A Robust Approach for Ensuring Total Order Execution of Replicated Sporadic Tasks in Fault-Tolerant Multiprocessor Real-Time Systems

AMIN NAGHAVI, Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Esch-sur-Alzette, Luxembourg
NICOLAS NAVET, Faculty of Science, Technology and Medicine (FSTM), University of Luxembourg, Esch-sur-Alzette, Luxembourg

Replication and diversification are commonly used fault-tolerance techniques to mask accidental faults or malicious behavior of compromised nodes in cyber-physical systems. In event-driven systems, executing diversified replicated tasks across multiple nodes can result in their different execution orders. Implementing a total order protocol for job execution across all nodes ensures consistency and facilitates recovery in case of failures. However, achieving total order comes with significant costs due to the high communication and coordination demands among nodes. Existing solutions require coordination either before each job execution or at each job release. Moreover, some total order protocols may lead to unbounded priority inversion on certain nodes in order to maintain a global execution order. Malicious nodes can deliberately exploit these protocols to launch priority inversion attacks, thereby jeopardizing the timeliness of tasks on healthy nodes in time-critical applications. We propose a total order execution protocol that guarantees bounds on the priority inversion tasks experience and ensures that tasks meet their deadlines in real-time systems. Our approach withstands priority inversion attacks and leverages common knowledge among nodes rather than relying on communication, allowing them to progress independently while still ensuring a consistent execution order of job replicas across nodes upon their release. Although inter-node communication is not required, the method can benefit from exchanged progress data to reduce job response times. It is compatible with coarsely synchronized clocks and, unlike other total order approaches, which are for non-preemptive scheduling, uses progress milestones to enable task preemption. We evaluate our method against existing approaches based on acceptance ratio and response times, and study how job response times vary with increasing communication delays when the approach is used.

CCS Concepts: • **Computer systems organization** → **Real-time systems**; **Dependable and fault-tolerant systems and networks**;

Additional Key Words and Phrases: Real-Time Systems, Multiprocessor Systems, Fault Tolerance, Replication, Total Order

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant reference C19/IS/13691843/ByzRT. For the purpose of open access, and in fulfilment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.
Authors' Contact Information: Amin Naghavi (corresponding author), Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Esch-sur-Alzette, Luxembourg; e-mail: amin.naghavi@uni.lu; Nicolas Navet, Faculty of Science, Technology and Medicine (FSTM), University of Luxembourg, Esch-sur-Alzette, Luxembourg; e-mail: nicolas.navet@uni.lu.

ACM 2378-9638/2025/11-ART39
https://doi.org/10.1145/3765620

ACM Reference format:
Amin Naghavi and Nicolas Navet. 2025. A Robust Approach for Ensuring Total Order Execution of Replicated Sporadic Tasks in Fault-Tolerant Multiprocessor Real-Time Systems. *ACM Trans. Cyber-Phys. Syst.* 9, 4, Article 39 (November 2025), 36 pages.
https://doi.org/10.1145/3765620

1 Introduction

Real-time systems are frequently exposed to harsh environments where they have to operate safely through accidentally induced faults. Furthermore, increasingly, application domains, such as industrial automation [29, 30, 41], telecommunication [3], the power grid [11, 39], and cooperative driving [2, 8], face cyberattacks aimed at jeopardizing system safety. One way to operate safely through such node failures is active replication (i.e., the execution of the same tasks across multiple nodes). Active replication masks the arbitrary behavior of up to f faulty nodes behind a majority of at least $f + 1$ healthy nodes. It has also been shown effective against targeted attacks [7, 12, 47] where adversaries have managed to compromise nodes, even at RTOS level.[1] To ensure that replicas do not share the same vulnerabilities, diversification techniques can be employed, such as n-version programming [9, 51], diverse compiling [28], and the use of different operating systems [19].

In the event-triggered real-time systems we focus on in this article, jobs of tasks are released in response to their respective events. Subsequent releases of the jobs of a task τ_i occur only after the minimum inter-arrival time T_i. When a task is released, its processing can take up to its **Worst-Case Execution Time (WCET)** C_i, and it must be completed within its deadline D_i; otherwise, it might lead to system failure. In systems employing replication, diversified replicas of the same jobs are executed across all nodes. Variation in the time taken to complete these replicas results in different execution orders on nodes, as nodes execute released jobs only after those that have already been executed. In certain applications, it is required to execute replicated jobs in the same order across all nodes. Total order in execution might be required in systems with data dependencies [24, 26] to guarantee comparable results on nodes. Additionally, ensuring a consistent execution order facilitates checkpointing in distributed systems to recover from faults or inconsistencies [15].

Rodrigues et al. [53] and Wang et al. [58] address the problem of queuing prioritized messages to ensure total order in their processing. We consider these messages as events and their processing as the execution of non-preemptive jobs released by these events. Implementing the methods proposed by Rodrigues et al. [53] and Wang et al. [58] in real-time systems may lead to deadline misses due to unbounded priority inversions. Moreover, these methods cannot guarantee the timeliness of tasks in the presence of *priority inversion injection attacks* [43]. Naghavi and Navet [43] addressed the problem of unbounded priority inversion when using total order protocols in real-time systems by introducing a method we refer to as **Limited Priority Inversion Total Order based on Front-Runner Progress (LPI-FRP)**. However, all these approaches rely on communication and become ineffective in meeting timing requirements when there are high communication delays or when occasionally messages are not delivered.

Another approach to guarantee total order is to leverage the time characteristics of tasks known across nodes, rather than relying on communicated progress. For example, a straightforward approach, referred to as the *Simple* method by Naghavi and Navet [43], waits until the WCET of each job has elapsed before executing subsequent jobs. Therefore, when a high-priority job is released, it can be inserted at the earliest position in the queue based on its priority, following the

[1]Vulnerabilities have been reported in Zephyr (CVE-2021-3625, -3835, -3861), Azure RTOS (CVE-2022-41051, -39343, -39344), FreeRTOS (CVE-2021-42553), and Tizen RT (CVE-2021-22684).

completion of the non-preemptive execution of the currently running job (if any). However, this approach requires precise synchronization and cancels out all the benefits of completing jobs early. Therefore, we propose our method that allows nodes to progress independently without waiting for their peers.

Our method is not dependent on communication. Similar to the Simple method, it also utilizes the time characteristics of tasks to determine the position of released tasks in the queue (i.e., their insertion point). This insertion point is calculated based on the slack time and the earliest arrival time of unreleased jobs, as well as the WCET and release time of jobs already in the queue. By ensuring that no job is executed beyond these insertion points—enforcing idle time if necessary—our approach also eliminates the need for rollbacks.

Although our method ensures that nodes can independently determine insertion points and schedule jobs without requiring communication, integrating a non-blocking communication mechanism can reduce job response times. Upon the release of a job, each node broadcasts its execution progress using a real-time reliable broadcast protocol such as the ones proposed in [13, 20]. If all nodes receive progress updates from every other node before a predefined timeout, they can use this shared data to execute more released jobs without needing to wait for other nodes to catch up.

Since non-preemptive scheduling often reduces the number of schedulable task sets, we propose a mechanism that makes our method compatible with limited preemptive scheduling as well. We consider progress milestones [10] within each task, ensuring that when a milestone is completed on one node, equivalent progress can be achieved on other nodes. We assume that tasks can only be preempted at these milestone boundaries.

We evaluate our approach using acceptance ratio and response time, comparing it against the methods proposed by Rodrigues et al. [53], Wang et al. [58], the LPI-FRP approach by Naghavi and Navet [43], and the Simple method. Our approach significantly outperforms Rodrigues and Wang in terms of acceptance ratio and, unlike LPI-FRP, remains effective under high communication delays. Furthermore, by allowing independent progress, our method improves response times compared to the Simple method, even under fairly high communication delays.

The rest of the article is organized as follows. After discussing the background and related work in Section 2 and examining the shortcomings of other total order protocols for prioritized tasks in Section 3, we introduce the system and threat model in Section 4. In Section 5, we propose our method, present the detailed algorithms in Section 6, and discuss their compatibility with coarsely synchronized clocks in Section 7. We evaluate our method in Section 8 and conclude the article in Section 9.

2 Background and Related Work

Accidental Faults. Accidental faults can occur during runtime due to non-malicious causes (e.g., alpha and gamma particles, wear-out, design flaws). These faults can temporarily cause incorrect or missing outputs or messages (transient faults) or result in a crash failure, where a node stops functioning and communicating with other nodes (permanent faults) [52]. Fault-tolerant mechanisms, such as replication and checkpointing, prevent faults from causing system failures. In active replication with diversification, different versions of task replicas are executed on separate nodes to mitigate common vulnerabilities and design faults. In such cases, to mask f faults, at least $2f + 1$ replicas must be executed across nodes, enabling fault masking through majority voting. In checkpointing, the system state is saved at specific points, allowing it to roll back to the last stored checkpoint and re-execute tasks upon fault detection [18].

Some works have focused on tolerating permanent [44] and transient accidental faults [1, 33, 46], including a survey by Reghenzani et al. [52] that reviews software-implemented hardware fault-tolerance methods.

Byzantine Faults and Time-Domain Attacks. Due to malicious targeted cyberattacks or software errors, faulty nodes may exhibit Byzantine behavior, where they act unpredictably or arbitrarily. This means they may send conflicting information to different parts of the system, provide incorrect data, or fail to respond entirely [35]. In time-domain attacks on real-time systems, an adversary may attempt to disrupt the timely execution of tasks or exploit the system's time predictability to steal information [43, 45].

To address such threats, securing real-time systems against cyberattacks has become a critical focus. For instance, Hasan et al. [27] investigate the allocation of security tasks. Nasri et al. [45] analyze conditions for successful time-domain attacks. Li et al. [40] extend the crash-fault tolerant coordination service Zookeeper to include real-time recovery. Yoon et al. [62], and Krüger et al. [37] investigate schedule randomization in event- and time-triggered systems, respectively, including for replicated tasks [38]. Zhang et al. [64] investigate recovery from sensor attacks.

Consistency and Total Order. Nodes may execute task replicas in different orders than other nodes due to varying processing speeds or actual execution times of diversified task replicas. This can lead to inconsistent states among nodes [53]. It may also cause divergent outputs when tasks have cause-effect relationships or data dependencies [32], as task replicas may receive inputs from different instances of their preceding tasks.

Several approaches have been proposed to address this issue by managing message delivery, coordinating voting, or synchronizing node states using datastores. Poledna et al. [50] ensure that all replicas deliver the same messages in the same order by introducing timed messages, using which tasks accept messages from other tasks only after their deadline or their worst-case response time. Additionally, Fara et al. [17] coordinate distributed voting on nodes either through passive waiting until the voting data is received or via Logical Execution Time. Gujarati et al. [23] enable interactive consistency in Byzantine fault-tolerant key-value datastores for reliable real-time control applications with task dependencies. They also address replica coordination over switched Ethernet under non-malicious Byzantine errors in [22], by dividing processes into tasks and executing them after periodic runs of the protocol proposed by Pease et al. [48].

Total order execution protocols ensure consistency in replica outputs by enforcing the same execution order across nodes. Nodes also maintain uniform state transitions, which simplifies the consistent rollback of task executions in case of faults [18]. Coordinating the execution order of replicas has been studied for replicated state machines [55] and primary/backup systems [67]. Rodrigues et al. [53] and Wang et al. [58] extend total order execution to prioritized tasks.

Priority Inversion Injection. Priority inversion occurs when a high-priority task is delayed because of a lower-priority task [14]. In real-time systems, this typically happens due to the non-preemptive execution of a lower-priority task [61]. Wang et al. [58] identify group priority inversion in systems using total order protocols, such as those proposed by Rodrigues et al. [53], where high-priority tasks may be delayed to enforce the same execution order across nodes. When priority inversion on a node is due to enforcing the execution order on another node, we refer to it as *injected priority inversion*.

Wang et al. [58] address the group priority inversion problem by enforcing the execution order of replicas completed by at least $f + 1$ nodes in a system where up to f nodes may crash, while rolling back the execution of replicas that have been completed only by some nodes. Unfortunately, Rodrigues' and Wang's methods are not designed for hard real-time systems, as they may fail to guarantee task timeliness on healthy nodes due to priority inversion injected by the total order protocol. When exploited by a malicious node to induce deadline misses, this is called a *priority inversion injection attack*, a time-domain attack introduced by Naghavi and Navet [43]. Naghavi and Navet [43] also propose a solution to ensure task timeliness while maintaining total order by bounding the priority inversion experienced by tasks according to their available slacks. However, their method requires highly reliable real-time communication that must never fail to deliver messages on time, with delays shorter than the WCET of the smallest task. Additionally, it is applicable only to non-preemptive task models.

The approach presented in this article ensures the total execution order of replicated prioritized tasks across all nodes when inserting them into the nodes' queues, while allowing each node to progress independently by executing jobs from its queue. To improve job response times, we use non-blocking communication and make no assumptions about the underlying network, except that it must support real-time reliable broadcast.

Reliable Broadcast. A reliable broadcast protocol ensures all correct nodes eventually deliver the same messages sent by a correct node [7]. A real-time reliable broadcast adds timeliness, delivering messages only if all nodes received them before a timeout [34, 35]. We use this protocol for communication while accounting for potential message omissions.

Reliable and atomic broadcasts have been widely studied [7, 12, 57, 66], with atomic broadcast ensuring both reliability and agreement on the order of message delivery. Some works focus on bounding delivery delays in atomic broadcast [13, 31]. Kozhaya et al. propose a real-time Byzantine reliable broadcast using time-triggered [34] and event-triggered protocols for atomic broadcast in arbitrary networks [35]. Roth and Haeberlen [54] discuss optimizations for systems with broadcast channels like Bus and Ethernet, leveraging the inherent reliability properties these channels provide.

Preemption and Milestone. In real-time systems, a schedule can be non-preemptive, meaning a task utilizes the processor until completion once it starts execution, or preemptive, where the scheduler can interrupt a lower-priority task to execute a higher-priority one. In the limited preemption model, high-priority tasks can only preempt lower-priority tasks at specific points, called preemption points.

Yao et al. [60] and Bertogna et al. [5] calculate the maximum blocking allowed before executing a task (i.e., slack) to facilitate schedulability tests for various preemption models and to place preemption points in tasks. Phan et al. [49] incorporate task release overhead in schedulability analysis. Task slack is used in other state-of-the-art works to schedule aperiodic tasks, handle self-suspension, and tolerate overrun in mixed-criticality systems [25]. We leverage task slack to ensure total order in execution.

Milestones can be placed within a task during the offline phase to track execution progress at runtime. While Kritikakou et al. [36] and Sinha et al. [56] insert milestones using a compiler or source-to-source translation, Chen et al. embed them directly into the task binary. These milestones can be used to detect timing violations or switch between modes in mixed-criticality systems. In this article, we enable limited preemption by leveraging milestones as preemption points, marking common states in replicated tasks. Our method is also compatible with non-preemptive task models, considering only a single milestone at the end of each task.

3 Total Execution Order of Prioritized Tasks

In event-triggered systems with prioritized tasks, variations in task execution times can result in different execution orders. This is shown in Figure 1(a), which illustrates three nodes—P_1, P_2, and P_3—and five tasks, τ_a (highest priority) through τ_e (lowest priority). In this example, consider two healthy nodes, P_2 and P_3. When the nodes insert the released task τ_a into their queues, P_2 has already executed τ_b, τ_c, and τ_d, whereas P_3 has executed only τ_b. In this case, P_2 inserts τ_a before the lower-priority task τ_e, while P_3 inserts τ_a before τ_c in its queue. Although the task deadlines are guaranteed in this case, the replicas of the tasks will be executed in a different order on different nodes. Therefore, for example, if there is an implicit data dependency between the tasks τ_a and τ_c, then the task τ_c on P_2 and P_3 will have different outputs. In the following, we discuss existing total order protocols, which are dependent on the communication, and explain how priority inversion resulting from these protocols can cause deadline misses in real-time systems.

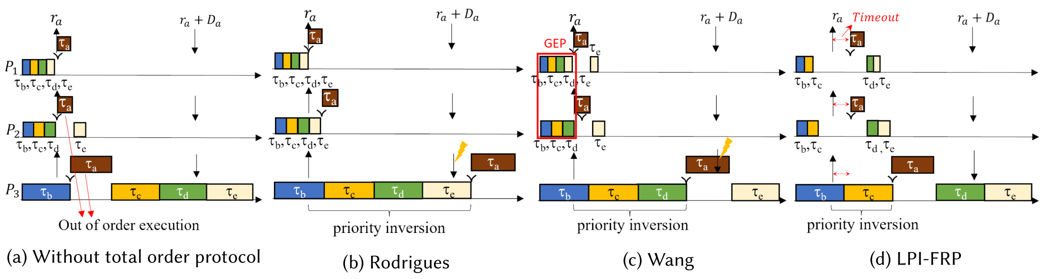


Fig. 1. Different execution orders of replicas without a total order execution protocol (left), deadline misses under the Rodrigues et al. [53] and Wang et al. [58] protocols (middle-left and middle-right), and communication dependency in LPI-FRP by Naghavi and Navet [43] (right).

3.1 Unbounded Priority Inversion

Rodrigues et al. [53] address the problem of guaranteeing total order in scenarios where nodes may fail silently. To achieve total order, nodes communicate upon the arrival of each event by sending the set of jobs they have already executed. The nodes then insert the released job into their queues after all jobs that have been executed by at least one node—that is, after the jobs in the union of all received sets. Using this method to ensure total order execution can lead to unbounded priority inversion, potentially causing missed deadlines in real-time systems. Moreover, in the presence of Byzantine faults, an attacker could exploit the protocol to jeopardize the timeliness of jobs on healthy nodes.

Figure 1(b) shows how a node that has completed more of the released jobs than other nodes can inject significant priority inversion on other nodes, resulting in deadline misses when Rodrigues' total order protocol is used. In this figure, the node P_1 has executed all the released jobs τ_b, τ_c, τ_d, and τ_e at the release of τ_a. At the time of the release, the node P_2 just finished the execution of τ_d while P_3 was executing τ_b. Hence, all nodes insert the job τ_a (released at r_a) after τ_e, which is already executed by P_1. Although P_2 met the deadline for τ_a after catching up with P_1's reported progress, P_3 experienced execution times close to task WCETs, requiring $C_b + C_c + C_d + C_e$ to finish the jobs in its ready queue up to τ_e before executing τ_a. Consequently, P_3 missed τ_a's deadline at $r_a + D_a$. A similar situation can occur if P_1 is malicious and performs a priority inversion injection attack by falsely claiming to have executed enough jobs to cause other nodes to miss deadlines of their tasks.

Wang et al. [58] prevent group priority inversion in the total order protocol of Rodrigues et al. [53] in fault-tolerant systems where f nodes are susceptible to crash failures. They use **Local Execution Progress (LEP)**, representing the number of completed jobs per node, and **Group Execution Progress (GEP)**, which is the minimum LEP among the $f + 1$ nodes with the highest progress. This ensures that at least one of the $f + 1$ nodes, which have executed jobs up to the GEP, will not crash. When an event occurs, its released job, e.g., τ_a, is placed in the queue after the GEP, with its priority determining its execution order among the rest of the jobs in the queue. Nodes whose LEP has exceeded the insertion point of τ_a must roll back to the previous consistent state and execute τ_a first. This requires undoing the effects of jobs executed after τ_a in the queue and re-executing them once τ_a has been completed.

Wang's method ensures that at least one healthy node avoids priority inversion, even if up to f nodes crash. However, in event-triggered real-time systems, other healthy nodes may still face unbounded priority inversion, leading to deadline misses. This occurs when $f + 1$ nodes execute jobs earlier and advance the GEP beyond the progress of other healthy nodes, forcing them to catch up before scheduling jobs based on their priorities.

Figure 1(c) illustrates a scenario where τ_a misses its deadline on the healthy node P_3 when using Wang's method. The progress of nodes P_1 and P_2 causes the GEP to advance to the progress of P_2, which has already completed the execution of τ_d. Consequently, the back-runner node P_3 inserts the released job τ_a after τ_d, but before the lower-priority job τ_e. The time required for P_3 to execute τ_b, τ_c, and τ_d to catch up with the GEP ultimately results in τ_a missing its deadline.

Wang et al. claim to mitigate group priority inversion, even with malicious faults, by requiring additional valid responses and a voting mechanism. Figure 1(c) illustrates that Wang's method requires more than $2f + 1$ nodes to effectively tolerate f Byzantine faults. In the scenario shown, if P_1 is faulty and produces an incorrect output, node P_3 is unable to complete τ_a before its deadline. Consequently, even though both P_2 and P_3 are healthy and eventually produce the correct output, the voter cannot determine the correct result in a timely manner.

Additionally, although in the example in Figure 1(c), adding another healthy node might provide the voter with enough timely responses from τ_a, P_3 would still miss the deadline of τ_a. Therefore, in the case of malicious faults, regardless of the number of nodes, f malicious nodes can exploit Wang's protocol to launch a priority inversion injection attack targeting healthy nodes. Malicious nodes can take advantage of the progress of healthy nodes that executed jobs earlier to advance the GEP, inducing sufficient priority inversion on other healthy nodes to cause jobs to miss their deadlines.

3.2 Limitations of Communication-Based Methods

The LPI-FRP method proposed by Naghavi and Navet [43] guarantees deadlines even in the case of priority inversion injection attacks. Nodes enforce a scheduling protocol to ensure that any priority inversion injected through their progress on other nodes is bounded by the slack of tasks. Nodes communicate at each release to determine the insertion point based on the progress of the front-runner (the node that completed jobs earlier than others) that has not been faulty, meaning it respected priority inversion bounds. Similar to Rodrigues and Wang, this method heavily relies on communication, as it requires knowledge of other nodes' progress to determine a task's insertion point in the queue. This communication also requires locking the ready queue and delaying job insertion until a *Timeout* which bounds the communication delay as shown in Figure 1(d). In the figure, the insertion of τ_a is delayed on P_1 and P_2 until r_a + *Timeout*, leaving these nodes idle. Furthermore, to guarantee total order and meet task deadlines, communication must neither fail nor exceed the *Timeout*, otherwise the algorithm cannot decide about the insertion point of the job. In addition, the *Timeout* must not exceed the WCET of the smallest task (i.e., *Timeout* < $\min_{i \in \{a,b,c,d,e\}} C_i$). If the *Timeout* is not bounded by the WCET of the tasks, communication may not occur in parallel with task execution on the back-runner, and task deadlines cannot be guaranteed. These limitations restrict the applicability and robustness of this method.

The method proposed in this article ensures deadlines of tasks on all healthy nodes, even in the presence of priority inversion injection attacks, and inserts jobs into nodes' queues without relying on communication. It allows the communication timeout to be arbitrarily large and tolerates missed or delayed messages that arrive after the timeout.

4 System, Task, and Fault Model

System and Task Model. We consider event-triggered systems, where tasks are sporadic. We characterize sporadic tasks $\tau_i \in \mathbb{T}$ of the task set $\mathbb{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ through a minimal inter-release time T_i, a relative deadline $D_i \le T_i$, and a WCET C_i. Each task τ_i generates a sequence of jobs where the jth job of task τ_i, denoted by $\tau_{i,j}$, has a release time $r_{i,j}$. We may omit the job index j when it is clear from the context.

Tasks (and the RTOS) are replicated over $m \ge 2f + 1$ nodes $P_1, P_2, \ldots, P_m$ to tolerate up to f faults (or malicious attacks). We assume each node executes all tasks in $\mathbb{T}$. Nodes may correspond to separate processors in multiprocessor systems or to individual cores in multicore systems. To reduce the likelihood of common-mode faults and vulnerabilities, we assume that different nodes execute distinct versions of replicas of the same tasks [51], potentially using different algorithms, data structures, and timing characteristics. To compare the progress of these variants, we use milestones within a task [10] to mark global points of progress across all variants. We assume limited preemption where tasks are preemptive at least at these milestones. We denote L_i as the maximum number of milestones that need to be completed from any variant of the task to finish its execution. We call each portion of task τ_i between two milestones as a *chunk*, denoted by τ_i^l, with WCET C_i^l, defined as the maximum time any variant may take to progress from milestone $l-1$ to l. We consider the WCET of a task to be the sum of the WCETs of its chunks, expressed as $C_i = \sum_{l=1}^{L_i} C_i^l$, which represents an upper bound on the WCET across all its variants.

We do not impose any assumptions on the control flow of tasks. Instead, we treat a chunk as an indicator of progress in the execution of a task. Additionally, we allow for milestones to be skipped, meaning chunks may complete with zero execution time. For example, the code block within the body of an "if" statement can be considered one or more chunks, which may be skipped if the condition is not met. Furthermore, a chunk τ_i^l may represent the set of instructions within both the "if" block and its corresponding "else" block. In such cases, C_i^l is determined by the maximum WCET of executing either set of instructions. In a loop, each iteration can be treated as a chunk (or multiple chunks). If the loop terminates early, some milestones may be skipped. For example, when searching in an array, each comparison can be considered a milestone. Here, L_i equals the length of the array. In the worst case, a variant may find the element in the last cell, requiring C_i execution time. Conversely, another variant may find the element earlier, bypassing the remaining milestones and completing their respective chunks without consuming any time.

We first assume a fixed priority scheduling where hp_i and lp_i are the sets containing all the higher and lower-priority tasks than τ_i, respectively. Later, we show that our method also works well with dynamic priority scheduling. The $slack_i$, which can be computed in the design time, represents the maximum blocking time that any job of task τ_i (when released) can tolerate due to the execution of a lower-priority job while still meeting its deadline.

Execution Model. In our method, we assume that each node P_k initially inserts released jobs into its local priority-sorted queue, called the *Ready Queue* (RQ_k). The job with the lowest priority in RQ_k is always placed at the tail of the queue. A node P_k moves the chunks (according to their execution order) from the job at the head of the ready queue to the tail of another queue, called the *Chunk Queue* (CQ_k). A job is removed from RQ_k once all its chunks have been moved to CQ_k. Once chunks are placed in the chunk queue, their execution order is finalized and will not change. Each entity in the ready queue, denoted by $\tau_x^{next_x,-}$, represents a job τ_x and the smallest ID of the chunks of τ_x that have not yet been moved to the chunk queue (i.e., $next_x$). If $\tau_a^{next_a,-}$ is at the head of the ready queue, the next chunk to be moved from the ready queue to the chunk queue is $\tau_a^{next_a}$.

For simplicity of the presentation, we assume that once chunks are moved to the chunk queue, they remain there and are not removed.[2] Consequently, the position of each chunk in the chunk queue (starting from the head) is referred to as a *progress*, representing the total number of chunks in the chunk queue immediately after the chunk is inserted. At any time t, we denote the progress of the tail of the CQ_k of a node P_k as $prog_k^{tail}$ and the current progress of the node as $prog_k^{cur}$. The current progress of a node ($prog_k^{cur}$) is the progress of the most recent chunk in the chunk queue that the node started executing. A node executes a chunk only if it is in the chunk queue (i.e., $prog_k^{cur} \leq prog_k^{tail}$).

[2]As detailed in Appendix A, chunks need not remain in the chunk queue indefinitely and are removed after execution in practice.

We refer to tasks whose jobs are not in the ready queue as *imminent*. These include tasks that have already been released but are not yet inserted into the ready queue, as well as those that may be released in the future. At time t, assuming r_i^{last} is the release time of τ_i's last released job, the earliest possible release time of the next job of an imminent task τ_i is $\rho_i(t) = \max(r_i^{last} + T_i, t)$. During runtime, each node tracks imminent tasks using a set IS_k.

Synchronization Model. We assume a synchronous system model in which at any instance of actual time, the local time of different nodes may differ by at most δ. This means that if one node reads time t, another node might read time $t \pm x$, where $x \leq \delta$. Events are delivered to all nodes within a maximum delay of Δ after generation, where we assume $\delta \leq \Delta$. However, for simplicity of explanation, we first describe the algorithm under the assumption of synchronized clocks, where nodes receive events simultaneously. In Section 7, we demonstrate that by leveraging timestamped events, with some minor adjustments, our method performs effectively even with coarsely synchronized clocks.

Communication Model. In multicore systems, nodes can communicate in negligible time using shared memory. In multiprocessor systems, however, where communication delays are larger, we assume that communication occurs in parallel with task execution using co-processors or DMA-based **Network Interface Controllers (NICs)** [6] integrated into the processing units. Nodes are connected through a network that may experience failures or excessive delays. However, a real-time reliable broadcast protocol [34, 35] guarantees that messages are either delivered unchanged to all healthy nodes within a maximum time $Timeout$ or not delivered to any nodes at all. $Timeout$ consists of two delay parameters: C_{msg} and C_{send}. The value of C_{msg}, which is the maximum expected network delay, can be determined at design time based on a conservative estimate. C_{send} represents the estimated maximum delay for a node to detect the release of any task and dispatch its progress information. Our method can be implemented in two ways: upon receiving a release event, (1) nodes immediately send their progress to other nodes (as assumed in [43]), or (2) nodes send progress only after completing the currently executing chunk. In the second case, C_{send} must exceed the WCET of each chunk, as progress transmission is delayed until the current chunk is completed.

Threat Model. We allow up to f nodes to become faulty in a Byzantine manner, where the node behaves arbitrarily. This includes intentionally malicious attacks such as cyberattacks. We make no assumptions about the behavior of faulty nodes and consider the possibility that the RTOS may be compromised. A malicious node can execute jobs out of order or may not execute them at all. It can also report false progress data to other nodes or withhold communication about its progress. However, we assume that a malicious node cannot interfere with healthy nodes in any other way. In multicore systems, it is necessary to implement appropriate isolation techniques to prevent faulty cores from directly interfering with healthy ones. This includes memory isolation techniques such as a bank-aware memory allocator [63] or cache partitioning for shared caches [42], which help mitigate interference from compromised cores.

5 Coordinating Replicated Execution in Event-Driven Systems

In this section, we present our method for maintaining total order in task execution across all nodes, which bounds priority inversion and is resilient to priority inversion injection attacks. Our method dynamically coordinates the execution order of jobs without requiring communication. However, communication is used solely to improve job response times. The non-blocking communication does not delay job execution, and late or lost messages are tolerated. This method satisfies the following properties: *Uniform Agreement*: If a replica of a job is completed by a node P_x, its other replicas will eventually be completed by all correct nodes P_y. *No Re-Execution*: Each replica of a job executes exactly once. *Total Order*: For any two chunks τ_i^j and τ_k^l, if τ_i^j is executed before τ_k^l on one node, then τ_k^j will be executed before τ_i^l on every other node. *Limited Priority Inversion*: The delay a job experiences before its execution due to lower-priority jobs is bounded. *Timeliness*: All replicas of all jobs on healthy nodes complete their execution before their designated deadlines.

5.1 Overview of Method Components and Their Interactions

The proposed method in this article operates within predefined bounds on priority inversion to ensure timeliness and maintain total order. These bounds correspond to task slacks, and the method ensures that the priority inversion each task experiences never exceeds its slack. This method consists of two main components: the *Scheduling* and *Release* algorithms, which execute on every node.

Before executing a chunk, the scheduling algorithm on a node P_k ensures that finalizing its execution order across nodes does not jeopardize the deadlines of tasks on any node. In other words, if every node processes the chunk in the order that P_k intends to execute it, it is guaranteed that tasks will not experience priority inversion beyond their slacks. If this condition is not met, node P_k remains idle. When node P_k determines that the execution order of a chunk can be finalized, it moves the chunk from its respective job in the ready queue RQ_k to the chunk queue CQ_k and executes it.

The release algorithm of node P_k inserts a released job into the ready queue, ensuring the execution order of all chunks potentially executed by other nodes is finalized before insertion. This assumes each healthy node executed the scheduling algorithm before processing each chunk. Consequently, as long as the priority inversion imposed on each task remains within its slack, the algorithm finalizes the execution order of chunks in RQ_k by moving them to CQ_k. Thus, it ensures the ready queue contains only jobs that have not yet been executed by any node, before inserting the released job in RQ_k. Since slack bounds are known by all nodes, they derive the same ready and chunk queues at each release, ensuring the released job is inserted in the same position. Additionally, by meeting slack bounds, our method ensures deadlines.

While the release and scheduling algorithms are interdependent, they operate without requiring communication between nodes. Each node accounts for two extreme scenarios: one for the back-runner and another for the front-runner. It assumes that a back-runner may take up to the jobs' WCETs to complete them, whereas a front-runner finishes jobs as early as possible while respecting the bounds on the priority inversion that it is allowed to inject on the back-runner. However, our method can integrate a third algorithm, *Update*, which leverages communication to improve job response times. At each release, nodes share their progress information with others, and the update algorithm, executed upon receiving this communication, updates each node's knowledge of the back-runner's progress. This helps reduce the idle time imposed by the scheduling algorithm.

In the following, we define key terms used throughout the article and provide a more detailed explanation of the Release, Scheduling, and Update algorithms.

5.2 Terms and Definitions

Worst-Case Projection (WCP). Let *minProg* represent the most recent update of the back-runner's progress available to nodes, and let t_{update} denote the time at which this update occurred (serving as an upper bound on when the back-runner may begin executing the chunk corresponding to progress *minProg* + 1). Assuming the back-runner always takes up to the WCET of chunks to complete them and has not been idle since t_{update}, the following provides a pessimistic estimate of when the back-runner will complete the chunk at the progress *prog* (where $prog > minProg$):

$$\omega(prog) = t_{update} + \sum_{\tau_a^l \in CQ_k[minProg+1:prog]} C_a^l \quad . \tag{1}$$

We call this the ***Worst-Case Projection (WCP)*** of progress *prog*. The WCP of $prog = minProg$ is set to t_{update}. Each node updates the variables *minProg* and t_{update} either upon inferring that the back-runner has been idle (without communication) or upon receiving a progress update from the healthy back-runner through communication.

Maximum Allowed Progress (MAP). We define a progress threshold at any time t, called the *MAP*, which the $prog_k^{tail}$ of any healthy node P_k must never exceed. If $prog_k^{tail}$ never exceeds the MAP, it is guaranteed that even the back-runner, by executing chunks up to $prog_k^{tail}$ before executing imminent tasks, will not cause these tasks to experience priority inversion beyond their slacks.

Chunks can be moved from the ready queue RQ_k to the chunk queue CQ_k without $prog_k^{tail}$ exceeding the MAP, as long as the following condition holds for every imminent task $\tau_i \in IS_k \cap hp_a$:

$$\omega(prog_k^{tail}) + C_a^{next_a} \leq \rho_i(t) + slack_i. \tag{2}$$

Here, $\tau_a^{next_a}$ represents the next chunk of the job at the head of RQ_k that will be moved to the chunk queue.

The MAP can be determined by moving chunks from the ready queue to the chunk queue. The Map is equal to $prog_k^{tail}$ when the ready queue is empty or when Equation (2) no longer holds for at least one imminent task.

Lemma 1. *Equation (2) guarantees that an imminent task τ_i meets its deadline even on the back-runner if all nodes execute all chunks in CQ_k as well as $\tau_a^{next_a}$ before executing τ_i.*

Proof. In the worst case, an imminent task τ_i may be released exactly at its earliest release time $\rho_i(t)$. In such a scenario, it must not be delayed by lower-priority jobs beyond its slack. This implies that if a lower-priority job is executing, it must be completed before $\rho_i(t) + \text{slack}_i$ to guarantee the deadline of τ_i. Since $prog_k^{tail}$ is the last progress in CQ_k, and based on the definition of WCP, $\omega(prog_k^{tail})$ represents the latest time at which the back-runner finishes all chunks in the chunk queue. By adding $\tau_a^{next_a}$ to the chunk queue, the latest time this chunk will complete on the back-runner is $\omega(prog_k^{tail}) + C_a^{next_a}$, which may delay the execution of τ_i until that point. However, if this completion time is less than $\rho_i(t) + slack_i$, the delay caused by executing $\tau_a^{next_a}$ will not cause τ_i to miss its deadline on the back-runner. Thus, moving $\tau_a^{\text{next}_a}$ to CQ_k does not cause τ_i to miss its deadline, if the inequality in Equation (2) holds. □

Illustrative Example. Figure 2 illustrates how our scheduling, release, and update algorithms operate. The example considers a scenario with two healthy nodes, P_1 and P_2, and limited preemptive scheduling, where we assume that chunks cannot be preempted by either other tasks or our algorithms. Task indices are sorted based on their priorities, from the highest priority task (τ_a) to the lowest priority task (τ_f). In this figure, the jobs of tasks τ_c, τ_d, τ_e, and τ_f are released at r_c. At r_c, it is assumed that both P_1 and P_2 have finished their last job, τ_g, and are idle at r_c, where the release algorithm sets *minProg* to the progress of the last chunk of τ_g and t_{update} to the release time r_c, as we will explain in Section 5.4. We presented the chunk queue and the ready queue for node P_1 at times t_1, t_3, and t_5, and for node P_2 at times t_2, t_4, and t_6. To demonstrate the WCP of each progress after *minProg*, the gray box in each illustration depicts the WCET of chunks in the chunk queue

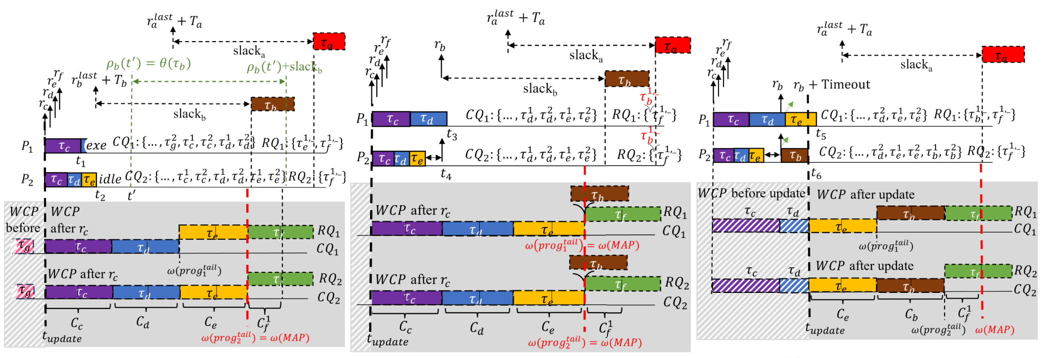

(a) Scheduling algorithm (b) Release algorithm (c) Update algorithm

Fig. 2. Example of the interaction between our scheduling (left), release (middle), and update (right) algorithms on two nodes.

and the ready queue, arranged in their execution order after t_{update}. In this illustration, the end of the WCET block of each chunk in the chunk queue represents the WCP of its respective progress.

We also showed the imminent tasks τ_a and τ_b in Figure 2(a) and (b), while in Figure 2(c), only τ_a is imminent. The earliest release times of the imminent tasks τ_a and τ_b are $r_a^{last} + T_a$ and $r_b^{last} + T_b$, respectively, while the subsequent releases of other tasks are significantly later and not shown. A red dashed line in each figure represents the WCP of the MAP. As illustrated, the MAP marks the progress point at which Equation (2) no longer holds for the next chunk in the ready queue, as adding its WCET to the WCP of the MAP would exceed the slack of higher-priority imminent tasks.

5.3 Scheduling: Executing Chunks Under Priority Inversion Bounds

The scheduling algorithm determines whether a node should execute the next chunk or remain idle. This decision ensures that all nodes can agree on the insertion point for a released task during the next release. In this subsection, we explain how the scheduling algorithm manages chunk execution and determines the required idle time.

The main role of the scheduling algorithm is to ensure that the progress of a node P_k and $prog_k^{tail}$ remain less than or equal to the MAP. This ensures that if a release occurs at any time, all nodes construct the same ready queue by finding the MAP and inserting the task into their ready queues in the same order. The scheduling algorithm executes chunks only when they are in the chunk queue, and before moving any chunk from the ready queue to the chunk queue to execute, it verifies that Equation (2) holds for all imminent tasks (therefore, for the resulting chunk queue, $Prog_k^{tail} \leq MAP$). If no chunk is available in the chunk queue for execution and Equation (2) is not satisfied for an imminent task, the node remains idle to keep its progress within the MAP limit.

Finding the Required Idle Time. During the idle period dictated by the scheduling algorithm, if no imminent tasks are released, the scheduling algorithm runs again at time t' where the Equation (2) holds for all imminent tasks with higher priority than the first job in the ready queue (i.e., $prog_k^{tail} < MAP$). To find t' for a node P_k that completed the final chunk in its chunk queue, the algorithm first calculates $\theta(\tau_i)$ for each imminent task τ_i for which the Equation (2) does not hold. Here, $\theta(\tau_i)$ represents the earliest time such that if $t = \theta(\tau_i)$, Equation (2) is satisfied. Therefore, τ_i's deadline is guaranteed if it is released at $\theta(\tau_i)$ and executed after τ_a^{next}, even when executed on the back-runner. According to Equation (2), $\theta(\tau_i)$ can be calculated as follows:

$$\theta(\tau_i) = \omega(prog_k^{tail}) + C_a^{next_a} - slack_i. \tag{3}$$

To guarantee that a node never exceeds the MAP, t' is determined as the maximum $\theta(\tau_i)$ among all imminent tasks τ_i, for which inequality (2) does not hold. At t', it is permissible to move a chunk from the ready queue to the chunk queue.

Illustrative Example. Figure 2(a) illustrates how our scheduling algorithm operates at time t_1 on P_1 and at time t_2 on P_2. At t_1, after completing the execution of τ_c, node P_1 runs the scheduling algorithm. Since Equation (2) holds for all imminent tasks with higher priority than τ_d, chunks of the job τ_d were moved to the chunk queue, and the execution of τ_d is started. The node P_2 remained idle at t_2 upon completing job τ_e because $prog_2^{tail}$ had already reached the MAP, and moving the first chunk of τ_f to the chunk queue was not allowed. This restriction was enforced due to the fact that the addition of the WCET of τ_f^1 to $\omega(prog_2^{tail})$ exceeds the earliest release time ($\rho_b(t_2)$) plus the slack ($slack_b$) of the higher-priority imminent task τ_b. Consequently, the algorithm calculated $\theta(\tau_b)$, and since τ_b is the only higher-priority task for which Equation (2) does not hold at t_2, the node must remain idle until $t' = \theta(\tau_b)$, waiting for the release of τ_b.

5.4 Release: Coordinating Job Insertion with Limited Priority Inversion

The release algorithm of each node determines the execution order of jobs upon their release while ensuring that all nodes insert them at the same position in their ready queues. Additionally, it updates *minProg* and t_{update} once all nodes have been confirmed to be idle after completing all released jobs.

At each task release, the release algorithm on each node ensures a consistent ready queue across healthy nodes, regardless of their progress, before inserting the released task. Since nodes are unaware of each other's progress, they assume that a front-runner node may have completed jobs up to the MAP. The MAP represents the maximum progress a healthy node can achieve, as the scheduling algorithm ensures the progress of healthy nodes remains at or below the MAP. By moving every chunk up to the MAP to the chunk queue based on Equation (2), all nodes can construct the same ready queue. Therefore, once the MAP is found, they insert the newly released job into the ready queue based on its priority, ensuring consistent job order across all nodes.

Determining WCP Parameters. The WCP of a progress is calculated based on the assumption that a back-runner executing chunks with their WCETs has never been idle since t_{update}. However, if no tasks are released for some time, even such a node would complete all released jobs and remain idle, at which point the values of *minProg* and t_{update} must be updated. The release algorithm can confirm whether the back-runner has been idle without communication.

Lemma 2. *If at any time t, on a healthy node P_k, RQ_k is empty and $\omega(prog_k^{tail}) \leq t$, then it is ensured that the back-runner has completed the execution of all released jobs and is idle, even if it executed chunks with their WCETs.*

Proof. $\omega(prog_k^{\text{tail}})$ represents an upper bound on when the back-runner would finish progress $prog_k^{\text{tail}}$. If RQ_k is empty, this implies that all chunks of the released jobs have been moved to the chunk queue. Therefore, $\omega(prog_k^{\text{tail}})$ provides an upper bound on when the back-runner completes all released jobs. Consequently, the condition $\omega(prog_k^{\text{tail}}) \leq t$ ensures that the back-runner has executed all released jobs by time t and has been idle. □

When a task τ_e is released as r_e, if the conditions in Lemma 2 hold for $t = r_e$, the release algorithm updates *minProg* to $prog_k^{tail}$ and t_{update} to r_e, as the back-runner starts executing the next chunk (i.e., the chunk at *minProg* + 1) at r_e.

Illustrative Example. Figure 2(b) depicts the release algorithm. In this example, task τ_b is released at r_b, when P_1 was executing τ_d and P_2 was idle due to the scheduling decision at t_2. At t_3, after running the release algorithm, node P_1 transferred τ_e to the chunk queue to align $prog_1^{tail}$ with the MAP, followed by the insertion of τ_b into the ready queue. Node P_2, at $t_4 = r_b$, inserts the released job into the ready queue since $prog_2^{tail}$ already aligns with the MAP, and the node is idle. As shown, both nodes P_1 and P_2 had identical ready queues after P_1 transferred chunks from RQ_1 to CQ_1. Thus, they inserted the released task τ_b into the ready queue in the same position, ahead of the lower-priority job τ_f.

In Figure 2(a), we observe how the release algorithm updates $minProg$ and t_{update} upon the release of τ_c. At time r_c, all chunks of the released jobs before τ_c would have completed execution, even if they had taken their WCETs (as shown in the hashed gray area). Thus, the release algorithm on both nodes P_1 and P_2 updates $minProg$ to the progress corresponding to the completion of the last chunk of τ_g. It also sets t_{update} to the release time r_c, ensuring that t_{update} indicates when the chunk at $minProg + 1$, which is τ_c^1, starts executing on the back-runner. These values are then used by both the release and scheduling algorithms to calculate the WCP for each progress, as shown in Figure 2(a) and (b).

5.5 Update: Reducing Pessimism via Communication

During runtime, nodes often complete chunks earlier than their WCETs, making the MAP calculation based on the WCP overly pessimistic. While our method inserts and executes released jobs without communication, using communication with an update algorithm reduces WCP pessimism and improves job response times.

By leveraging communication, nodes can track the healthy back-runner's actual progress to update $minProg$ and t_{update}. At each release of a task τ_e at r_e, nodes initiate a non-blocking communication, allowing the update algorithm to use the received information at the timeout $r_e + Timeout$. Depending on the communication delay, nodes can use the received progress from the healthy back-runner to estimate a less pessimistic WCP for each progress. In such cases, the update algorithm sets $minProg$ to the progress of the healthy back-runner and t_{update} to $r_e + Timeout$, confirming that the healthy back-runner has reached the reported progress by the timeout. This reduces the WCP for each progress beyond $minProg$, increasing the MAP and enabling the scheduling algorithm to execute more chunks before forcing a node to idle.

To guarantee consistency, nodes use a real-time reliable broadcast protocol ensuring either all nodes receive a message sent after the release time r_e before $r_e + Timeout$, or no correct replica delivers it. A node must receive progress information from all other nodes before the timeout to execute the update algorithm; otherwise, no healthy node will execute it. Additionally, nodes run the update algorithm according to $r_e + Timeout$ to ensure all nodes follow the same execution order for the release and update algorithms, maintaining consistency in $minProg$ and t_{update} at all times.

Detecting Faulty Back-Runners. Faulty nodes may attempt to exploit the update algorithm to establish incorrect WCP parameters, disrupting the correct operation of the scheduling and release algorithms. To prevent this, the update algorithm uses progress information to detect and dismiss faulty back-runners, as described in the following lemma:

Lemma 3. *Assume that $prog_k^{sent}$ represents the progress reported by a healthy node P_k to other nodes at the release of a task τ_e. Node P_k identifies another node, P_b, as a faulty back-runner if the progress reported by P_b, $prog_b^{sent}$, is smaller than $prog_k^{sent}$ and the WCP of P_b's progress is less than r_e (i.e., $\omega(prog_b^{sent}) < r_e$).*

Proof. The WCP represents the time when the back-runner completes a progress if chunks execute up to their WCETs. At time r_e, if $\omega(prog_b^{sent}) < r_e$, the node has either idled (Lemma 2) or exceeded the chunks' WCETs. A healthy node idles only if it is about to exceed the MAP or has

Algorithm 1: Release of Task τ_e at Time r_e on P_k

```
variables:
    $RQ_k$, $CQ_k$, $IS_k$ - ready queue, chunk queue, and the set of imminent tasks,
    $minProg$, $t_{update}$ - the last update on the progress of the back-runner, and the time of the update,
output:
    updated $RQ_k$, $CQ_k$, $minProg$, and $t_{update}$
release ($\tau_e$,$r_e$):
    $W_{tail} = t_{update} + \sum_{\tau_a^l \in CQ_k[minProg+1:prog_k^{tail}]} C_a^l$
    if $RQ_k = \emptyset$ and $r_e \geq W_{tail}$ :
        $minProg = prog_k^{tail}$; $t_{update} = r_e$
    else:
        broadcast($prog_k^{cur}$) to all other nodes
        set timeout to $r_e + Timeout$
        Let $SlackBound \longleftarrow \infty, IT \longleftarrow IS_k$
        foreach $\tau_a^{next_a,-}$ in $RQ_k$ :
            $SlackBound = \min(SlackBound, \min_{\tau_i \in IT \cap hp_a} (\rho_i(r_e) + slack_i))$
            $RQ_k, CQ_k, W_{tail}$ = moveChunks($W_{tail}$,$\tau_a$,$next_a$,$SlackBound$)
            if $\tau_a \notin RQ_k$ : $IT = IT \backslash hp_a$
            else: break
    $RQ_k$.insertAtPriority($\tau_e^{1,-}$)
    $r_e^{last} = r_e$
```

finished all released jobs. However, since P_k is healthy and $prog_b^{sent} < prog_k^{sent} \leq MAP$, it means P_b idled without its next progress exceeding the MAP while jobs were pending. Therefore, P_b is faulty. Additionally, P_b is faulty if exceeding the chunks' WCETs during execution. □

Illustrative Example. In the example depicted in Figure 2(c), after r_b is released, the back-runner P_1 broadcasts its progress following the completion of τ_d. After the timeout, the update algorithm is executed on P_1 at t_5 and on P_2 at t_6. Upon exchanging progress data, nodes P_1 and P_2 check each other's progress. Since P_1 has smaller progress and the WCP of P_1's progress before the update is greater than r_b, it is detected as the healthy back-runner. Consequently, the nodes update $minProg$ based on the progress broadcast by P_1 and t_{update} based on $r_b + Timeout$. As a result, the updated WCP considers only the WCET of chunks from τ_e^1 and subsequent chunks. This less pessimistic WCP increases the MAP, enabling nodes to move more chunks to the chunk queue for execution without idling.

6 Detailed Algorithms

In this section, we detail the algorithms that coordinate the execution order of tasks among nodes. The pseudo-codes for the three components of our method are outlined in Algorithms 1–4. Algorithm 2 serves as a supplementary function, reducing repetition. These algorithms address job releasing, scheduling chunks, and updating $minProg$ and t_{update} based on communicated information. Depending on the system model, the release and update algorithms can run either immediately after receiving an event and after the communication timeout, respectively, or be deferred until the currently executing chunk completes. The scheduling algorithm, however, always runs before executing chunks.

Algorithm 2: Slack Check to Move Chunks of a Chunk from the Ready Queue to the Chunk Queue

```
variables:
    $RQ_k$, $CQ_k$ - ready queue, chunk queue,
output:
    updated $RQ_k$, $CQ_k$ and $W_{tail}$ which is the WCP of the final chunk in the chunk queue
moveChunks ($W_{tail}$,$\tau_a$,$next_a$, $SlackBound$):
    $safe_a = max\{x | W_{tail} + \sum_{l=next_a}^{x} C_a^l \leq SlackBound\}$
    $CQ_k$.insertChunksAtTail($\tau_a$, $next_a$, $safe_a$)
    $W_{tail} = W_{tail} + \sum_{l=next_a}^{safe_a} C_a^l$
    if $safe_a < L_a$ : $RQ_k$.updateHead($next_a = safe_a + 1$)
    else: $RQ_k$.remove($\tau_a^{next_a,-}$)
    return $RQ_k$, $CQ_k$, $W_{tail}$
```

Algorithm 3: Scheduling the Next Chunk on P_k

```
variables:
    $RQ_k$, $CQ_k$, $IS_k$ - ready queue, chunk queue, and the set of imminent tasks,
    $minProg$, $t_{update}$ - the last update on the progress of the back-runner, and the time of the update,
    $t$, $prog_k^{cur}$ - the current local time, and the pointer to last chunk of $CQ_k$ that $P_k$ started its execution
output:
    updated $CQ_k$ and $RQ_k$,
schedule():
    if $prog_k^{cur} < prog_k^{tail}$ : execute($CQ_k[prog_k^{cur} + 1]$)
    else if $RQ_k$.isEmpty() : idle
    else:
        $W_{tail} = t_{update} + \sum_{\tau_a^l \in CQ_k[minProg+1:prog_k^{tail}]} C_a^l$
        Let $\tau_a^{next_a,-} = RQ_k$.head()
        $SlackBound = \min_{\tau_i \in IS_k \cap hp_a} (\rho_i(t) + slack_i)$
        if $W_{tail} + C_a^{next_a} \leq SlackBound$ :
            $RQ_k$, $CQ_k$, $W_{tail}$ = moveChunks($W_{tail}$,$\tau_a$,$next_a$,$SlackBound$)
            execute($CQ_k$.chunks[$prog_k^{cur} + 1$])
        else:
            $ISV = \{\tau_i \in IS_k \cap hp_a |$ Equation 3 does not hold$\}$
            idle until $t' = \max_{\tau_i \in ISV} (\theta(\tau_i))$
```

We will discuss the compatibility of our method with coarsely synchronized clocks in Section 7 and Appendix B, elaborate on the implemented optimizations in Appendix A, and analyze the runtime overhead of the algorithms in Appendix C.1.

6.1 Release Algorithm

Algorithm 1 shows the pseudo-code for releasing a task τ_e at time r_e on a node P_k. The release algorithm performs the following at the release of each task:

— Updates $minProg$ and t_{update} if it can be verified that the back-runner is idle (lines 8–9).

Algorithm 4: Updating the Information of the Back-Runner Node on P_k

```
variables:
    $PROG$ - The set of progress information $prog_j^{sent}$ (received from nodes before $r_e + Timeout$ by messages
      timestamped before $r_e + C_{send}$), including $prog_k^{sent}$ which is the progress of the node $P_k$ sent at the release $\tau_e$,
    $minProg$, $t_{update}$ - the last update on the progress of the back-runner, and the time of the update,
output:
    updated $minProg$, $t_{update}$
Update ($r_e$):
    do:
        $brp = \min_{prog_j^{sent} \in PROG} prog_j^{sent}$
        $PROG$.remove($brp$)
        $W_{br} = t_{update} + \sum_{\tau_a^l \in CQ_k[minProg+1:brp]} C_a^l$
    while $brp < prog_k^{sent}$ and $W_{br} < r_e$;
    $minProg = brp$, $t_{update} = min(W_{br}, r_e + Timeout)$
```

—Initiates the non-blocking communication to send the node's progress to other nodes (lines 11–12).
—Moves chunks from the jobs in the ready queue to the chunk queue until finding MAP (lines 13–18).
—Inserts the released job into the resulting ready queue based on the job's priority (line 19)

First, the algorithm computes the WCP of $prog_k^{tail}$ (line 7) to determine whether the back-runner has been idle and, if so, updates $MinProg$ and t_{update}. If the conditions in Lemma 2 hold for $t = r_e$, the algorithm updates $minProg$ and t_{update} to reflect $prog_k^{tail}$ and r_e respectively (lines 8–9). Otherwise, the algorithm broadcasts the current progress of P_k to all other nodes while setting a timeout to r_e + *Timeout* for invoking the update algorithm (lines 11–12).

After initiating the communication, the algorithm finds the MAP by transferring the chunks of the jobs from the ready queue to the chunk queue. To find the MAP, first, the algorithm calculates *SlackBound* as the minimum of $\rho_i(r_e) + slack_i$ for all imminent tasks τ_i with higher priority than the first job in the ready queue τ_a (line 15). Then it moves the chunks of τ_a to the chunk queue (line 16) using the function depicted in Algorithm 2. The descending order of priorities in RQ_k enables Algorithm 1 to calculate *SlackBound* efficiently by leveraging the previously computed value of *SlackBound* obtained from the last iteration. For example, to calculate the *SlackBound* for hp_b, after moving τ_a to the chunk queue, the algorithm just needs to update the *SlackBound* to include the minimum of $\rho_i(r_e) + slack_i$ for hp_b in the set of IT which only contains $IS_k \backslash hp_a$ (line 17). Algorithm 1 continues moving chunks to the chunk queue until the *moveChunks* function is unable to remove a job from the ready queue which means Equation (2) does not hold for $\tau_a^{next_a}$. At this point, the release algorithm exits the loop (line 18) as the final progress in the chunk queue ($prog_k^{tail}$) is the MAP. Therefore, the algorithm inserts the released job based on its priority in the ready queue and sets the ID of its first chunk to 1 (line 19). At line 20, the time of the last release of τ_e is updated to r_e.

Moving Chunks. The *moveChunks* function shown in Algorithm 2 moves the chunks of a job τ_a at the head of the ready queue to the chunk queue as long as Equation (2) holds. Initially, it finds the maximum ID, $safe_a$, among the chunks of τ_a, such that adding the WCET of chunks from $next_a$ until $safe_a$ to the WCP of $prog_k^{tail}$ does not exceed the earliest release plus the slack of higher-priority imminent tasks (line 6). Subsequently, it enqueues the chunks of τ_a up to $\tau_a^{safe_a}$ at the tail of the chunk queue (line 7). It also updates the WCP of $prog_k^{tail}$ after each modification (line 8). $next_a$

is updated based on $safe_a$, or job τ_a is removed from the ready queue if all its chunks have been moved to the chunk queue (lines 9–10).

6.2 Scheduling Algorithm

Algorithm 3 illustrates the pseudo-code for our schedule function. This function executes only when there are no pending releases or updates (otherwise, these algorithms will be executed first). It runs in response to any of the following events: (1) when a node switches to a new job after finishing a chunk, (2) when imminent tasks have not been released by the end of an idle period, or (3) when the node is idle after running the release or update algorithm. In a nutshell, the scheduling algorithm performs the following operations before starting the execution of a chunk:

— Executes the next chunk in the chunk queue that has not yet been executed (line 8).
— If Equation (2) holds for all $\tau_i \in IS_k \cap hp_a$, moves chunks of the next job of RQ_k to CQ_k and executes (lines 13–16).
— If no chunk is awaiting execution in CQ_k, enforces idle time until when the Equation (2) holds again (lines 18–19).

First, the algorithm checks for any unexecuted chunks in CQ_k. If there are chunks in CQ_k after $prog_k^{cur}$ (i.e., $prog_k^{cur} < prog_k^{tail}$), the algorithm executes the next chunk in CQ_k (line 8). Otherwise, the algorithm checks RQ_k. If RQ_k is empty, the node remains idle (line 9). If there are jobs in RQ_k, the algorithm checks whether Equation (2) holds (lines 11–14). If true, the algorithm calls *moveChunks* function to transfer chunks from the job at the head of RQ_k to CQ_k (line 15) and proceeds to execute the next chunk (line 16). Otherwise, the node calculates the time t' by which the next chunk of the first job in RQ_k can be moved to CQ_k for execution. t' is determined by the maximum of $\theta(\tau_i)$, calculated using Equation (3), for each imminent task $\tau_i \in hp_a$ that Equation (2) does not hold for. The node remains idle until t' (line 19), awaiting the release of any task. If no task is released by t', the scheduling algorithm will be executed again.

Note that, although it is not explicitly illustrated in Algorithm 3, there are instances where the first chunk in RQ_k is moved to CQ_k without needing to check the slack of higher priority imminent tasks. This occurs when all of the chunks in the CQ_k after *MinProg* have been completed close to their WCETs, meaning that the WCP of $prog_k^{tail}$ is close to the current time. Moving the first chunk of the RQ_k to CQ_k in this scenario is always possible since the schedulability test ensures that the slack of any task is at least as large as the maximum length of any chunk of any lower-priority task.

6.3 Update Algorithm

Nodes broadcast their progress when a task τ_e is released at r_e and the conditions of Lemma 2 do not hold for $t = r_e$. Once a node receives progress information from all other nodes and the broadcast timeout is triggered at $r_e + Timeout$, the node runs the update algorithm. The main functions of the Update algorithm are the following:

— Finding faulty back-runners (lines 8–11).
— Updating *MinProg* and t_{update} based on the progress of the non-faulty back-runner and the time of update (line 12).

Before executing the update algorithm, while node P_k is receiving progress information, it constructs the set *PROG*, which contains progress data sent by other nodes ($prog_j^{sent}$) before $r_e + C_{send}$, along with its own progress ($prog_k^{sent}$). Initially, the update algorithm identifies the progress of faulty back-runners. The algorithm chooses the minimum progress in *PROG* for checking the conditions in Lemma 3 while removing it from the set (lines 8–9). If a progress is found faulty, the node evaluates the next back-runner progress in the remaining *PROG* until it identifies the non-faulty back-runner (lines10–11). If there is no non-faulty progress less than the node's own progress (i.e., $brp = prog_k^{sent}$), the node considers its own progress as the non-faulty back-runner's progress.

Finally, the algorithm updates *minProg* to the non-faulty back-runner's progress. It updates t_{update} to the minimum of the time of the timeout and W_{br}, which is the WCP of the non-faulty back-runner's progress computed based on the previously set *minProg* and t_{update}. This ensures that the WCP always reflects the behavior of a back-runner that requires the WCET of chunks to execute them, and prevents the late update at $r_e + Timeout$ from increasing the WCP.

6.4 Guaranteeing Timely Execution

Our method ensures that tasks meet their deadlines under both fixed and dynamic priority scheduling algorithms, provided that the task set is schedulable on a single node using the same scheduling algorithm. This assumes tasks are only preemptible at chunk boundaries, and that the WCET of each chunk is defined as the maximum WCET of the chunk among all its replicas. For dynamic priority scheduling algorithms such as EDF, job priorities are determined at runtime. Once a job is released, its absolute deadline—and consequently its priority under the EDF algorithm—is fixed. Similarly, for an imminent task (τ_i), we can calculate its earliest possible absolute deadline at time (t) by adding its earliest possible release time ($\rho_i(t)$) to its relative deadline (D_i). This earliest absolute deadline is then used to establish the task's priority.

We calculate the slack for each task τ_i based on [43], which employs the method from [5] and incorporates the release overhead as described in [49]. If S_i is defined as the set of lower-priority tasks than τ_i for RM (i.e., $S_i = lp_i$), or as the set of tasks with a larger (or equal) relative deadline than D_i for EDF (i.e., $S_i = \{\forall \tau_j \in \mathbb{T} | j \neq i \wedge D_i \leq D_j\}$), then the task set is schedulable using RM or EDF, respectively, if:

$$\forall \tau_j \in S_i, \max_{k \leq L_j}(C_j^k) \leq slack_i. \tag{4}$$

This guarantees that the slack of each task exceeds the WCET of the non-preemptive chunks of tasks in S_i, which is a sufficient condition for a task set to be schedulable by our method. We will now describe how our release, scheduling, and update algorithms ensure that no job misses its deadline.

Lemma 4. *Inserting a released job after all chunks until the MAP and before the lower-priority jobs remained in the ready queue does not lead to deadline misses.*

Proof. The release algorithm, when finding the MAP, ensures the slack of imminent tasks exceeds the maximum priority inversion from executing chunks until the MAP (lines 15–16 in Algorithm 1 and lines 6–7 in Algorithm 2). Also, after moving chunks until the MAP to the chunk queue and inserting a task before its lower-priority jobs in the ready queue (line 19 in Algorithm 1), no other lower-priority task is inserted before the insertion point. Therefore, it is always ensured that the priority inversion is less than the slack of the tasks and the deadlines of the tasks are guaranteed. □

Lemma 5. *The idle time enforced by the scheduling algorithm does not cause any deadline miss.*

Proof. This idle time is only introduced when a node completes chunks earlier than their WCETs. If a node finishes the execution of chunks close to their WCETs, then $t \approx \omega(prog_k^{cur})$. In this case, if $prog_k^{cur} < prog_k^{tail}$, the next chunk will be executed (line 8 in Algorithm 3). If $prog_k^{cur} = prog_k^{tail}$, since the condition in Equation (4) guarantees that the WCET of the chunks of each task is less than the slack of its higher-priority tasks, the condition in Equation (2) is satisfied, allowing the next chunk to be moved to the chunk queue for execution (lines 14–15 in Algorithm 3). Therefore, our scheduling algorithm never imposes idle time on a back-runner executing chunks near their WCETs and only utilizes dynamic slack created by the early completion of jobs. □

Communication for updating *minProg* and t_{update} does not cause idle time, as nodes broadcast progress information either in parallel with job execution (using DMA-based NICs or co-processors) or when the node is already idle. Neither the release nor the scheduling algorithms wait for communication results, so communication never impacts timeliness or delays job execution. Instead, the update algorithm leverages the received information to reduce idle time by decreasing WCP pessimism and increasing MAP, allowing nodes to execute more chunks from the released jobs without remaining idle. Furthermore, no node can cause deadline misses on others by sending faulty or malicious progress information, making the method immune to priority inversion injection attacks.

LEMMA 6. *Assuming the WCP of each progress already provides a correct upper bound on when a back-runner reaches a given progress, timeliness can still be guaranteed after an update even if a node sends malicious progress information.*

PROOF. In our method, nodes determine chunk execution order without relying on others' progress. Instead, they use the WCP, slack, and task release times to compute MAP, fix chunk order in the chunk queue, and insert jobs into the ready queue. Although the update algorithm assigns *minProg* and t_{update} based on communicated data, a malicious node cannot disrupt WCP from being a correct upper bound. If the WCP is already correct before the update, Lemma 3 ensures that malicious back-runners—those reporting progress lower than what is achievable by executing chunks at their WCETs—are dismissed. Thus, *minProg* remains safely bounded between the progress of a node executing chunks at WCETs (still a healthy node) and the progress of the healthy node running the update algorithm. Similarly, t_{update} is determined based on the WCP before the update and the time of the timeout, rather than by communicated data. These ensure the WCP remains a correct upper bound, preventing malicious information from causing deadline misses. □

7 Extension to Coarsely Synchronized Clocks

Although we assumed synchronized clocks for simplicity, our method also works with coarsely synchronized clocks if events are timestamped before being sent to nodes. In this section, we account for Δ, the maximum difference between an event's timestamp and its reception time according to a node's local clock, and δ, the maximum difference between time values read by nodes from their respective clocks. We then outline the modifications required for each algorithm to operate under coarse synchronization. As shown in the Appendix B, these changes do not affect timeliness when Δ is treated as release jitter in the schedulability test.

7.1 Accounting for Event Latency at Release

Delays in receiving events can postpone task execution, requiring an adjustment to the WCP parameters. To account for this delay, we incorporate it into t_{update} by setting $t_{\text{update}} = r_e + \Delta$ at line 9 in Algorithm 1. This ensures that the WCP of a progress remains an upper bound on the point in time when the back-runner finishes that progress, even when the events are delayed. Despite using coarsely synchronized clocks, the release algorithm requires no further modifications, as it relies on event timestamps rather than the local time of individual nodes to determine the MAP.

Note that the release algorithm executes only when the node's local time reaches or exceeds the event's timestamp (r_e). This ensures that a node never reports a progress lower than what it has at r_e, nor executes a job before its release time according to its local clock.

7.2 Maintaining Correct Interaction of Scheduling and Release Algorithms

The scheduling algorithm requires specific adjustments to handle coarsely synchronized clocks effectively. These adjustments ensure that if the scheduling algorithm moves some chunks to the chunk queue before a job's release, all other nodes also transfer those chunks to their respective chunk queues before inserting the released job into the ready queue. Consider a scenario where an event to release τ_e with timestamp r_e is received immediately after a node executes the scheduling algorithm. It might be the case that due to the coarsely synchronized clock or late arrival of the event, the local time t of the node during scheduling which happened just before the release, is ahead of r_e. As in the scheduling algorithm the next release of imminent tasks is evaluated based on the local time of the nodes, it is crucial to restrict the chunks moved by the scheduling algorithm to the chunk queue. This ensures that when inserting τ_e into the ready queue, the MAP computed by the release algorithm based on r_e is always greater than or equal to $prog_k^{tail}$, allowing all nodes to construct identical ready queues and insert τ_e in the same position.

The following modifications to the scheduling algorithm make it compatible with coarsely synchronized clocks:

Lemma 7. *The scheduling algorithm at time t on node P_k ensures $prog_k^{tail}$ never exceeds the MAP calculated at a job's release r_e by the release algorithm, if it considers the earliest release of imminent tasks from $t - \Delta$ ($\leq r_e$) instead of t.*

Proof. In the scheduling algorithm for coarsely synchronized clocks, the next release time of each imminent task τ_i is calculated as the maximum of $r_e^{last} + T_i$ and $t - \Delta$. Before the release of a task τ_e, the earliest release time of every imminent task τ_i, determined based on $t - \Delta$, is always earlier than or equal to its earliest release time determined at r_e (i.e., $\rho_i(t - \Delta) \leq \rho_i(r_e)$). Therefore, the chunks that are moved to the CQ_k by the scheduling algorithm based on $t - \Delta$ and Equation (2), never exceed the chunks that are moved by the release algorithm based on r_e, even if $r_e < t$ due to the coarse synchronization. As a result, the scheduling algorithm ensures that $prog_k^{tail}$ never exceeds the MAP, which is determined at the time of the next release (r_e). □

Note that, if the scheduling algorithm fails to move any chunk to the chunk queue, the node P_k will remain idle until $t' + \Delta$, at which point it can proceed with moving a chunk from the first job in the RQ_k to CQ_k.

7.3 Safeguarding Consistent WCP Updates

With coarsely synchronized clocks, two key adjustments are needed regarding the update algorithm. First, the release and update algorithms must be called in the correct order based on their timestamps. Second, nodes must use the same progress data when updating *minProg* and t_{update} to ensure identifying the same healthy back-runner across all nodes.

To maintain the execution order of the release and update algorithms based on their timestamps, the update algorithm only executes after r_e + Timeout + Δ. This guarantees that no event with a timestamp earlier than r_e + Timeout remains pending before running the update algorithm. As a result, at line 12 of Algorithm 4, we must set t_{update} to $r_e + Timeout + \Delta$ instead of $r_e + Timeout$. This ensures that if all nodes are fast and are being idle to avoid exceeding MAP, the next WCP starts at $r_e + Timeout + \Delta$, the earliest time at which they can resume executing chunks after the update. Consequently, WCP remains an upper bound on the time when a node completes a progress.

The update algorithm relies on both the event's timestamp, r_e, and a *Timeout* value known across all nodes. Therefore, since it is not reliant on the local time of individual nodes, the rest of the algorithm remains unchanged even with coarsely synchronized clocks. However, to ensure that the algorithm updates *minProg* and t_{update} to identical values on all nodes, we modify C_{send} and *Timeout*.

Lemma 8. *Let C_{send}^{sync} denote the maximum delay before a node begins sending its progress information after receiving an event in a fully synchronized system, where all nodes receive events simultaneously. In a system with coarsely synchronized clocks, all nodes will consider the same progress information when executing the update algorithm after the timeout, provided that $C_{send} \geq C_{send}^{sync} + \Delta$ and $Timeout \geq C_{send} + C_{msg} + \delta$.*

Proof. For an event timestamped at r_e, if C_{send} includes Δ, it can be ensured that every healthy node receives the event and sends its progress before $r_e + C_{send}$ even if it experiences the maximum delay in receiving the event. Any progress information timestamped after $r_e + C_{send}$ is disregarded by all nodes. Additionally, every healthy node receives and delivers progress information from others within $C_{send} + C_{msg} + \delta$, despite clock differences up to δ. If a node does not receive this information within this time, all nodes discard the information, since the delay cannot be attributed to clock deviation, which has already been accounted for. Therefore, for any *Timeout* greater than this value, we ensure that all healthy nodes consider the same set of progress values, or they all discard the progress information. □

Since all nodes execute the release and update algorithms in the same order and receive identical progress information, they can calculate the same WCP for each progress based on the previous *minProg* and t_{update} and also update *minProg* and t_{update} to the same values, using the non-faulty back-runner's progress and the timeout timestamp.

8 Evaluation

We implemented our approach, referred to in this section as **Limited Priority Inversion Total Order based on Maximum Allowed Progress (LPI-MAP)**, and compared it with the method by Rodrigues et al. [53], the approach by Wang et al. [58], and LPI-FRP by Naghavi and Navet [43]. We compare these methods based on their acceptance ratio and analyze job response times using each approach. To ensure a fair comparison of response times, we modified all methods to support limited preemption and work with chunks. Additionally, we compare our approach with a variant of the *Simple* method mentioned in Naghavi and Navet [43]. This method is non-work-conserving and always idles until a chunk's WCET, when the chunk is completed before its WCET. To maintain the same total execution order, Simple requires either precisely synchronized clocks to ensure that preemptions occur simultaneously across all nodes, or the nodes must remain idle after finishing each chunk long enough to offset the clock imprecision. Notice also that while Rodrigues and Wang are concerned with the total order in the processing of prioritized events, these works do not claim to achieve any real-time guarantees.

We evaluate the ratio of schedulable task sets, as well as the impact of the aforementioned methods on both average and worst-case response times. Additionally, we demonstrate that Rodrigues and Wang fail to meet task deadlines and measure the effect of communication delay on the performance of our method. The overhead of our solution, along with the impact of each method on the response time for individual tasks are explored in the Appendix C.2 of this article.

8.1 Experimental Setup

We implemented all the approaches evaluated in this study in C++,[3] considering synchronized clocks to simulate the execution of jobs and obtain their response times using different methods.

[3]Available at https://github.com/aminnaghavi/TOFT-RTS, released under AGPL v3.0 license. Results are reproducible via the provided code and instructions.

Additionally, we used a Raspberry Pi 4B running Linux at a frequency of 1.5 GHz to measure the WCET of the TACLeBench benchmark [16] tasks. We used the same platform to assess the overhead of our method, which is reported in the Appendix C.1.

We generated tasks using the Dirichlet-Rescale algorithm [21]. For each task set, we obtained 100 uniformly distributed random utilization values u_i, which collectively sum up to the targeted total utilization U. We then calculate minimum inter-release times from these u_i values, as $T_i = C_i/u_i$ where C_i is derived from the TACLeBench tasks, with WCET ranging between 0.1 and 100 ms. We generate chunks with a minimum worst-case execution time equivalent to WCET of the smallest task and ensure that chunk sizes are closely similar. Thus, the WCETs for the chunks are uniformly and randomly selected from the interval $[c^{min}, c^{max}]$, where we set $c^{min} = 0.1$ ms and $c^{max} = 0.12$ ms. We allow some chunks (chosen randomly) to uniformly exceed c^{max} to ensure that the sum of WCETs of chunks of a task τ_i equals C_i (i.e., $\sum_{l=1}^{L_i} C_i^l = C_i$). Therefore, the number of chunks of a task τ_i, depending on the WCET of the task and the WCET of the chunks, ranges between $\max(\lfloor C_i/c^{max} \rfloor, 1)$ and $\lfloor C_i/c^{min} \rfloor$.

We set $D_i = T_i$, though our method works for any $D_i \le T_i$. We assume the **Best-Case Execution Time (BCET)** of each task, representing the minimum time for its jobs to complete, is 20% of its WCET, as in [4, 65]. The choice of BCET influences the range from which chunk execution times are selected, which in turn impacts the variation in execution times across different replicas of a job. The period adjustment method proposed by Xu [59] was implemented to mitigate the occurrence of extreme hyperperiods. For each task set, we generate 100,000 jobs. The release times of jobs are generated randomly so that between two successive jobs of a task τ_i, there is a gap between T_i and $2T_i$.

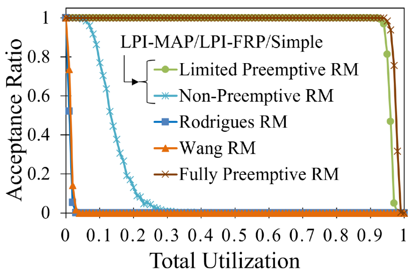
We use RM and EDF scheduling algorithms and simulate job execution in two scenarios: *Normal* and *Worst-Case*. *Normal scenario* is defined as a scenario where jobs' actual execution times are chosen randomly using a uniform distribution between their BCETs and WCETs. We consider *worst-case scenario* on the other hand as a scenario in which $\lfloor \frac{n}{2} \rfloor$ malicious nodes mimic the progress of a healthy front-runner that executes jobs at their BCETs, while another healthy node executes jobs at their WCETs. We randomly choose the execution time between each two milestones from $[0, C_i^l]$ so that they sum up to the actual execution time of the job.

We added synthetic communication delays to measure response times. We set $Timeout = 20\,\mu s$ as the upper bound of the communication delays to send the progress information. Even though our LPI-MAP method remains effective even under increased communication delays (see Section 8.4), we chose this small delay to ensure a fair comparison with Wang, Rodrigues, and LPI-FRP methods, as their effectiveness heavily relies on communication (see Section 3). We also performed a parameter study to understand how our approach works under different communication conditions.

8.2 Acceptance Ratio

We determine the acceptance ratio of different methods for various total utilizations using RM and EDF scheduling. We used the settings described in Section 8.1 to generate 1,000 task sets, each containing 100 tasks, for each total utilization value. Figure 3 shows the acceptance ratio for our LPI-MAP approach, LPI-FRP, and simple methods (under both limited preemptive and non-preemptive scheduling), along with the approaches of Rodrigues and Wang. It also includes the acceptance ratio of unmodified preemptive RM/EDF on a uniprocessor for reference.

The non-preemptive and limited preemptive versions of our method use the same criteria for accepting tasks as the non-preemptive and limited preemptive versions of LPI-FRP and the Simple methods, respectively. Note that the limited preemptive version of LPI-FRP can only guarantee



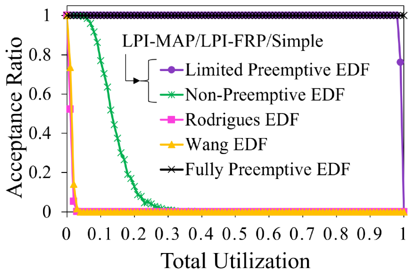


Fig. 3. Fraction of schedulable task sets under our LPI-MAP approach, LPI-FRP, and the Simple approaches (limited or non-preemptive), compared with Rodrigues' and Wang's methods, as well as fully preemptive uniprocessor scheduling for RM (left) and EDF (right).

deadlines if the communication timeout is smaller than the WCET of the smallest chunk, as is the case in the selected experimental settings.

Assuming a plausibility check ensures jobs are not released earlier than their minimum inter-release times, malicious nodes can exploit Rodrigues' method and RM by falsely claiming to have executed one job from each lower-priority task upon τ_i's release. This can delay τ_i's execution significantly, potentially causing a deadline miss. Similarly, under EDF, malicious nodes may claim to have executed one job from each released task with a larger deadline to inject priority inversion. Additionally, in Wang's method, malicious nodes may exploit a front-runner's progress by pretending to execute jobs at their BCETs while a healthy node executes them at their WCETs. In this scenario, a newly released job on the back-runner will be delayed until the back-runner finishes all jobs completed by the front-runner, potentially causing a deadline miss. If tasks have sufficient slacks, they can neutralize such attacks in both Rodrigues' and Wang's methods. According to this, Naghavi and Navet [43] presented a schedulability test for these methods. Since Rodrigues' and Wang's methods remain vulnerable to these attacks even with limited preemption, the schedulability test in [43] can assess whether a task set remains schedulable under these methods, regardless of the preemption model.

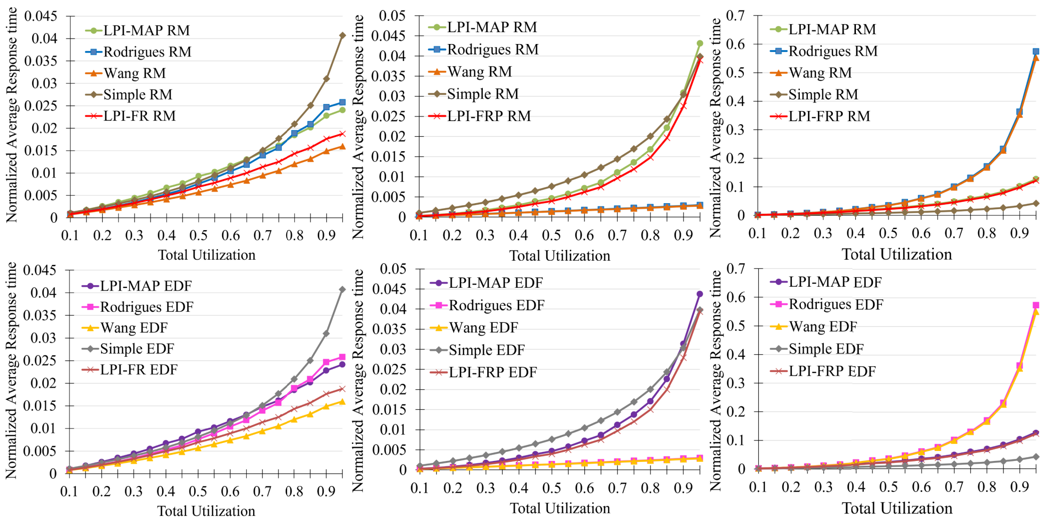
As shown in Figure 3, Wang and Rodrigues relatively quickly (after a total utilization of 0.05) run into sets for which not all tasks can be guaranteed to meet their deadlines. In contrast, our solution, along with the modified versions of Simple and LPI-FRP methods that work with limited preemption, stays close to the bound for fully preemptive RM and EDF, accepting task sets with utilization up to 0.91 for RM and 0.98 for EDF. Note that the unmodified LPI-FRP method, which only supports non-preemptive scheduling, would yield a lower acceptance ratio, as shown in Figure 3.

8.3 Response Times

Although real-time systems are primarily concerned with guaranteeing task deadlines, there are several secondary concerns where the actual response times of jobs play a crucial role (e.g., power management, aperiodic tasks, background load, mixed-criticality applications).

To investigate response times, we generated 100 task sets for each total utilization using parameters mentioned in Section 8.1 and simulated the execution of 100,000 jobs per task set, replicated on five nodes (i.e., 500,000 jobs in total for each task set). Each of these task sets is schedulable based on the criteria mentioned in Equation (4).

Figures 4(a) and 5(a) depict, for different total utilization values, the normalized average and worst response times of tasks averaged across all tasks and task sets in the normal scenario. The normalized average and worst response time for each task represent the average and maximum



(a) normal scenario (all nodes) (b) worst-case scenario (front-runner) (c) worst-case scenario (back-runner)

Fig. 4. The impact of different methods on the average response time of tasks normalized by their deadlines. Results are shown for all nodes in the normal scenario (left), for the front-runner node (middle), and the back-runner node (right) in the worst-case scenario. Results for RM are displayed at the top and for EDF at the bottom.

response time, respectively, across all jobs of the task on all nodes, normalized by the task's deadline. Figures 4(b) and 5(b) illustrate the normalized average and worst response times of tasks averaged over all tasks and task sets for the healthy front-runner node in the worst-case scenario, while Figures 4(c) and 5(c) show the same for a healthy back-runner. We present the results for both RM and EDF in Figures 4 and 5.

Our LPI-MAP method enables nodes to progress independently, only requiring a node to remain idle if executing the next chunk jeopardizes the deadline of imminent tasks on the back-runner. In total, as can be seen in Figure 4(a), this leads to a reduction in average response times compared to Simple in the normal scenario. For example, at a total utilization of 0.95, using our LPI-MAP method, the average response time of tasks is 2.4% of their deadline intervals, while in the Simple method, tasks respond in 4.1% of their deadlines. On average, the LPI-MAP method performs similarly to the Rodrigues method. While the limited preemptive version of LPI-FRP and Wang methods achieve better average response times in normal scenarios, their reliance on communication makes them less robust.

On the front-runner in the worst-case scenario (Figure 4(b)), for the total utilization values smaller than 0.9, our method performs closely to the limited preemptive version of the LPI-FRP method and provides lower average response times compared to the Simple method. Furthermore, the similarity of the response times of the Simple method in Figure 4(a) and (b) indicates the Simple method's inability to benefit from the early completion of jobs. Our method performs similarly to the Simple method when total utilization reaches 0.9 or higher. This convergence occurs because a back-runner node executes jobs at their WCET; therefore, at high utilization, due to the small slacks of the tasks, front-runner node cannot execute many chunks ahead of the back-runner. As a result, the front-runner often remains idle to prevent jeopardizing jobs' deadlines on the back-runner, leading to increased response times.

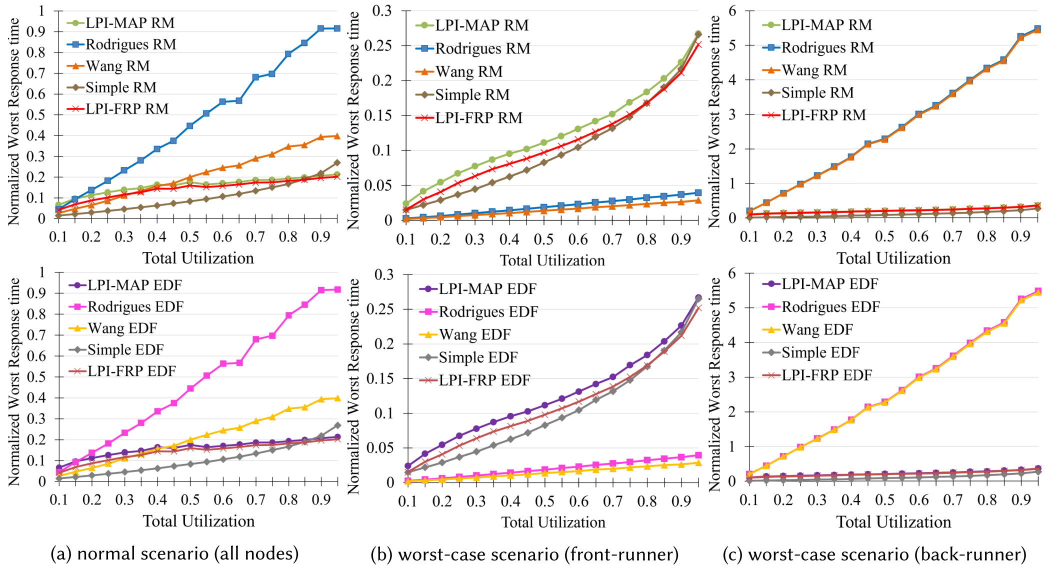


Fig. 5. The impact of different methods on the worst response time of tasks normalized by their deadlines. Results are shown for all nodes in the normal scenario (left), for the front-runner node (middle), and the back-runner node (right) in the worst-case scenario. Results for RM are displayed at the top and for EDF at the bottom.

The average response time of tasks on the front-runner in Rodrigues and Wang's methods is very low because the front-runner does not wait for the execution of jobs on the back-runner. However, we will later demonstrate that this results in deadline misses on the back-runner.

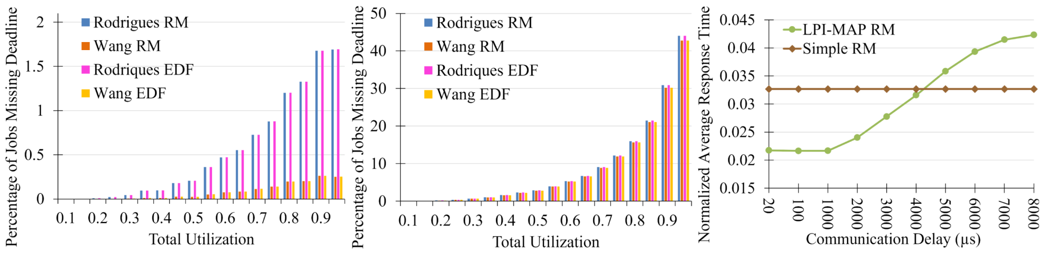
Based on Figure 4(c), the Simple method achieves the lowest average response time for jobs on the back-runner. The reason is that the Simple method does not allow priority inversion beyond the WCET of a single chunk. This figure also demonstrates that our method is able to effectively maintain low response times on the back-runner.

Figure 5(a)–(c) shows that the worst response time is lower in the Simple method than other methods. This is because these methods, unlike the Simple method, might delay the execution of a job by more than one chunk of lower-priority jobs. Nevertheless, the worst response time of our LPI-MAP method (as well as the LPI-FRP) remains significantly below 1, even on the back-runner in the worst-case scenario. As can be seen in Figures 5(a), Rodrigues and Wang have higher worst response times in the normal scenario than our method and the Simple method. Especially, in high total utilizations where Rodrigues has the worst normalized response times close to 1. This shows that in most of the task sets with high total utilization, tasks had worst response times near their deadlines.

In the worst-case scenario on the back-runner (Figure 5(c)), at the total utilization of 0.3 or higher, Rodrigues's and Wang's normalized worst response times exceed 1. This shows the back-runner's worst response times are significantly impacted by the front-runner's behavior, leading to instances where tasks exceed their deadlines on the back-runner.

Rodrigues and Wang's methods miss job deadlines not only in the worst-case scenario but also in the normal scenario. We measured the percentage of jobs whose deadlines were not met across all five nodes in the normal scenario (Figure 6(a)) and on the back-runner in the worst-case scenario (Figure 6(b)) for RM and EDF.

At a total utilization of 0.95 with RM in the normal scenario, on average (over 100 task sets), Rodrigues misses the deadline of 1.69% of jobs, while Wang misses the deadline of 0.25%. In the



(a) Deadline misses in Rodrigues and Wang: normal scenario

(b) Deadline misses in Rodrigues and Wang: worst-case scenario

(c) Communication impact on response time of LPI-MAP method

Fig. 6. Percentage of jobs missing their deadlines in Rodrigues and Wang using RM and EDF in the normal (left) and the worst-case (middle) scenarios, and impact of communication delay on the average response time of our LPI-MAP method using RM (right).

worst-case scenario, at the same total utilization and with RM, the number of jobs which missed their deadlines on the back-runner node are 44% and 42.77% for Rodrigues and Wang respectively. Our LPI-MAP method, as well as the simple and LPI-FRP method, never missed any deadlines and are therefore not depicted.

8.4 Impact of Communication Delay on Average Response Times

Learning that other nodes have completed jobs earlier than their WCETs when communicating progress information, allows nodes to process additional chunks. To evaluate how communication delays affect our approach, we measured the normalized average response time of all jobs across 100 schedulable task sets with total utilization of 0.9 for different communication delays. In Figure 6(c), communication times are varied from 20 μs to 100 μs, then to 1,000 μs, and are increased up to 9,000 μs in 1,000 μs increments. We measured the normalized average response time of our method, with the results from the Simple method—which is independent of communication time—serving as the baseline.

It can be observed that, for the task specifications mentioned in Section 8.1, our LPI-MAP method maintains a better average response time than Simple for communication delays of 4,000 μs and below. Beyond this point, Simple begins to outperform our method. This is due to the fact that progress updates received after such high communication delays cannot be effectively leveraged by our method. Consequently, the pessimistic WCP parameters reduce our method's efficiency in executing tasks early, while delaying higher-priority tasks by executing lower-priority tasks. Therefore, in high communication delays, our method improves the response times of lower-priority tasks slightly compared to their deadlines at the expense of increasing the response times of higher-priority tasks with shorter deadlines. This leads to an overall average response time higher than the Simple method. However, our method never misses any deadline even if communication between nodes cannot take place.

9 Conclusions

In this article, we propose a total order execution protocol that bounds priority inversion and tolerates priority inversion injection attacks without relying on communication. Communication is used solely to improve task response times and prevent nodes from idling. The protocol employs task milestones to ensure compatibility with both limited preemptive and non-preemptive scheduling and operates effectively with coarsely synchronized clocks. Our method leverages the temporal characteristics of tasks and the actual release times of their jobs to find consistent insertion points for the released jobs across nodes, ensuring total order. It allows nodes to progress independently in executing chunks of released jobs by utilizing the slack of higher-priority imminent tasks, without requiring coordination, while still meeting task deadlines on all nodes. Additionally, to avoid rollbacks, our approach may restrict the progress of nodes that have advanced further in executing chunks compared to other nodes. This restriction ensures that other nodes can continue executing chunks in the same order without jeopardizing task deadlines.

Our approach outperforms the methods proposed by Rodrigues and Wang in terms of acceptance ratio and improves the average response times of jobs compared to the Simple method, even under fairly high communication delays. Compared to Naghavi's method, our new approach imposes no restrictions on communication delay and works even without communication. Future research directions include extending our work to scenarios where not all tasks are replicated across nodes, but instead involve mixed workloads containing both replicated and non-replicated tasks.

References

[1] Hakan Aydin Abhishek Roy and Dakai Zhu. 2021. Energy-aware primary/backup scheduling of periodic real-time tasks on heterogeneous multicore systems. *Sustainable Computing: Informatics and Systems* 29 (2021), 100474. DOI: https://doi.org/10.1016/j.suscom.2020.100474

[2] Mani Amoozadeh, Arun Raghuramu, Chen-nee Chuah, Dipak Ghosal, H. Michael Zhang, Jeff Rowe, and Karl Levitt. 2015. Security vulnerabilities of connected vehicle streams and their impact on cooperative driving. *IEEE Communications Magazine* 53, 6 (2015), 126–132. DOI: https://doi.org/10.1109/MCOM.2015.7120028

[3] Alaa Askkar. 2011. PA Telecommunications Minister: Palestinian Internet under Hacking Attacks. IMENC. Retrieved from http://www.imemc.org/article/62409

[4] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. 2001. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS '01) (Cat. No. 01PR1420)*, 95–105. DOI: https://doi.org/10.1109/REAL.2001.990600

[5] Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito, and Marco Caccamo. 2010. Preemption points placement for sporadic task Sets. In *2010 22nd Euromicro Conference on Real-Time Systems*, 251–260. DOI: https://doi.org/10.1109/ECRTS.2010.9

[6] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt. 2005. Performance analysis of system overheads in TCP/IP workloads. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*, 218–228. DOI: https://doi.org/10.1109/PACT.2005.35

[7] Miguel Castro and Barbara Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems* 20, 4 (Nov. 2002), 398–461. DOI: https://doi.org/10.1145/571637.571640

[8] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. 2011. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*.

[9] Liming Chen and A. Avizienis. 1995. N-version programming: A fault-tolerance approach to reliability of software operation. In *25th International Symposium on Fault-Tolerant Computing*, '*Highlights from Twenty-Five Years*', 113. DOI: https://doi.org/10.1109/FTCSH.1995.532621

[10] Weifan Chen, Ivan Izhbirdeev, Denis Hoornaert, Shahin Roozkhosh, Patrick Carpanedo, Sanskriti Sharma, and Renato Mancuso. 2023. Low-overhead online assessment of timely progress as a system commodity. In *35th Euromicro Conference on Real-Time Systems (ECRTS '23)*. Alessandro V. Papadopoulos (Ed.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 262, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Dagstuhl, Article 13, 1–26. DOI: https://doi.org/10.4230/LIPIcs.ECRTS.2023.13

[11] US Federal Energy Regulatory Commission. 2016. Reliability Standards for Physical Security Measures. RD14-6-000.

[12] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. 2013. BFT-TO: Intrusion tolerance with less replicas. *The Computer Journal* 56, 6 (2013), 693–715. DOI: https://doi.org/10.1093/comjnl/bxs148

[13] Flaviu Cristian, Danny Dolev, Ray Strong, and Houtan Aghili. 1990. Atomic broadcast in a real-time environment. In *Fault-Tolerant Distributed Computing*. Barbara Simons and Alfred Spector (Eds.), Springer, New York, NY, 51–71.

[14] Sadegh Davari and Lui Sha. 1992. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *ACM SIGOPS Operating Systems Review* 26, 2 (Apr. 1992), 110–120. DOI: https://doi.org/10.1145/142111.142126

[15] Tobias Distler. 2021. Byzantine fault-tolerant state-machine replication from a systems perspective. *ACM Computing Surveys* 54, 1, Article 24 (Feb. 2021), 38 pages. DOI: https://doi.org/10.1145/3436728

[16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A benchmark collection

to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis.* DOI: https://doi.org/10.4230/OASIcs.WCET.2016.2

[17] Pietro Fara, Gabriele Serra, Alessandro Biondi, and Ciro Donnarumma. 2021. Scheduling replica voting in fixed-priority real-time systems. In *33rd Euromicro Conference on Real-Time Systems (ECRTS '21).* Björn B. Brandenburg (Ed.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Dagstuhl, Article 13, 1–21. DOI: https://doi.org/10.4230/LIPIcs.ECRTS.2021.13

[18] Neeraj Gandhi, Edo Roth, Robert Gifford, Linh Thi Xuan Phan, and Andreas Haeberlen. 2020. Bounded-time recovery for distributed real-time systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 110–123. DOI: https://doi.org/10.1109/RTAS48715.2020.00-13

[19] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. 2011. OS diversity for intrusion tolerance: Myth or reality? In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN)*, 383–394. DOI: https://doi.org/10.1109/DSN.2011.5958251

[20] Ajei Gopal, Ray Strong, Sam Toueg, and Flaviu Cristian. 1990. Early-delivery atomic broadcast. In *9th Annual ACM Symposium on Principles of Distributed Computing*, 297–309.

[21] David Griffin, Iain Bate, and Robert I. Davis. 2020. Generating utilization vectors for the systematic evaluation of schedulability tests. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, 76–88. DOI: https://doi.org/10.1109/RTSS49844.2020.00018

[22] Arpan Gujarati, Sergey Bozhko, and Björn B. Brandenburg .2020. Real-time replica consistency over ethernet with reliability bounds. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 376–389. DOI: https://doi.org/10.1109/RTAS48715.2020.00012

[23] Arpan Gujarati, Ningfeng Yang, and Björn B. Brandenburg. 2022. In-ConcReTeS: Interactive consistency meets distributed real-time systems, again! In *2022 IEEE Real-Time Systems Symposium (RTSS)*, 211–224. DOI: https://doi.org/10.1109/RTSS55097.2022.00027

[24] Mario Günzel, Harun Teper, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. 2023. On the equivalence of maximum reaction time and maximum data age for cause-effect chains. In *35th Euromicro Conference on Real-Time Systems (ECRTS '23).* Alessandro V. Papadopoulos (Ed.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 262, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Dagstuhl, Article 10, 1–22. DOI: https://doi.org/10.4230/LIPIcs.ECRTS.2023.10

[25] Zhishan Guo, Sudharsan Vaidhun, Abdullah Al Arafat, Nan Guan, and Kecheng Yang. 2023. Stealing static slack via WCRT and sporadic p-servers in deadline-driven scheduling. In *2023 IEEE Real-Time Systems Symposium (RTSS)*, 40–52. DOI: https://doi.org/10.1109/RTSS59052.2023.00014

[26] Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. 2021. Timing analysis of asynchronized distributed cause-effect chains. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 40–52. DOI: https://doi.org/10.1109/RTAS52030.2021.00012

[27] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. 2018. A design-space exploration for allocating security tasks in multicore real-time systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 225–230. DOI: https://doi.org/10.23919/DATE.2018.8342007

[28] Andrea Höller, Tobias Rauter, Johannes Iber, and Christian Kreiner. 2015. Diverse compiling for microprocessor fault detection in temporal redundant systems. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, 1928–1935. DOI: https://doi.org/10.1109/CIT/IUCC/DASC/PICOM.2015.285

[29] Gregg Keizer. 2010. Is Stuxnet the 'Best' Malware Ever? Retrieved from http://www.infoworld.com/article/2626009/malware/is-stuxnet-the--best--malware-ever-.html

[30] J. Kim, G. Park, H. Shim, and Y. Eun. 2016. Zero-stealthy attack for sampled data control systems: The case of faster actuation than sensing. In *IEEE Conference on Decision and Control (CDC)*, 5956–5961.

[31] K. H. Kim, Jing Qian, Zhen Zhang, Qian Zhou, Kyung-Deok Moon, Jun-Hee Park, Kwang-Roh Park, and Doo-Hyun Kim. 2010. A scheme for reliable real-time messaging with bounded delays. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 18–27. DOI: https://doi.org/10.1109/ISORC.2010.45

[32] Leonie Köhler, Phil Hertha, Matthias Beckert, Alex Bendrick, and Rolf Ernst. 2023. Robust cause-effect chains with bounded execution time and system-level logical execution time. *ACM Transactions on Embedded Computing Systems* 22, 3, Article 50 (Apr. 2023), 28 pages. DOI: https://doi.org/10.1145/3573388

[33] H. Kopetz and G. Grunsteidl. 1993. TTP—A time-triggered protocol for fault-tolerant real-time systems. In *23rd International Symposium on Fault-Tolerant Computing (FTCS '23)*, 524–533. DOI: https://doi.org/10.1109/FTCS.1993.627355

[34] D. Kozhaya, J. Decouchant, and P. Esteves-Verissimo. 2019. RT-ByzCast: Byzantine-resilient real-time reliable broadcast. *IEEE Transactions on Computers* 68, 03 (Mar. 2019), 440–454. DOI: https://doi.org/10.1109/TC.2018.2871443

[35] David Kozhaya, Jérémie Decouchant, Vincent Rahli, and Paulo Esteves-Verissimo. 2021. PISTIS: An event-triggered real-time byzantine-resilient protocol suite. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2277–2290. DOI: https://doi.org/10.1109/TPDS.2021.3056718

[36] Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. 2014. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In *22nd International Conference on Real-Time Networks and Systems (RTNS '14)*. ACM, New York, NY, 139–148. DOI: https://doi.org/10.1145/2659787.2659799

[37] Kristin Krüger, Marcus Völp, and Gerhard Fohler. 2018. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS '18)*. Sebastian Altmeyer (Ed.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 106, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Article 22, 1–17. DOI: https://doi.org/10.4230/LIPIcs.ECRTS.2018.22

[38] Kristin Krüger, Nils Vreman, Richard Pates, Martina Maggio, Marcus Völp, and Gerhard Fohler. 2021. Randomization as mitigation of directed timing inference based attacks on time-triggered real-time systems with task replication. *Leibniz Transactions on Embedded Systems* 7, Article 1 (Aug. 2021), 1–29. DOI: https://doi.org/10.4230/LITES.7.1.1

[39] Robert M. Lee, Michael J. Assante, and Tim Conway. 2016. Analysis of the Cyber Attack on the Ukrainian Power Grid. E-ISAC. Retrieved from https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf

[40] Haoran Li, Chenyang Lu, and Christopher D. Gill. 2021. RT-ZooKeeper: Taming the recovery latency of a coordination service. *ACM Transactions on Embedded Computing Systems* 20, 5s, Article 103 (Sept. 2021), 22 pages. DOI: https://doi.org/10.1145/3477034

[41] Jeanne Meserve. 2007. Mouse Click Could Plunge City into Darkness, Experts Say. Retrieved March 12, 2017 from http://edition.cnn.com/2007/US/09/27/power.at.risk/index.html

[42] Sparsh Mittal. 2017. A survey of techniques for cache partitioning in multicore processors. *ACM Computing Surveys* 50, 2, Article 27 (May 2017), 39 pages. DOI: https://doi.org/10.1145/3062394

[43] Amin Naghavi and Nicolas Navet. 2025. Total execution order in fault-tolerant real-time systems. In *32nd International Conference on Real-Time Networks and Systems (RTNS '24)*. ACM, New York, NY, 12–24. DOI: https://doi.org/10.1145/3696355.3699704

[44] Amin Naghavi, Sepideh Safari, and Shaahin Hessabi. 2021. Tolerating permanent faults with low-energy overhead in multicore mixed-criticality systems. *IEEE Transactions on Emerging Topics in Computing* 10, 2 (2021), 985–996. DOI: https://doi.org/10.1109/TETC.2021.3059724

[45] Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M. Gerdes. 2019. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 103–116. DOI: https://doi.org/10.1109/RTAS.2019.00017

[46] Risat Mahmud Pathan. 2014. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems* 50 (2014), 509–547. DOI: https://doi.org/10.1016/j.suscom.2020.100474

[47] P. E. Veríssimo, N. F. Neves, and M. P. Correia. 2003. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems.* R. Lemos, C. Gacek, and A. Romanovsky (Eds.).Lecture Notes in Computer Science, Vol. 2677. Springer, Berlin. DOI: https://doi.org/10.1007/3-540-45177-3_1

[48] M. Pease, R. Shostak, and L. Lamport. 1980. Reaching agreement in the presence of faults. *Journal of the ACM* 27, 2 (April 1980), 228–234. DOI: https://doi.org/10.1145/322186.322188

[49] Linh T. X. Phan, Meng Xu, Jaewoo Lee, Insup Lee, and Oleg Sokolsky. 2013. Overhead-aware compositional analysis of real-time systems. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 237–246. DOI: https://doi.org/10.1109/RTAS.2013.6531096

[50] S. Poledna, A. Burns, A. Wellings, and P. Barrett. 2000. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transactions on Computers* 49, 2 (2000), 100–111. DOI: https://doi.org/10.1109/12.833107

[51] Riccardo Pucella and Fred B. Schneider. 2010. Independence from obfuscation: A semantic framework for diversity. *Journal of Computer Security* 18, 5 (2010), 701–749.

[52] Federico Reghenzani, Zhishan Guo, and William Fornaciari. 2023. Software fault tolerance in real-time systems: Identifying the future research questions. *ACM Computing Surveys* 55, 14s, Article 306 (July 2023), 30 pages. DOI: https://doi.org/10.1145/3589950

[53] Luís Rodrigues, Paulo Veríssimo, and Antonio Casimiro. 1995. Priority-based totally ordered multicast. In *3rd IFIP/IFAC workshop on Algorithms and Architectures for Real-Time Control (AARTC '95)*.

[54] Edo Roth and Andreas Haeberlen. 2021. Do not overpay for fault tolerance! In *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '21)*. DOI: https://doi.org/10.1109/RTAS52030.2021.00037

[55] FredB. Schneider.1993. Replication management using the state machine approach. In *Distributed Systems* (2nd Ed.). ACM Press/Addison-Wesley Publishing Co., USA, 169–197.

[56] Soham Sinha, Richard West, and Ahmad Golchin. 2020. PAStime: Progress-aware scheduling for time-critical computing. In *32nd Euromicro Conference on Real-Time Systems (ECRTS '20)*. Marcus Völp (Ed.), Leibniz International

Proceedings in Informatics (LIPIcs), Vol. 165, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Dagstuhl, Article 3, 1–24. DOI: https://doi.org/10.4230/LIPIcs.ECRTS.2020.3

[57] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers* 62, 1 (2013), 16–30. DOI: https://doi.org/10.1109/TC.2011.221

[58] Yun Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin. 2002. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers* 51, 8 (2002), 900–915. DOI: https://doi.org/10.1109/TC.2002.1024738

[59] Yi-wen Zhang and Rui-feng Guo. 2013. Power-aware scheduling algorithms for sporadic tasks in real-time systems. *Journal of Systems and Software* 86, 10 (2013), 2611–2619. DOI: https://doi.org/10.1016/j.jss.2013.04.075

[60] Jia Xu. 2010. A method for adjusting the periods of periodic processes to reduce the least common multiple of the period lengths in real-time embedded systems. In *2010 IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, 288–294. DOI: https://doi.org/10.1109/MESA.2010.5552058

[61] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. 2009. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 351–360.

[62] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. 2010. Feasibility analysis under fixed priority scheduling with fixed preemption points. In *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, 71–80. DOI: https://doi.org/10.1109/RTCSA.2010.40

[63] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. 2016. TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 1–12. DOI: https://doi.org/10.1109/RTAS.2016.7461362

[64] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 155–166. DOI: https://doi.org/10.1109/RTAS.2014.6925999

[65] Lin Zhang, Kaustubh Sridhar, Mengyu Liu, Pengyuan Lu, Xin Chen, Fanxin Kong, Oleg Sokolsky, and Insup Lee .2023. Real-time data-predictive attack-recovery for complex cyber-physical systems. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 209–222. DOI: https://doi.org/10.1109/RTAS58335.2023.00024

[66] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 633–649. https://www.usenix.org/conference/osdi20/presentation/zhang-yunhao

[67] H. Zou and F. Jahanian. 1998. Real-time primary-backup (RTPB) replication with temporal consistency guarantees. *In Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No. 98CB36183)*, 48–56. DOI: https://doi.org/10.1109/ICDCS.1998.679486

Appendices

A Complexity and Optimization

To facilitate presentation, the Algorithms 1–4 represent a simplified version of our implemented method. The implemented schedule algorithm has linear time complexity with respect to the number of imminent tasks along with the maximum number of chunks within a task. The release algorithm in addition to imminent tasks and the maximum number of chunks within a task, examines the jobs in the ready queue to identify the MAP. Despite this additional step, its time complexity remains bounded by the number of tasks (n) plus the maximum number of chunks within a task. Regarding the update algorithm, it operates with an overhead of $O(m)$, where m denotes the number of nodes. Throughout our implementation, we have thoroughly investigated several optimizations, which we will detail next.

Release and Schedule. In Algorithms 1 and 3, maintaining a continuous update of the WCP for the tail of the chunk queue eliminates the need for calculations performed at lines 7 and 11 respectively. The insertion of multiple chunks of a job into the chunk queue in Algorithm 2 is an $O(1)$ operation, where only a single record is added to the queue with each insertion. This record includes a pointer to the job's specifications, an index indicating the first chunk of the job, and the count of subsequent chunks of the job moved to the chunk queue. Therefore, each element in a chunk queue includes multiple chunks of the same job bundled together. Additionally, precalculating and storing the accumulated WCETs up to each task's milestone as $C_a^{[1:k]} = \sum_{l=1}^{k} C_a^l$ enables us to bypass calculating the summation at line 8 in Algorithm 2. Furthermore, a simple comparison between $W_{tail} + (C_a - C_a^{[1:next_a - 1]})$ and $SlackBound$ enables us to decide if a job τ_a in the head of ready queue can be moved to CQ. Therefore, only if by this comparison the algorithm fails to move all chunks of the job, the maximum at line 6 of Algorithm 2 will be applied to determine the eligible chunks from τ_a for moving to the chunk queue. This maximum can be calculated through either a linear or binary search, extending between the milestones $l_{min} = \lfloor SlackBound/max(C_i^l) \rfloor$ and $l_{max} = min(\lfloor SlackBound/min(C_i^l) \rfloor, L_i)$ (where $min(C_i^l)$ and $max(C_i^l)$ denote the lengths of the smallest and largest chunks of τ_i respectively).

In Algorithm 3, when calculating *SlackBound* at line 13, the algorithm can store and reuse the *SlackBound* calculated previously in the prior schedule (similar to line 16 in Algorithm 1). The only difference is that the *SlackBound* value in the previous schedule at time t^{prev} might be pessimistic for the current schedule at time t ($t^{prev} < t$). Therefore, the node still sometimes needs to recalculate the *SlackBound* of imminent tasks with higher priority than the previously executed job (considering their next release based on the current time) when adding the WCET of the next chunk of the job at the head of the ready queue to the WCP of $prog_k^{tail}$ exceeds *SlackBound*.

Update. In Algorithm 4, by sequentially inserting received progress updates into *PROG* in ascending order, we can circumvent the need for calculating the minimum within the loop at line 8.

When a bundle of chunks is executed, they are removed from the chunk queue. Each node tracks its progress by recording the number of chunks it has completed. Additionally, it stores the cumulative WCETs of the chunks for each recent progress update in a ring buffer. The cumulative WCET represents the summation of the WCETs of all chunks that have been executed. The size of this ring buffer must be sufficient to store the cumulative WCETs of all chunks executed since *minProg*. This ensures constant time ($O(1)$) computation of the sum at line 10.

B Ensuring Timeliness in a Coarsely Synchronized System

Given the impact of coarsely synchronized clocks on our algorithms, the following lemma explains how incorporating late event arrivals into the schedulability test allows our method to maintain the timeliness of tasks.

Lemma 9. *If the WCET of each chunk is chosen sufficiently large such that no chunk exceeds its WCET on any node, and the task set remains schedulable when Δ is treated as release jitter in the schedulability test, then the coarse synchronization and release delay, along with the corresponding changes in our algorithm, will not result in deadline misses:*

(1) *Delayed execution of the release algorithm and accounting for Δ when adjusting t_{update} at the release of a task.*
Proof. In systems where a total order protocol is not enforced, accounting for release delays as release jitter in schedulability tests ensures that tasks do not miss their deadlines due to such delays. In our proposed method, to establish a total order, we defer the insertion point of tasks into the ready queue by moving chunks of jobs from the ready queue to the chunk queue until the MAP is reached. The MAP is identified when either no jobs remain in the ready queue or the slack of the released task would be fully utilized by the chunks in the chunk queue on the back-runner, assuming the back-runner executes them at their WCETs. Considering release jitter in the schedulability test reduces the available slack for tasks, which in turn decreases the time for which our method delays a job's execution. Additionally, by setting t_{update} to $r_e + \Delta$, we ensure that the release delay is accounted for in the WCP, thereby

maintaining the WCP as an upper bound on the worst-case behavior of the back-runner. Therefore, since the effects of coarse synchronization are considered in both WCP and slack, and Equation (2) ensures that the WCP of any progress prior to MAP does not exceed the slack of imminent tasks, the priority inversion on the back-runner always remains within these slack bounds, guaranteeing that deadlines are met. □

(2) *Additional idle time before executing a chunk when considering earlier release times for imminent tasks in the scheduling algorithm.*
Proof. The system may experience a slight decrease in performance regarding response times when the scheduling algorithm considers the next release of imminent tasks based on $t - \Delta$, as this imposes additional idle time. However, this algorithm always moves the first chunk from the ready queue to the chunk queue and executes it as soon as the WCP of the node's progress converges with the node's local time, regardless of the next release time of imminent tasks. This convergence occurs when a node finishes chunks close to their WCETs. Thus, a node never idles unless there is dynamic slack from completing chunks earlier than their WCETs (as mentioned in Section 6.2), ensuring that the imposed idle time never causes deadline misses. □

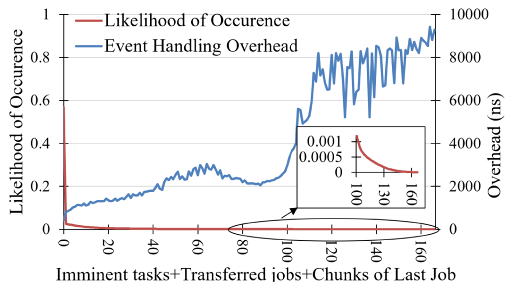
(3) *Delayed execution of the update algorithm by Δ, with Δ explicitly accounted for when setting t_{update}.*
Proof. The purpose of the update algorithm is to indicate that the healthy back-runner made progress earlier than the time specified by the WCP of that progress. Delaying the execution of the update algorithm reduces the efficiency of our method in terms of response times (as discussed in Section 8.4). This is because it increases the t_{update} for the progress of the healthy back-runner *minProg*, which in turn increases the WCP for each progress and limits the number of chunks that can be moved to the chunk queue. However, the algorithm ensures that t_{update} is always set to the minimum of the update time and the WCP of the healthy back-runner's progress, calculated using the previous *minProg* and t_{update}. While postponing the execution of the release algorithm may delay the update time to $r_e + Timeout + \Delta$, the t_{update} for the progress of the healthy back-runner (*minProg*) remains upper-bounded by the WCP of this progress, calculated using the previous *minProg* and t_{update}. As a result, a delayed update never produces WCP values higher than those calculated without the update, and the WCP continues to provide an upper bound on the time the back-runner finishes a progress. Consequently, deadlines are still guaranteed. □

C Supplementary Results

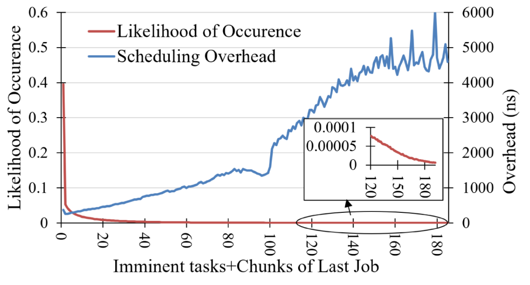
C.1 Overhead

The overhead of our method was measured on a Raspberry Pi using the setup described in Section 8.1, but with 3 nodes, each executing 500 task sets at a total utilization of 0.9 and 100,000 jobs per task set.

Figure C1(a) illustrates the runtime costs of executing the release algorithm and the probability of encountering such overhead. Figure C1(b) presents similar results for the scheduling algorithm. The overhead values shown in Figure C1(a) and (b) include all overheads, such as those from data structures, but exclude message transmission and context switching overheads. The overhead of the update algorithm is negligible and has not been depicted. To mitigate the impact of operating system interference on our results, we focused on the 99th percentile of measured overheads for measurement points that occurred at least 1,000 times (out of a total of at least 150 million executions).



(a) Overhead when releasing a task.



(b) Scheduling overhead

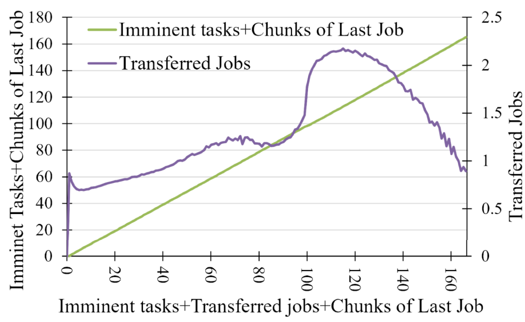
Fig. C1. The overhead of our release (left) and scheduling (right) algorithms under different runtime conditions, and the frequency of occurrence of each condition.

The overhead for inserting tasks (Algorithm 1) is mainly determined by the total count of the following factors: the tasks explored in the set of imminent tasks, the jobs transferred from the ready queue to the chunk queue, and the chunks moved to the chunk queue from the last job that the algorithm attempted to remove from the ready queue.

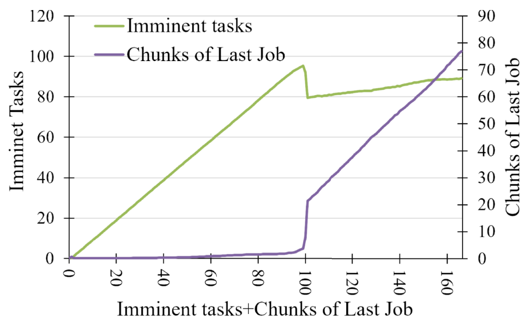
Figure C1(a) illustrates the observed overhead and the corresponding likelihood of occurrence for each sum of the mentioned values. As can be seen, most of the time, the release overhead is negligible (with 98.5% of occurrences experiencing overhead of less than 3.35 μs), and even the maximum observed overhead remains below 9.43 μs.

The scheduling algorithm (Algorithm 3) moves chunks to *CQ* only from the job in the head of the ready queue. Figure C1(b) illustrates the scheduling overhead when chunks are moved from the ready queue to the chunk queue (otherwise, the overhead is negligible). We observed a maximum of 6.04 μs scheduling overhead with the 99.6% of occurrences being less than 1.59 μs.

The graph in Figure C2 helps explain the trends observed in the overhead graphs (Figure C1(a) and (b)). Figure C2(a) shows how the number of jobs transferred from the ready queue to the chunk queue in the release algorithm affects the x-axis summation in Figure C1(a). This variation in transferred jobs contributes to the wave-like pattern observed in the overhead of the release algorithm. This is because the different parameters in the summation influence the overhead results to different degrees.

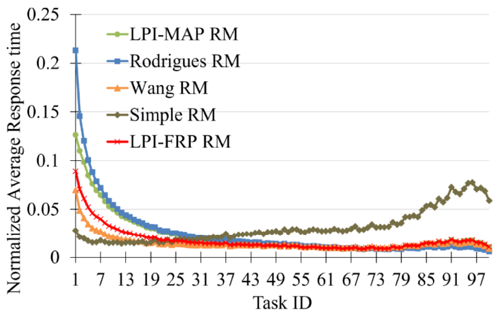


(a) Release algorithm



(b) Schedule algorithm

Fig. C2. The relation between different factors affecting the overhead of the release (left) and the scheduling (right) algorithms.



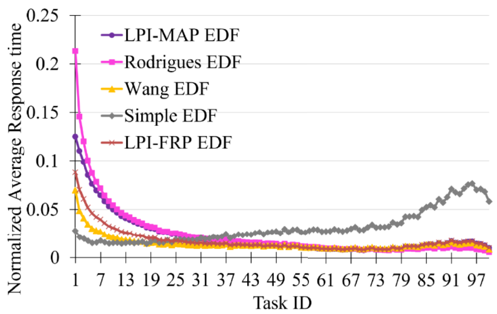
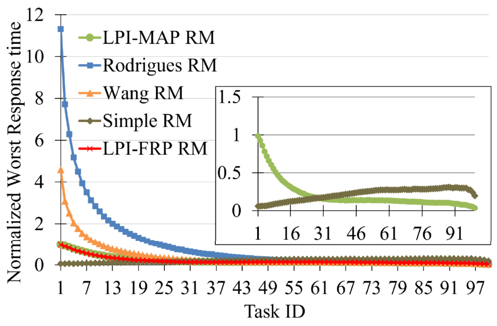


Fig. C3. The normalized average response time of individual tasks averaged on the tasks with the same priority on 100 task sets scheduled using RM (left) and EDF (right).



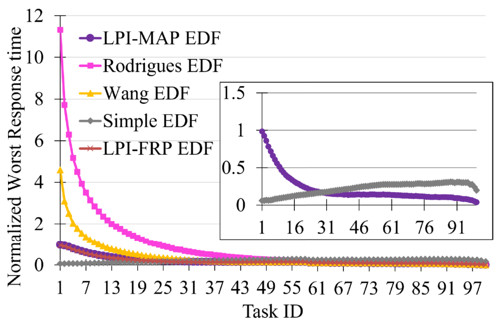


Fig. C4. The normalized worst response time of individual tasks averaged on the tasks with the same priority on 100 task sets scheduled using RM (left) and EDF (right).

Furthermore, both the imminent tasks and the chunks of the final job being transferred impact the overhead trend of the scheduling algorithm. Figure C2(b) illustrates how these factors influence the x-axis summation in Figure C1(b). The trend depicted in Figure C1(b) is a result of the influence of each of the factors shown in Figure C2(b) on the overhead.

It is important to note that the values in Figure C2(b) were observed during execution and are reflective of task specifications such as the WCET of benchmark tasks and the size of chunks.

C.2 Task Response Time

Assuming that tasks have been sorted by their minimum inter-arrival time, where smaller task indices correspond to shorter minimum inter-arrival times (and relative deadlines), Figures C3 and C4 illustrate for RM and EDF, the average and worst response time (normalized by task deadlines) of tasks with the same indices averaged over 100 task sets in the normal scenario. In these figures, each task set has a total utilization of 0.9 and comprises 100 tasks generated based on the settings mentioned in Section 8.1.

As depicted in these figures, our LPI-MAP method generally improves the response time for the majority of tasks compared to the Simple method, albeit with a slight increase in response time for a minority of tasks with shorter minimum inter-arrival times. This on average leads to better average response time for tasks (illustrated in Figure 4(a) in Section 8.3), as our method benefits from communication to reduce the idle time needed for ensuring total order while guaranteeing deadlines.

Figure C4 demonstrates that the approach proposed by Rodrigues and Wang results in a significant increase in the worst response time of tasks with shorter minimum inter-arrival times. This leads to the worst response time of tasks, with the shortest minimum inter-arrival time being almost 12 times for Rodrigues and 4 times for Wang more than their deadlines. This shows how priority inversion can cause deadline misses in tasks with shorter minimum inter-arrival times in Rodrigues and Wang. The worst-case response time of our method, as well as the modified LPI-FRP method by Naghavi and Navet [43], remained within the task deadlines (i.e., the normalized worst response times of tasks are below 1), meaning neither method missed any task deadlines.

Received 16 September 2024; revised 12 March 2025; accepted 13 August 2025