter deadlines. However, preemption also introduces additional costs due to discarded executions and re-synchronization, as governed by Rules 11 and 12. The next subsection quantifies this trade-off in the schedulability analysis.

B. Corresponding schedulability analysis

We first introduce the notion of ceiling priority, which is the core of ceiling-based protocols. For any resource r^x on core $\lambda(\tau_i)$, its ceiling priority $\Pi^x_{\lambda(\tau_i)}$ is defined as the highest priority among all tasks allocated to that core that may access it:

$$\Pi^x_{\lambda(\tau_i)} = \max\{\, P_q \mid \tau_q \in \Gamma(\lambda(\tau_i)) \wedge N^x_q > 0 \,\} \tag{14}$$

Here, $\Gamma(\lambda(\tau_i))$ denotes the set of tasks allocated to the same core as τ_i, and $N^x_q > 0$ indicates that task τ_q issues at least one request to resource r^x. Intuitively, $\Pi^x_{\lambda(\tau_i)}$ captures the highest priority level that may contend for r^x on that core.

In LEFT-RS, local and global resources are governed by different synchronization disciplines, leading to two sources of arrival blocking. First, under Rule 9, global resource accesses are executed non-preemptively; hence, a task may be released while a lower-priority task on the same core is already executing such an access. Second, local resources (Sec. IV-A) are regulated by a ceiling-based protocol, under which a task may also incur arrival blocking when a lower-priority task accesses a local resource whose ceiling priority exceeds P_i. Accordingly, the set of resources that may impose arrival blocking on τ_i is characterized as follows.

$$F^A(\tau_i) = \left\{ r^x \in \mathcal{R} \;\middle|\; \overset{(\exists \tau_l \in llp(i),\ N^x_l > 0)}{\wedge\ (r^x \text{ is global } \vee\ (r^x \text{ is local } \wedge\ \Pi^x_{\lambda(\tau_i)} > P_i))} \right\} \tag{15}$$

The first conjunct in Eq. (15) checks whether there exists a lower-priority local task $\tau_l \in llp(i)$ that issues at least one request to r^x (i.e., $N^x_l > 0$). The second conjunct then determines whether r^x can cause arrival blocking to τ_i: this holds if r^x is global (and hence non-preemptive under Rule 9), or if r^x is local but has a ceiling priority $\Pi^x_{\lambda(\tau_i)} > P_i$.

When the priority-ceiling protocol is uniformly applied to both local and global resources (as in CAFT-RS), arrival blocking for τ_i can only be caused by resources whose ceiling priorities exceed P_i. The corresponding arrival-blocking resource set is denoted as $F^A_{\text{ceil}}(\tau_i)$.

$$F^A_{\text{ceil}}(\tau_i) = \left\{ r^x \in \mathcal{R} \;\middle|\; (\exists \tau_l \in llp(i),\ N^x_l > 0) \wedge \Pi^x_{\lambda(\tau_i)} > P_i \right\} \tag{16}$$

It follows immediately that $F^A_{\text{ceil}}(\tau_i) \subseteq F^A(\tau_i)$, since global resources with $\Pi^x_{\lambda(\tau_i)} \le P_i$ may cause arrival blocking under the non-preemptive design in LEFT-RS, but are excluded once a uniform ceiling protocol is enforced. Consequently, for task τ_i, the arrival-blocking bound B_i in Eq. (10) remains unchanged in form, but is evaluated over a smaller arrival-blocking resource set (i.e., $F^A_{\text{ceil}}(\tau_i)$). Therefore, B_i is guaranteed to be no larger than that obtained under LEFT-RS and may be significantly smaller.

However, when transitioning to a uniform priority-ceiling protocol, preemptions may occur during global resource accesses. In CAFT-RS, a preempted task does not block other pending requests (Rule 11), but it must discard its ongoing execution and later re-synchronize with the FIFO queue before restarting (Rule 12). This introduces an additional *preemption overhead* that must be accounted for in the schedulability analysis.

Consider a task τ_i accessing a global resource r^x with execution cost c^x. If τ_i is preempted during this access, then (i) the partial execution of the current access is wasted, which in the worst case can be upper bounded by c^x, and (ii) after the preemption, τ_i must re-synchronize with the FIFO discipline before restarting the access, which we conservatively upper bound by an additional cost of c^x. Hence, each preemption

during an access to r^x incurs a preemption overhead of at most $2 \cdot c^x$.

To bound the preemption overhead, we denote by $F^G(\tau_i)$ the set of global resources accessed by τ_i as shown in Eq. (17). Under a ceiling protocol, τ_i can be preempted during an access to a resource $r^x \in F^G(\tau_i)$ only by a local higher-priority task τ_h whose priority exceeds the resource ceiling, i.e., $P_h > \Pi^x_{\lambda(\tau_i)}$ (eligible resources). For a given preempting task τ_h, we consider the worst-case impact on τ_i, which is attained when every such preemption occurs while τ_i is accessing an eligible resource that yields the largest per-preemption overhead. Accordingly, we define $c_i^{\max}(\tau_h)$ in Eq. (18) as the maximum c^x among the eligible global resources, with $c_i^{\max}(\tau_h) = 0$ if no such resource exists.

$$F^G(\tau_i) = \{\, r^x \in \mathcal{R} \mid N_i^x > 0 \ \wedge \ r^x \text{ is global} \,\} \tag{17}$$

$$c_i^{\max}(\tau_h) = \max\Big\{ c^x \ \Big| \ r^x \in F^G(\tau_i) \wedge \ P_h > \Pi^x_{\lambda(\tau_i)} \Big\} \tag{18}$$

Each job of τ_h can preempt τ_i at most once, because once the preemption occurs, τ_i will not resume its global resource access until τ_h completes. Hence, the total number of such preemptions is upper-bounded by the number of job arrivals of τ_h within the response-time window of τ_i, which is given by $\lceil R_i/T_h \rceil$ under the standard response-time analysis. Since each such preemption incurs an additional preemption overhead of at most $2 \cdot c_i^{\max}(\tau_h)$, the total preemption overhead incurred by τ_i is bounded as follows. This overhead should be incorporated into R_i (Eq. (3)) as an additional term for the analysis of CAFT-RS.

$$O_i^{pre} = \sum_{\tau_h \in hp(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot 2 \cdot c_i^{\max}(\tau_h) \tag{19}$$

In summary, applying a priority-ceiling protocol uniformly to both local and global resources can reduce arrival blocking by allowing higher-priority tasks to preempt ongoing global resource accesses when permitted by the ceiling rule. Analytically, this change preserves the overall structure of the response-time bound in Eq. (3), reduces the arrival-blocking term B_i in Eq. (10) by restricting it to the smaller resource set $F^A_{\text{ceil}}(\tau_i)$ (Eq. (16)), and introduces an additional preemption-overhead term O_i^{pre} (Eq. (19)). Intuitively, the reduction in arrival blocking can be substantial, especially when long lower-priority global accesses would otherwise delay newly released higher-priority tasks, whereas the incurred preemption overhead is often limited because preemption can occur only when the preempting task's priority exceeds the resource ceiling. Overall, the relative advantage depends on whether the reduction in B_i outweighs the additional overhead O_i^{pre}. In Sec. VIII, we evaluate LEFT-RS and CAFT-RS under a wide range of system configurations to quantify this trade-off.

C. Applicability and Implementation Considerations

LEFT-RS and CAFT-RS target shared-resource accesses that can be structured as read-compute-atomic-update operations: a task reads a consistent snapshot, executes the critical-section logic on a private copy, and commits the result through a single atomic update after successful validation, consistent with Rules 1, 3, and 4. Under this model, compatible resources include shared data structures, software-maintained status variables, configuration parameters, shared counters, and certain memory-mapped I/O states whose externally visible effects can be delayed until the final commit point. This access model also defines the boundary of the proposed approach: non-idempotent operations, non-rollbackable multi-step state transitions, and direct device interactions that trigger irreversible effects during local execution are outside the assumed model unless they are refactored so that the irreversible action occurs only at commit time.

Eq. (2) captures the overheads introduced by MSRP-FT's helping mechanism. When a request becomes the FIFO head, MSRP-FT constructs and publishes a wrap descriptor; waiting tasks then retrieve the descriptor, prepare local replicas, and cooperatively execute replicas of another task's request. These operations are not required by LEFT-RS or CAFT-RS, where tasks execute only their own critical-section logic and do not construct wrap descriptors or execute replicas of another task's request. The synchronization delay of LEFT-RS, however, is a protocol-induced timing effect and is already included in the response-time analysis in Eq. (9).

Other implementation costs, which may be non-negligible but are not specific to LEFT-RS or CAFT-RS, include checkpoint-based fault detection, FIFO bookkeeping, atomic updates, memory-visibility control, and resource-state preparation. These costs are incurred by the compared protocols in different forms and are therefore not assigned exclusively to any single protocol in the synthetic schedulability evaluation in Secs. VIII-A–VIII-B. LEFT-RS and CAFT-RS may still incur implementation-dependent bookkeeping beyond this analytical timing term, such as stale-copy/version tracking, tentative-state bookkeeping, Rule 7 update-eligibility checks, Rule 8 execution-round tracking, and CAFT-RS post-preemption bookkeeping. These costs mainly involve lightweight metadata checks and per-request state updates rather than executing another task's critical-section logic. Their impact is therefore evaluated separately through the overhead sensitivity analysis in Sec. VIII-C.

VIII. Evaluation

In this section, we evaluate the proposed approach in two parts. First, we use synthetic task sets to systematically vary the main parameters affecting fault-tolerant resource sharing. Second, we use automotive-calibrated task sets derived from the Bosch engine-control benchmark characteristics [28] to conduct two complementary analyses: one assesses schedulability under a more realistic workload profile, and the other examines the impact of implementation-dependent overheads through sensitivity analysis.

Across these evaluations, we compare the schedulability of systems under LEFT-RS, CAFT-RS, Oracle-Best, MSRP-FT, MSRP-FT-OF, and Checkpointing. All methods use the same checkpoint placement strategy for fairness; they differ only in how global resource sharing is handled under faults. LEFT-RS uses non-preemptive global resource access, whereas CAFT-RS uses a ceiling-based preemptive scheme. Oracle-Best reports the better result between LEFT-RS and CAFT-RS for

each task set. MSRP-FT and MSRP-FT-OF use the helping mechanism described in Sec. III-B, with MSRP-FT-OF excluding the protocol overhead in Eq. (2). Checkpointing instead re-executes faulty critical sections sequentially while holding the resource lock, without parallelism or concurrent critical-section execution.

A. Synthetic Experimental Setup

We generated synthetic task sets following the same experimental setup as in our previous work [5]. We consider multicore platforms with $M = [2, 4, 6, 8, 10, 12, 14, 16]$ cores and vary the number of tasks per core as $N = [2, 3, 4, 5, 6, 7, 8, 9]$, providing a broad range of schedulable systems for comparison. Task utilizations U_i are generated using UUnifast [29], with total system utilization $U_{\text{total}} = 0.04 \cdot M \cdot N$. Task periods are drawn from $[1\text{ms}, 1000\text{ms}]$ using a log-uniform distribution, and deadlines are set equal to periods. The total WCET including critical sections is $\widehat{C}_i = U_i \cdot T_i$.

Each system contains $K = M$ shared resources. A fraction of tasks, determined by the resource-sharing factor $rsf \in [0, 0.8]$, is selected to access a random number of resources up to K. The maximum number of accesses to each resource per release is varied as $A = [1, 5, 10, 15, 20, 25, 30, 35]$, and each resource length is randomly generated from $L \in [1\mu s, 300\mu s]$. Let C_i^r denote the total critical-section length of τ_i; we set $C_i = \widehat{C}_i - C_i^r$ and enforce $C_i \geq 0$.

The maximum number of faults per task release is varied as $f \in [0, 7]$, where a task may incur any number of faults from 0 to f in one release. This follows the fault model in [4] for fair comparison and reflects realistic multi-bit or multi-node transient upsets caused by radiation or EMI [30]. Tasks are allocated using Worst-Fit, scheduled by FP-FPPS, and assigned priorities according to Deadline Monotonic ordering.

To parameterize the structural coordination overheads of MSRP-FT (Limitation 2), we use empirical measurements from real-time operating systems on multicore platforms. The wrap construction costs O_{wrap} and $O_{\text{self\_wrap}}$ in Eq. (2) account for publishing a shared descriptor, including function pointers, resource references, synchronization flags, memory visibility, and fencing. Prior LITMUS$^{\text{RT}}$ measurements report wrap setup costs below $1\,\mu s$ [23], while a Linux-based fault-tolerant messaging prototype reports cross-core publication latency of about $0.55\,\mu s$ [25]. We therefore set $O_{\text{wrap}} = O_{\text{self\_wrap}} = 1\,\mu s$.

The per-replica overhead O_{replica} covers helper-side polling, descriptor fetching, and resource-state copying before replica execution. Existing measurements report cooperative execution costs of about $8.4\,\mu s$ under MrsP on LITMUS$^{\text{RT}}$ [24], with related costs such as migration and helper context re-entry measured at $5.6\,\mu s$ and $1.5\,\mu s$, respectively [23]. Comparable RTOS measurements include $11\,\mu s$ on VxWorks and $13.4\,\mu s$ on RTLinux [31]. To avoid overstating MSRP-FT overhead, we use the conservative value $O_{\text{replica}} = 6\,\mu s$.

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 Fig. 7, the y-axis is the rate of schedulable systems over 1000 generated systems (denoted as schedulability). Table III and Fig. 8 summarize, respectively, the number and fraction of systems schedulable exclusively under one approach in pairwise comparisons.

B. Synthetic Workload Results

Observation 1: *As shown in Fig. 7, 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.

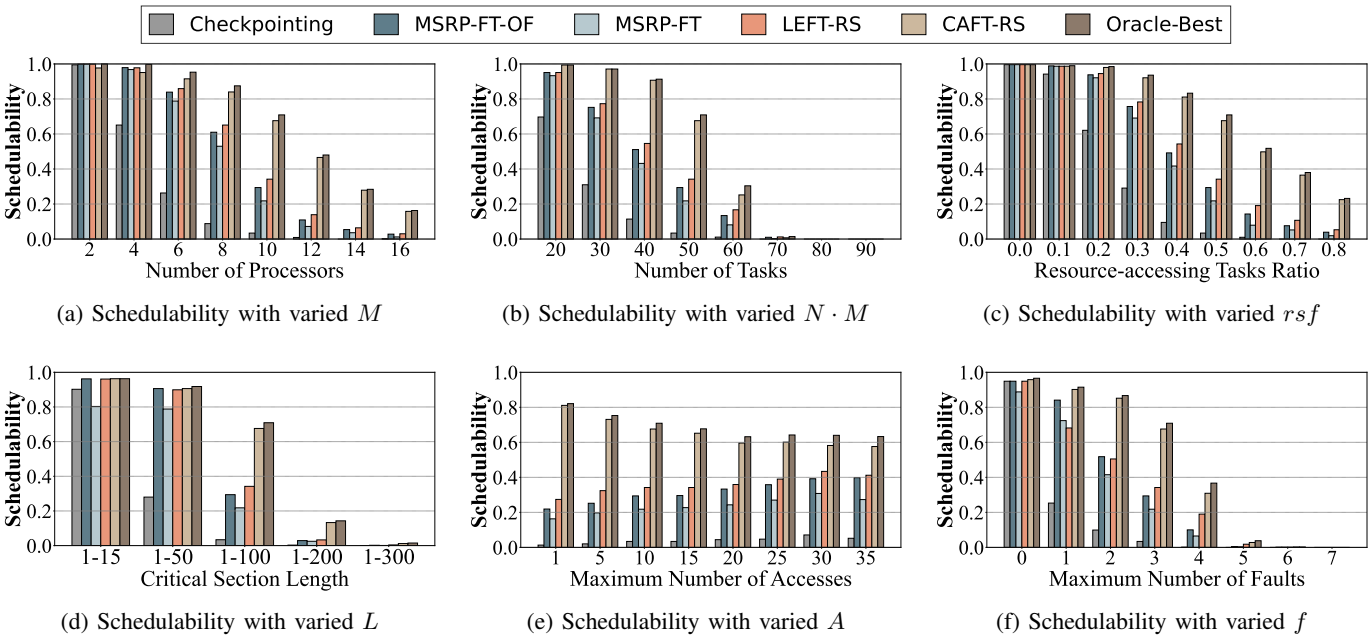
Observation 2: *LEFT-RS consistently outperforms MSRP-FT in Figs. 7a, 7b, 7c, 7d, and 7e.*

Fig. 7a shows that increasing the number of cores from 2 to 16 with $N = 5$ reduces schedulability for all methods due to intensified global-resource contention. However, the gap between LEFT-RS and MSRP-FT widens with the number of cores: LEFT-RS schedules 10 more systems than MSRP-FT at 4 cores, and 71 more systems at 6 cores, showing better scalability under higher contention. Although MSRP-FT accelerates recovery by allowing spinning tasks to execute critical sections in parallel, it still enforces sequential FIFO resource access, so a fault in one task can delay all subsequent requests (Limitation 1); it also incurs coordination overhead that grows with the number of contending requests across cores (Limitation 2, Eq. (2)). In contrast, LEFT-RS executes critical sections independently and enforces FIFO only for update publication, allowing tasks to leave the queue after successful updates and avoiding fault-amplified FIFO delays and MSRP-FT-specific overheads. Consequently, LEFT-RS improves over MSRP-FT by 51.3% on average in Fig. 7a.

The same trend appears in Figs. 7b, 7c, and 7d, where varying the total number of tasks, the proportion of resource-sharing tasks, and the critical-section length increases global-resource contention. LEFT-RS achieves average improvements of 62.9%, 58.8%, and 84.5% over MSRP-FT in Figs. 7b, 7c, and 7d, respectively. In Fig. 7d, when $L = [1\mu s, 15\mu s]$, Checkpointing outperforms MSRP-FT, indicating that for short critical sections, sequential re-execution can be cheaper than helping. Finally, Fig. 7e shows that increasing the maximum number of resource accesses does not necessarily reduce schedulability, because under utilization constraints, more accesses may lead to shorter generated critical sections, as also reported in [4], [14]. LEFT-RS nevertheless remains better than MSRP-FT, with an average improvement of 53%.

Observation 3: *In Fig. 7f, LEFT-RS loses its advantage over MSRP-FT when the number of faults is very small.*

As shown in Fig. 7f, when $f = 0$, Checkpointing is equivalent to traditional MSRP by definition, since no faults are considered. LEFT-RS also performs identically to Checkpointing because its synchronization overhead is avoided according to Rule 10 in Sec. IV, while the additional overheads of MSRP-FT make it worse than the other protocols. When the maximum



(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. 7: System schedulability of the evaluated approaches under varied system parameters. Unless otherwise stated, parameters are fixed at $N = 5$, $M = 10$, $A = 10$, $L \in [1, 100]\mu s$, $rsf = 0.5$, $f = 3$, with $K = M$ shared resources.

TABLE III: Pairwise exclusive schedulability comparison between MSRP-FT and LEFT-RS. Left: varying A (Fig. 7e). Right: varying f (Fig. 7f). Columns report systems schedulable only by MSRP-FT or only by LEFT-RS out of 1000 systems.

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

number of faults per task is randomly set to at most 1, most resource accesses remain fault-free, and LEFT-RS loses some advantage to MSRP-FT by 6.2%. This matches Corollary 4: with rare faults, each remote request holds the resource-holder position only briefly, and the overhead of MSRP-FT remains manageable relative to the critical-section length.

As f increases, however, the benefit of LEFT-RS becomes clear. Excluding the $f = 1$ case, LEFT-RS outperforms MSRP-FT by 110% on average in the scenarios where it has an advantage. Under MSRP-FT, global resource accesses follow one-serve-at-a-time FIFO semantics, and the remote blocking from a remote request τ_j is $\lceil n_j^x / k_j \rceil \cdot c^x$ (Corollary 1), which grows with fault-induced re-executions. In contrast, under LEFT-RS, each simultaneously contending remote request contributes a constant remote-blocking delay of at most one c^x, independent of faults (Lemma 2).

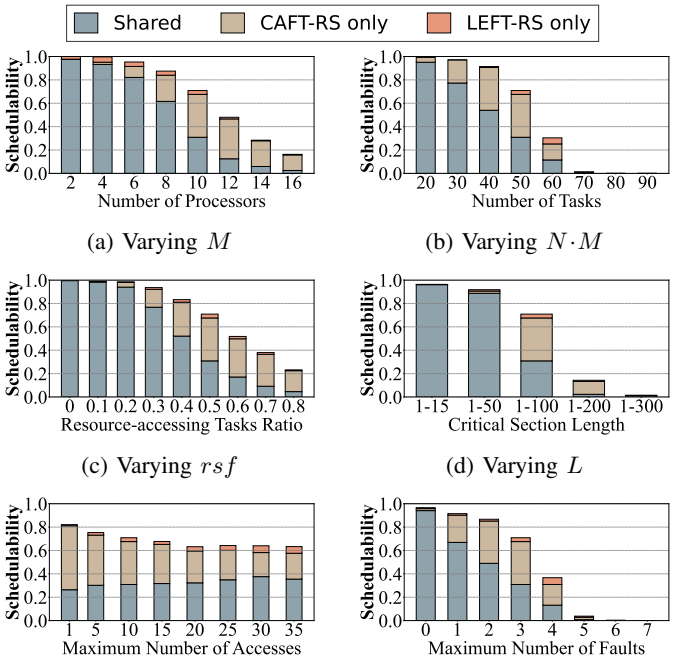
Observation 4: *In Table III, LEFT-RS not only improves overall schedulability over MSRP-FT, but also performs better in scenarios where MSRP-FT struggles.*

When varying A, Table III shows that LEFT-RS exhibits near one-way dominance over MSRP-FT: it schedules almost all systems schedulable by MSRP-FT and many additional systems that MSRP-FT cannot schedule. The only exception occurs at $A = 25$, where one system is uniquely schedulable by MSRP-FT. This is consistent with Corollary 4, since LEFT-RS does not theoretically dominate MSRP-FT in every case. Here, with $f = 3$, generated tasks may experience between 0 and 3 faults; MSRP-FT can rarely be better when tasks incur few faults and its overhead remains manageable relative to the critical-section lengths.

A similar pattern appears when varying the fault-tolerance level f, shown on the right side of the table. At $f = 1$, MSRP-FT schedules more systems that LEFT-RS cannot, because many tasks are fault-free or experience only one fault, which limits the benefit of LEFT-RS. Even in this case, LEFT-RS misses only 52 out of 1000 systems. When $f = 2$, this number immediately drops to 0, and as f increases further, the dominance of LEFT-RS becomes clear. These results further support the reliability and effectiveness of LEFT-RS under increasingly fault-tolerant scenarios.

Observation 5: *LEFT-RS maintains a clear advantage over MSRP-FT-OF as shown in Fig. 7.*

The comparison with MSRP-FT-OF is intended to isolate the impact of MSRP-FT's external overheads, which are absent in LEFT-RS, and should be interpreted accordingly. As shown in Fig. 7, 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 Fig. 7f) 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, presenting



(a) Varying M (b) Varying $N \cdot M$

(c) Varying rsf (d) Varying L

(e) Varying A (f) Varying f

Fig. 8: Comparison between LEFT-RS and CAFT-RS under varied system parameters. Each stacked bar shows the fraction of systems that are shared-schedulable, CAFT-RS-only schedulable, and LEFT-RS-only schedulable. Unless otherwise stated, $N = 5$, $M = 10$, $A = 10$, $L \in [1, 100]\mu s$, $rsf = 0.5$, $f = 3$, and $K = M$.

MSRP-FT-OF confirms that the advantage of LEFT-RS stems from its protocol design rather than solely from the exclusion of overheads.

Observation 6: *CAFT-RS consistently outperforms LEFT-RS across most system configurations in Figs. 7 and 8.*

Fig. 7 shows that CAFT-RS achieves higher schedulability than LEFT-RS in almost all settings, with the largest gains under contention-dominated workloads where non-preemptive global critical sections cause severe arrival blocking for high-priority tasks. The improvement mainly comes from reducing B_i (Eq. (10)) for high-priority tasks, which typically have tighter deadlines and benefit most from avoiding non-preemptive blocking. These tasks rarely incur preemption overhead themselves, since preemption mainly affects lower-priority global accesses; moreover, the ceiling-based preemption overhead is bounded by Eq. (19) and is primarily incurred by lower-priority tasks with less urgent deadlines.

As contention increases, e.g., with more processors in Fig. 7a or a larger resource-sharing factor in Fig. 7c, LEFT-RS suffers from larger arrival-blocking windows and more deadline misses due to non-preemptive global accesses. This effect is further amplified by longer critical sections (Fig. 7d) and higher fault levels (Fig. 7f), where lower-priority tasks may perform more repeated executions and thus prolong non-preemptive blocking. By enabling preemption through a uniform ceiling protocol, CAFT-RS mitigates these blocking effects and consistently improves over LEFT-RS in such blocking-dominated regions. Averaged over the settings where CAFT-RS outperforms LEFT-RS, the schedulability improvements are 188.5%, 99.4%, and 120.3% in Figs. 7a, 7c, and 7d, respectively.

Fig. 8 further confirms this advantage: in most settings, the number of systems schedulable only under CAFT-RS exceeds those schedulable only under LEFT-RS, and the gap grows with contention. For example, in Fig. 8a, when $M = 6$, CAFT-RS schedules 94 additional systems that LEFT-RS cannot schedule, while LEFT-RS schedules only 38 systems that CAFT-RS cannot. When $M = 12$, these numbers become 341 versus 14, demonstrating the substantially larger benefit of CAFT-RS under higher contention.

Observation 7: *LEFT-RS can outperform CAFT-RS in a small number of low-contention configurations.*

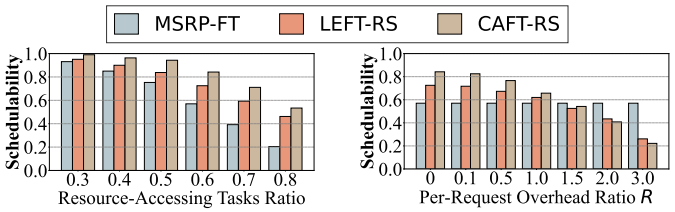
This behavior is most visible in Fig. 7a. When contention is light, i.e., with a small processor count, arrival blocking under non-preemptive global accesses is already limited, so the overheads of ceiling-based preemption may dominate. Specifically, at $M = 2$, LEFT-RS schedules 999 out of 1000 systems, whereas CAFT-RS schedules 977 systems, i.e., 22 fewer. At $M = 4$, LEFT-RS schedules 978 systems compared with 951 under CAFT-RS, i.e., 27 fewer. A similar effect appears in the extreme corner case of Fig. 7b: when $N \cdot M = 70$, LEFT-RS schedules 12 systems while CAFT-RS schedules 7.

Overall, Observations 6 and 7 show that the preferable preemption scheme mainly depends on the contention regime of the target system. In general, CAFT-RS provides better schedulability and becomes increasingly beneficial in contention-dominated configurations, such as systems with more processors or longer critical sections, where reducing arrival blocking yields substantial gains. LEFT-RS is preferable only in a limited set of low-contention settings, where preemption overhead can dominate its benefit. In practice, the choice between LEFT-RS and CAFT-RS can follow whichever achieves better schedulability for the target system, analogous to Oracle-Best.

C. Automotive-Calibrated Workload and Overhead Sensitivity

This subsection complements the synthetic evaluation with two targeted experiments on workload realism and implementation-dependent overhead. We use the real-world automotive benchmark characteristics reported by Kramer et al. [28], extracted from an industrial Bosch engine-control application. Task periods follow the reported runnable-period distribution, and resource-access intensity is calibrated from the reported inter-task communication density over sender–receiver period pairs. This preserves the benchmark's activation and communication structure while instantiating it in the fault-tolerant resource-sharing model considered here. Critical-section lengths, fault parameters, and concrete resource requests are therefore taken from our system model and the main-evaluation defaults.

For the overhead-sensitivity experiment, we use the same automotive-calibrated workload with $rsf = 0.6$. For each global resource execution attempt under LEFT-RS and CAFT-RS, we add $R \cdot c^x$ as implementation overhead, making the



(a) Bosch-calibrated workload. (b) Overhead sensitivity.

Fig. 9: Additional evaluation on the Bosch-calibrated automotive workload. Unless otherwise stated, $M = 8$, $N = 4$, $A = 10$, $c^x \in [1, 100]\,\mu s$, $f = 3$, $K = M$, and 1000 systems are generated per point. (a) varies rsf; (b) fixes $rsf = 0.6$ and varies the LEFT-RS/CAFT-RS overhead ratio R.

effective cost $(1 + R) \cdot c^x$. Thus, R is the added overhead as a percentage of the original critical-section length. We sweep R from 0 to 3.0, i.e., from 0% to 300% additional overhead per attempt. This is a deliberately conservative stress test: the possible additional costs discussed in Sec. VII-C, such as stale-copy/version tracking, tentative-state bookkeeping, update-eligibility checks, execution-round tracking, and post-preemption bookkeeping, are mainly metadata and per-request state updates, yet we conservatively charge them as a percentage of the full critical-section length. MSRP-FT is kept unchanged with the helping overhead in Eq. (2).

Fig. 9a shows that the ordering observed in the synthetic evaluation is preserved under the Bosch-calibrated automotive workload: MSRP-FT $<$ LEFT-RS $<$ CAFT-RS for all tested rsf values. For example, at $rsf = 0.6$, schedulability increases from 0.570 under MSRP-FT to 0.725 under LEFT-RS and 0.842 under CAFT-RS; at $rsf = 0.8$, the corresponding values are 0.205, 0.462, and 0.534. The gap between MSRP-FT and LEFT-RS grows with rsf, which is consistent with the fact that a larger number of global-resource requests amplifies the accumulated sequential blocking in MSRP-FT. Compared with LEFT-RS, CAFT-RS further improves schedulability by reducing non-preemptive arrival blocking through ceiling-based preemption.

Fig. 9b shows that the comparison degrades gradually as additional LEFT-RS/CAFT-RS overhead is introduced. At $R = 0.1$, LEFT-RS and CAFT-RS achieve schedulability values of 0.717 and 0.825, respectively, compared with 0.570 for MSRP-FT; at $R = 0.5$, the values remain 0.673 and 0.766. Even at $R = 1.0$, both remain above MSRP-FT, with schedulability values of 0.620 and 0.657. LEFT-RS crosses MSRP-FT only between $R = 1.0$ and $R = 1.5$, showing that its advantage does not rely solely on being treated as overhead-free. For very large R, CAFT-RS can fall below LEFT-RS because the inflated critical-section cost also amplifies the preemption-related term O_i^{pre}, while the reduction in arrival blocking remains bounded by the task-set structure. This is consistent with Observation 7 and shows that the preferable preemption scheme depends on both contention and implementation overhead.

IX. Conclusion

We previously proposed the Lock-frEe Fault-Tolerant Resource Sharing (LEFT-RS) protocol for multicore real-time systems. Unlike MSRP-FT, where global requests are still served in FIFO order despite parallel replica execution, LEFT-RS allows multiple requests to read the shared resource concurrently and execute their own critical sections locally. This enables earlier completion after successful execution and reduces blocking under frequent transient faults. This paper introduces the CAFT-RS (Ceiling-based Access for Fault-Tolerant Resource Sharing) protocol, which further reduces arrival blocking for high-priority tasks through ceiling-enforced preemptive global resource accesses while preserving the lock-free FIFO-based update discipline. We provide dedicated post-preemption rules and schedulability analysis to preserve correctness and bound the resulting overhead. Evaluations show that CAFT-RS improves schedulability by up to 188.5% on average over LEFT-RS. Future work will further broaden the coverage of CAFT-RS by extending the protocol design and analysis to heterogeneous many-core architectures, more fine-grained resource-access patterns, and richer fault models.

Acknowledgment

This research was funded in part by Innovate UK SCHEME project (10065634). EPSRC Research Data Management: No new primary data was created during this study.

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] N. Chen, X. Dai, T. Cheng, A. Burns, I. Bate, and S. Zhao, "LEFT-RS: A lock-free fault-tolerant resource sharing protocol for multicore real-time systems," in *2025 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2025, pp. 541–552.

[6] 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.

[7] 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.

[8] 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.

[9] S. Zhao, "A FIFO spin-based resource control framework for symmetric multiprocessing," Ph.D. dissertation, University of York, 2018.

[10] 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.

[11] 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.

[12] B. B. Brandenburg, "The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 61–71.

[13] 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.

[14] 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.

[15] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM computing surveys (CSUR)*, vol. 43, no. 4, pp. 1–44, 2011.

[16] S. Afshar, M. Behnam, R. J. Bril, and T. Nolte, "Per processor spin-based protocols for multiprocessor real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 4, no. 2, p. 03, 2017.

[17] S. Nogd, G. Nelissen, M. Nasri, and B. B. Brandenburg, "Response-time analysis for non-preemptive global scheduling with fifo spin locks," in *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2020, pp. 115–127.

[18] D. Faggioli, G. Lipari, and T. Cucinotta, "The multiprocessor bandwidth inheritance protocol," in *2010 22nd Euromicro Conference on Real-Time Systems*. IEEE, 2010, pp. 90–99.

[19] H. Takada and K. Sakamura, "A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels," in *Proc. IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 1997, pp. 134–143.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *6th International workshop on analysis tools and methodologies for embedded and real-time systems (WATERS)*, vol. 130, 2015, p. 43.

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

[30] 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.

[31] B. Ip, "Performance analysis of vxworks and rtlinux," Department of Computer Science, Technical Report, 2001.

Nan Chen is a Research Fellow in the Department of Computer Science at the University of York, UK. He received the Ph.D. degree in Computer Science in 2024 and the B.Eng. degree in Electronic Engineering in 2019, both from the University of York. His research interests include DAG task scheduling, reliable resource sharing, and mixed-criticality systems.

Xiaotian Dai is a Lecturer (Assistant Professor) with the Real-Time and Distributed Systems Group, University of York, UK. He received the Ph.D. degree in 2019 from the University of York (Best Thesis). He also received the M.Sc. degree in Control Systems from the University of Sheffield and the B.Sc. degree in Control Engineering. His research focuses on timing analysis, task scheduling for multicore systems, and assurance for autonomous and cyber-physical systems.

Tong Cheng is a M.Sc. student at Sun Yat-sen University, Guangzhou, China. She received the B.Sc. degree in 2025 from Xidian University. Her research interests include real-time and embedded systems, cache coherence, heterogeneous systems, and computer architecture. Contact: chengt53@mail2.sysu.edu.cn.

Alan Burns (Fellow, IEEE) is a Professor with the Department of Computer Science, University of York, UK. He is a Fellow of the Royal Academy of Engineering and has served as Chair of the IEEE Technical Committee on Real-Time Systems. He has published over 500 papers and 10 books.

Iain Bate is a Professor of Real-Time Systems with the Department of Computer Science, University of York, UK. His research interests include scheduling, timing analysis, and the design and certification of cyber-physical systems. He has chaired three leading international conferences and served on many programme committees. He was Editor-in-Chief of *Microprocessors and Microsystems* and the *Journal of Systems Architecture* for 15 years, and a Visiting Professor at Mälardalen University, Sweden, for five years. He is a member of the IEEE.

Shuai Zhao received the Ph.D. degree in Computer Science from the University of York, UK, in 2018. He is currently an Associate Professor at Sun Yat-sen University, China. His research interests include scheduling algorithms, multiprocessor resource sharing, schedulability analysis, worst-case execution-time analysis, and safety-critical programming languages. He is a member of the IEEE. Contact: zhaosh56@mail.sysu.edu.cn.