Overrun-Resilient Multiprocessor Real-Time Locking

Zelin Tong
University of North Carolina at Chapel Hill, USA

Shareef Ahmed
University of North Carolina at Chapel Hill, USA

James H. Anderson
University of North Carolina at Chapel Hill, USA

Abstract

Existing real-time locking protocols require accurate worst-case execution time (WCET) estimates for both tasks and critical sections (CSs) in order to function correctly. On multicore platforms, however, the only seemingly viable strategy for obtaining such estimates is via measurements, which cannot produce a true WCET with certainty. The absence of correct WCETs can be partially ameliorated by enforcing *execution budgets* at both the task and CS levels and by using a locking protocol that is *resilient to budget overruns*, *i.e.*, that ensures that the schedulability of non-overrunning tasks is not compromised by tasks that do overrun their budgets. Unfortunately, no fully overrun-resilient locking protocol has been proposed to date for multiprocessor systems. To remedy this situation, this paper presents two such protocols, the OR-FMLP and the OR-OMLP, which introduce overrun-resiliency mechanisms to two existing multiprocessor protocols, the spin-based FMLP and suspension-based global OMLP, respectively. In devising such mechanisms, undo code can be problematic. For the important locking use case of protecting shared data structures, it is shown that such code can be avoided entirely by using *abortable critical sections*, a concept proposed herein that leverages obstruction-free synchronization techniques. Experiments are presented that demonstrate both the effectiveness of the mechanisms introduced in this paper and their cost.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases Real-Time Systems, Real-Time Synchronization, Budget Enforcement

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.9

Funding Supported by NSF grants CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

1 Introduction

Many safety-critical systems require *real-time safety certification* that hinges on both timing analysis and schedulability analysis. The goal of *timing analysis* is to produce *worst-case execution times* (WCETs) for executable code. *Schedulability analysis* then determines whether a system's timing constraints are met, assuming valid WCETs are provided. Due to the advent of multicore technologies, work on timing and schedulability analysis has largely focused on the multiprocessor case in recent years [11, 17, 34].

A troubling disconnect. In the multiprocessor case, a largely unnoticed fundamental disconnect exists when using timing- and schedulability-analysis together to validate real-time correctness. There is consensus today that static timing-analysis tools may never be a practical reality for multicore machines due to the highly complex nature of multicore architectures [43]. The only alternative is to use *measurement-based* timing analysis, a topic that has been the focus of considerable recent work [16, 18]. With measurement-based timing analysis, however, one can never be certain that the true WCET of a piece of code is ever

captured (hereafter, we assume "WCET" means "true WCET"). Thus, it is necessary to distinguish between the WCET of a piece of code and its *provisioned execution time* (PET) as obtained via measurements and assumed in schedulability analysis. Note that WCETs are generally unknowable, and a PET may likely be less than the corresponding WCET.

Mitigating this disconnect. Any safety risk introduced by assuming such PETs can be avoided by instead using *(guaranteed) execution-time (upper) bounds* (ETBs) obtained under unrealistically pessimistic conditions. For example, any execution speedups due to caches and pipelining might be defined away, bus contention might be over-approximated, *etc.* While such ETBs might be safe to use as PETs, this would likely be impractical on a multicore machine, as an ETB could easily be many times larger than the corresponding WCET. In fact, system-wide pessimism could be high enough to negate the processing capacity of all "additional" cores [32]. Thus, more reasonable measurement-based PETs are inevitable.

A system provisioned assuming such PETs can have *correct tasks* whose PETs are at least their WCETs, and *faulty tasks* whose PETs are less than their WCETs. Hopefully, PET overruns (faults) should be rare. Moreover, when they do occur, they should be *contained*. The usual approach here is to treat PETs as *execution-time budgets* that are enforced by the operating system (OS). Such an approach can ensure the following desirable property.

P1 The response times of correct tasks, as derived using PETs, are not increased by a PET overrun of a faulty task.

Task-level budgeting is not enough. Unfortunately, due to various realities of real systems, task-level budget management alone is an incomplete solution to the timing/schedulability disconnect. This paper is directed at providing a deep look at one such reality: *the need to support locking protocols for arbitrating accesses to shared resources*. In this setting, we actually care about various different PETs, WCETs, and ETBs. For example, in addition to task-level PETs, PETs are needed for individual critical sections (CSs), and various locking-protocol and OS code sequences. To avoid confusion, we will add a qualifying prefix when referring to non-task-level terms—*e.g.*, "CS PET" refers to the PET of a CS, while "PET" (without qualification) refers to a task-level PET. When locking protocols are introduced, CS PETs (and also various protocol- and OS-related PETs) are used to determine blocking times when tasks access shared resources. *Incorrect blocking-time estimates due to inaccurate PETs can completely compromise schedulability guarantees.*

Overrun-resilient locking protocols. To address this issue, we propose in this paper the notion of an *overrun-resilient locking protocol*. In addition to not causing a violation of P1, such a protocol must also uphold its CS-level variant:

P2 The response times of correct tasks, as derived using PETs, are not increased by a CS PET overrun of a faulty task.

It's not so easy. The obvious solution to satisfying P1 and P2 is to assign budgets to both tasks and CSs as given by their respective PETs. The main new complication that arises when doing this is the need to abort the CS of some task when one of these budgets is overrun. Such aborts should be avoided if possible, but they cannot be entirely precluded. For example, a CS budget overrun will necessarily cause a CS abort. The usual approach to aborting work is to execute undo code. Presumably, a PET would have to be associated with such code. What happens if the abort code overruns its PET? Additionally, certain code sequences exist pertaining to lock and budget management for which overruns are similarly problematic. For example, when a CS is aborted, the unlock logic must execute to free the resource. What if the PET associated with this unlock code is overrun? It is not clear how these perplexing "chicken and egg" problems can be addressed. Whatever the solution, an

Table 1 Properties satisfied by prior work. ("NC" means the work does not consider how to satisfy the specified property. As explained in Sec. 9, some of these "NC" entries can be changed to "Y" at the expense of very pessimistic provisioning assumptions.)

Protocol	Multi-processor	P1	P2	P3	Protocol	Multi-processor	P1	P2	P3
ICSs [30]	N	Y	Y	Y	FMLP [9]	Y	N	N	N
RRP [3]	N	Y	Y	NC	M-BWI [20]	Y	N	N	NC
RACPwP [39]	N	Y	Y	NC	vMPCP [31]	Y	Y	N	NC
SIRAP [6]	N	Y	N	NC	M-BROE [8]	Y	Y	N	NC
OMLP [12]	Y	N	N	N	This Work	Y	Y	Y	Y

overrun-resilient locking protocol must uphold a third property:

P3 The response times of correct tasks, as derived using PETs, are not increased by the budget-enforcement mechanism.

Related work. Various locking protocols have been proposed in prior work that considers budget overruns. However, no prior work focusing on multiprocessors fully considers properties P1–P3. Relevant prior work is summarized in Tbl. 1 and discussed in detail in Sec. 9.

Contributions. In this paper, we present overrun-resilient multiprocessor locking protocols that satisfy P1–P3. Our contributions are fourfold. First, we introduce the *overrun-resilient flexible multiprocessor locking protocol* (OR-FMLP), an overrun-resilient extension of the spin-based FMLP [9]. Second, we introduce the *overrun-resilient global optimal multiprocessor locking protocol* (OR-OMLP), an overrun-resilient extension of the suspension-based global OMLP [12]. Third, for the important locking use case of coordinating accesses to shared data structures, we propose the concept of an *abortable CS*, which facilitates satisfying P2 and P3. Finally, we present the results of an experimental evaluation of the cost of overrun-resilient locking and its isolation benefits with respect to timing faults.

Both the OR-FMLP and OR-OMLP use a concept called a "forbidden zone" [28] to satisfy P1. A *forbidden zone* (FZ) is a length of time at the end of a job's task budget during which any lock request will be denied. However, the application of this concept is very different in the two protocols. To circumvent the various chicken-and-egg problems related to P3, ETBs must be used for certain code sequences. As ETBs can be very pessimistic, reliance on them should be minimized. With this in mind, we carefully sift through the various design choices and conclude that in a spinlock like the OR-FMLP, coarse-grained FZs should be used that include both CS execution time and blocking time, while in a suspension-based lock like the OR-OMLP, fine-grained FZs based on CS execution times only are better.

Our notion of an abortable CS requires no undo code when aborting CSs. An abortable CS uses word-based *obstruction-free* [25] software transactional memory (STM) techniques to linearize a CS to a single write instruction. Obstruction-freedom is a type of non-blocking synchronization that must be used with a contention manager to ensure progress under contention. In our case, the contention manager is a locking protocol. We show that using such a strong contention manager enables significant simplifications in obstruction-free code.

Organization. In the rest of this paper, we provide necessary background information (Sec. 2), delve further into task and CS budget management (Sec. 3), present the OR-FMLP, the OR-OMLP, and the abortable CS concept (Secs. 4–6), present our experimental results (Sec. 7), discuss certain practical implications of our work (Sec. 8), review related work (Sec. 9), and conclude (Sec. 10).

2 System Model and Background

Task model. We consider a system of n implicit-deadline[1] sporadic *tasks* $\tau_1, \tau_2, \ldots, \tau_n$ to be scheduled on m identical processors by a global job-level fixed-priority scheduler; we assume global earliest-deadline-first (G-EDF) scheduling, unless stated otherwise. Each task τ_i releases a potentially infinite sequence of *jobs* $J_{i,1}J_{i,2}\ldots$ (we omit the job index if it is irrelevant). Each task τ_i has a *period* T_i specifying the minimum spacing between consecutive job releases. Each task has a PET obtained via measurement-based timing analysis.

Resource model. We consider a system that has a set $\{\ell_1, \ldots, \ell_{n_r}\}$ of serially reusable shared resources. To ensure mutually exclusive resource access, a *locking protocol* must be employed. When a job J_i requires a resource ℓ_k, it *issues* a *request* $\mathcal{R}$ for ℓ_k. $\mathcal{R}$ is *satisfied* as soon as J_i *holds* ℓ_k, and *completes* when J_i *releases* ℓ_k. $\mathcal{R}$ is *active* from its issuance to its completion. J_i must wait until $\mathcal{R}$ can be satisfied if it is held by another job. It may do so either by *busy-waiting* (or *spinning*) in a tight loop, thereby wasting processor time, or by being *suspended* by the OS until $\mathcal{R}$ is satisfied. A resource access is called a *critical section* (CS). Each CS has a CS PET obtained via measurement-based timing analysis. We consider non-nested resource requests only. We let Γ_k to denote the set of tasks that share ℓ_k.

Priority inversions. *Priority-inversion blocking* (or *pi-blocking*) occurs when a job is delayed and this delay cannot be attributed to higher-priority demand for processing time. Under a given real-time locking protocol, a job may experience pi-blocking each time it requests a resource—this is called *request blocking*. In addition, a preemptive ready job may experience pi-blocking due to the non-preemptive execution of lower-priority jobs—this is called *non-preemptive blocking*. On multiprocessors, the formal definition of pi-blocking actually depends on how schedulability analysis is done. For example, of relevance to suspension-based locks, analysis may be either *suspension-oblivious* (*s-oblivious*) or *suspension-aware* (*s-aware*) [12]. Under s-oblivious analysis, suspension time is *analytically* treated as computation time.

FMLP. Under the FMLP [9], non-preemptive spin locks are used to ensure mutually exclusive resource access.[2] Each job that is blocked on a resource busy-waits within a FIFO queue.

Global OMLP. The global OMLP [12] is a suspension-based locking protocol that has asymptotically optimal pi-blocking under s-oblivious analysis. The global OMLP ensures $O(m)$ pi-blocking by utilizing a dual-queue structure, with an m-element FIFO queue fed into by a priority queue, as shown in Fig. 1. A new request is enqueued in the FIFO queue (resp., priority queue) if there are fewer than (resp., at least) m active requests. When the request at the head of the FIFO queue (*i.e.*, the resource holder) completes, it is dequeued, the next request (if any) in the FIFO queue becomes satisfied, and the highest-priority request (if any) in the priority queue is moved to the tail of the FIFO queue.

▶ **Example 1.** Fig. 2 shows five jobs that issue requests to the global OMLP with $m = 3$. Fig. 1 shows the global OMLP queues at time 3.5, where J_i's request is denoted by $\mathcal{R}_i$. The first three issued requests are enqueued directly in the FIFO queue. Thus, $\mathcal{R}_4$ is satisfied before $\mathcal{R}_3$, although J_4 has lower priority than J_3. Since the FIFO queue is full, $\mathcal{R}_5$ and $\mathcal{R}_3$ are enqueued in the priority queue upon issuance. When $\mathcal{R}_1$ completes at time 6, $\mathcal{R}_2$

[1] The results of this paper do not depend on the choice of deadline constraints. Implicit deadlines are assumed for simplicity.

[2] There are actually two FMLP variants: short (spin-based) and long (suspension-based). We are considering the short variant here.

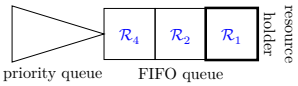


Figure 1 The global OMLP queue structure for 3 processors.

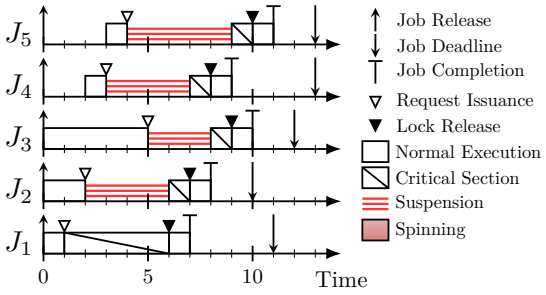


Figure 2 Jobs issuing requests to the global OMLP with $m = 3$. (The notation in this figure is also used in subsequent figures.)

becomes satisfied, and $\mathcal{R}_3$ is moved from the priority queue to the FIFO queue, as J_3 has higher priority than J_5. Thus, $\mathcal{R}_3$ is satisfied before $\mathcal{R}_5$, despite being issued later.

For ease of notation, we henceforth assume that all jobs of each task include one request for the same resource, and this request is preceded and followed by non-resource-accessing code. This assumption enables us to refer to a job's CS without ambiguity. We stress that we are making this assumption only for simplicity; none of our results actually depend on it.

3 Budget Management

Execution budgets that are enforced at runtime are obtained by inflating *base budgets* that pertain to the execution of task code by adding certain overheads. Additional overheads are then added to obtain *analytical budgets* that are used in schedulability analysis. In this section, we provide details concerning these budgeting notions and relevant overheads.

Base budgets. We define the *base task budget* (resp., *base CS budget*) of a task τ_i (resp., τ_i's CS), denoted by C_i^b (resp., L_i^b), as its PET (resp., CS PET).

What is and is not included in base budgets. Timing analysis is applied to determine relevant PETs for a task independently of the task system that contains it. As lock-related *blocking times* are system-dependent, we assume that they are not included in base task budgets. The *lock/unlock logic* of a suspension-based lock is executed in the OS and hence would not be included in base task budgets. In contrast, for a spinlock, this logic executes at user level. However, as seen later, to satisfy P3, we must take special care in dealing with this logic, so we assume it is not included in base task budgets.

When measurement-based timing analysis is applied, preemptions are notoriously difficult to deal with due to difficulties in predicting cache interactions. For this reason, we assume that *CSs are executed non-preemptively* and that their base budgets are determined assuming cold caches. However, we do not preclude preemptions outside of CSs, as long as base task budgets include *cache-related preemption and migration delay (CPMD)*, which is a cost that is incurred by a job to re-establish lost cache affinity after a preemption or migration. As the focus of this paper is not timing analysis, determining valid CPMDs is out of scope.

Timers. To enforce base task and CS budgets, we require the usage of timers supported by the OS, which we call *task-* and *CS-budgeting timers*, respectively. Such a timer *starts* when the relevant entity (an entire job or a CS) starts executing and *stops* when the entity is preempted (not allowed for CSs), aborted (see below), or completes execution. Between starting and stopping, a timer is *active*.

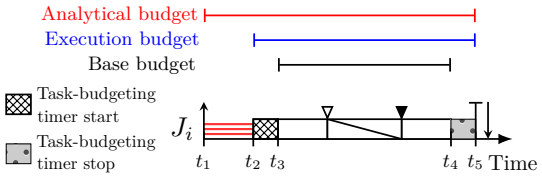


Figure 3 Illustration of base, execution, and analytical budgets.

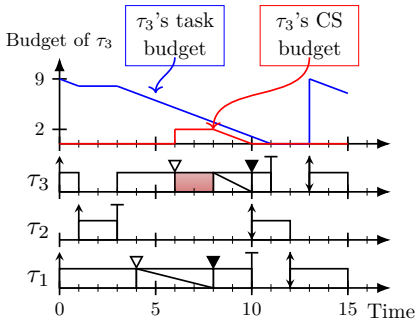


Figure 4 Budget consumption and replenishment. Overheads/delays other than spinning are omitted to avoid clutter.

Execution budgets. In reality, timers cannot be started and stopped in zero time. To start a timer, timer-handling code executes in the OS. We assume no knowledge of the exact structure of this code but do require that an ETB is specified for it. When this code executes, we know that the timer starts at some time point, but not precisely when. Stopping a timer is similar. In order to safely police base budgets, we must account for these timer activities. To do so, we instead police adjusted execution budgets as defined next.

The *task execution budget* C_i^e of a task τ_i is obtained by inflating its base task budget C_i^b by adding the worst-case cost of all task- and CS-budgeting timer overheads, as provisioned by their ETBs, that may be incurred by a job of τ_i. The *CS execution budget* L_i^e of τ_i's CS is similarly obtained by inflating its base CS budget L_i^b by adding the worst-case cost of all CS-budgeting timer overheads, as provisioned by their ETBs, associated with that CS.

In overrun-resilient locking protocols that we propose, these execution budgets are enforced at runtime. Specifically, we set the task- or CS-budgeting timer of a job to *expire* when the corresponding task or CS execution budget is exhausted. A job *overruns* its task/CS execution budget if it does not complete execution before the relevant timer expires.

▶ **Example 2.** Consider job J_i in Fig. 3. (We consider analytical budgets later.) Starting (resp., stopping) J_i's task-budgeting timer entails executing OS code during $[t_2, t_3)$ (resp., $[t_4, t_5)$). Thus, J_i's task execution budget is derived by inflating its base task budget by $(t_3 - t_2) + (t_5 - t_4)$ units.

We assume that execution budgets are managed via the following rules.

> **Consumption Rule:** A job J_i consumes its task (resp., CS) execution budget at the rate of one execution unit per unit of time when its task-budgeting (resp., CS-budgeting) timer is active.

Since a task- or CS-budgeting timer expires when the corresponding execution budget is exhausted, a job cannot consume that execution budget when it is 0.

> **Replenishment Rule:** J_i's task execution budget is set to C_i^e when it is released. J_i's CS execution budget is set to L_i^e when it issues a lock request.

▶ **Example 3.** Fig. 4 depicts three G-EDF-scheduled tasks on two processors. τ_1 and τ_3 use resource ℓ_1, which is protected by a spinlock. τ_3's task (resp., CS) execution budget is 9.0 units (resp., 2.0 units). At time 0, $J_{3,1}$ is scheduled and its task-budgeting timer starts. $J_{3,1}$ consumes 1.0 unit of its task execution budget within the time interval $[0, 1)$ during which its task-budgeting timer is active. $J_{3,1}$ is preempted by $J_{2,1}$ at time 1, causing its

Table 2 OR-FMLP and OR-OMLP overhead impact.

Overheads	Base task budgets		Task exec. budgets	
	OR-FMLP	OR-OMLP	OR-FMLP	OR-OMLP
Budgeting-timer	×	×	✓	✓
Locking & unlocking	×	×	✓	✓
Request blocking	×	×	✓	×
Non-preemptive blocking	×	×	×	×

task-budgeting timer to stop. Thus, $J_{3,1}$'s task execution budget remains the same during the time interval $[1, 3)$. At time 3, $J_{3,1}$ is scheduled again and it continues executing until completing at time 11, consuming 8.0 units of its task execution budget.

$J_{3,1}$ issues a request for ℓ_1 at time 6 that is satisfied at time 8 (when its CS-budgeting timer starts) and completes at time 10 (after consuming its entire CS execution budget).

Analytical budgets. Some overhead/delay sources do not cause task or CS execution budget to be consumed. However, such sources can impact schedulability. We define the *analytical task budget* of task τ_i, denoted C_i^a, by inflating its task execution budget to account for all overheads/delays. We define the *analytical CS budget* of τ_i's CS, denoted L_i^a, by inflating its CS execution budget to account for all overheads/delays affecting that CS.

▶ **Example 2** (Cont'd). J_i in Fig. 3 suffers pi-blocking during the time interval $[t_1, t_2)$ due to a non-preemptively executing lower-priority job. Since J_i is not scheduled during $[t_1, t_2)$, its execution budget does not decrease during this interval. However, J_i may miss its deadline due to the delay caused by this pi-blocking. Thus, J_i's analytical task budget is derived by inflating its task execution budget by $t_2 - t_1$.

Overheads/Delays. We consider the following overheads/delays that are either locking- or timer-related overheads. We summarize the overheads/delays that affect base and execution task and CS budgets under the OR-FMLP and OR-OMLP in Tbl. 2 and all introduced notation in Tbl. 3. Note that we require ETBs of these overheads to avoid introducing "chicken and egg" problems in satisfying P3, as discussed in Sec. 1.

(i) *Budgeting-timer overheads.* We denote the ETBs of starting, stopping, and expiring a budgeting-timer by Δ_{tb}, Δ_{te}, and Δ_{tt}, respectively. Since we focus on timer overheads that are due to a CS execution, accounting for overheads due to starting/stopping a task-budgeting timer for resuming/suspending a job's non-CS code is out of scope.
(ii) *Locking and unlocking overheads.* We denote the ETBs of executing the lock and unlock logic (for both spinlocks and suspension-based locks) by Δ_{lock} and Δ_{unlock}, respectively.
(iii) *Request blocking.* We let B_i denote a bound on request blocking incurred by τ_i's request.
(iv) *Non-preemptive blocking.* We let NPB_i denote a bound on non-preemptive blocking incurred by τ_i.

Accounting for these overheads in *analytical* budgets is a well-researched topic [10]. We detail the required overhead inflation in task and CS *execution* budgets under the OR-FMLP and OR-OMLP in Secs. 4 and 5, respectively.

Simplifying assumptions. In order to focus only on those overheads/delays of direct relevance to overrun-resilient locking and to simplify the description of the OR-FMLP and OR-OMLP, we make the following assumptions.

A1 ETBs of all overheads are known.
A2 The cost of aborting a CS is included in the ETB of expiring the CS-budgeting timer.

Table 3 Notation summary.

Symbol	Meaning	Symbol	Meaning
n	Number of tasks	L_i^a	Analytical CS budget of τ_i's CS
m	Number of processors	Δ_{tb}	ETB of starting overhead for a task- or CS-budgeting timer
τ_i	i^{th} task	Δ_{te}	ETB of stopping overhead for a task- or CS-budgeting timer
$J_{i,j}$	j^{th} job of τ_i	Δ_{tt}	ETB of expiring overhead for a task- or CS-budgeting timer
T_i	Period of τ_i	Δ_{lock}	ETB of locking overhead
ℓ_k	k^{th} shared resource	Δ_{unlock}	ETB of unlocking overhead
C_i^b	Base task budget of τ_i	Δ_{abort}	ETB of overhead for aborting a request
C_i^e	Task execution budget of τ_i	B_i	Maximum request blocking time of τ_i
C_i^a	Analytical task budget of τ_i	NPB_i	Maximum non-preemptive blocking time of τ_i
L_i^b	Base CS budget of τ_i's CS	Γ_k	Set of tasks that shares a resource ℓ_k
L_i^e	CS execution budget of τ_i's CS	f_i	Forbidden-zone length for J_i

A3 All overheads/delays other than task- and CS-budgeting timer overheads, locking and unlocking overheads, and request and non-preemptive blocking are negligible.

We discuss how A1 and A3 can be relaxed in Sec. 8 and how to support A2 in Sec. 6.

Refining P1–P3. Properties P1–P3 can be ensured by maintaining the following properties.

P1.1 If a job's task execution budget expires, then it has no active request (to satisfy P1).

P2.1 If the CS execution budget of a CS expires, then the CS is aborted without corrupting shared-resource state (to satisfy P2).

P3.1 Execution-time variances in executing timer-handling and lock/unlock logic cannot cause task and CS execution budgets to be exceeded (to satisfy P3).

We show how to satisfy P1.1 and P3.1 in the OR-FMLP and OR-OMLP in Secs. 4 and 5. We also show how to satisfy P2.1 in Sec. 6 for the case of shared data structures. In order to focus on P1.1 and P3.1 for now, we make the following assumption.

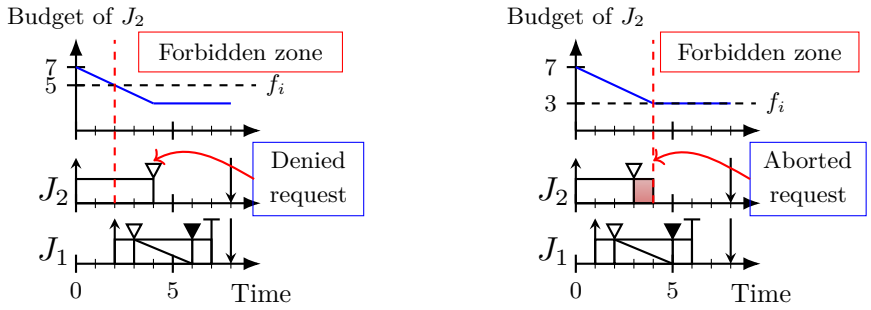
A4 Property P2.1 is satisfied.

4 OR-FMLP

In this section, we introduce the *overrun-resilient flexible multiprocessor locking protocol (OR-FMLP)*, an extension of the FMLP [9] that achieves overrun resiliency by enforcing task and CS execution budgets. Like the FMLP, a job is non-preemptive when executing the OR-FMLP (while both spinning and executing its CS). The OR-FMLP satisfies P1.1 by using a previously proposed concept called a *forbidden zone* (FZ) [27] that aids in budget enforcement—in fact, the OR-FMLP is very similar to a protocol called the "Skip Protocol" presented in [27]. In our setting, however, much care is required in deriving execution budgets so that "chicken and egg" problems are avoided. The goal of avoiding such problems has a major bearing on the overall lock design and its analysis.

Design goal. Spinlocks provide mutual exclusion without OS support, eliminating system-call overheads. While some timer-related OS support is needed, our overriding design goal is nonetheless the following.

G1 Minimize the number of the OS invocations.



(a) OR-FMLP **(b)** Fine-grained variant

Figure 5 Illustration of FZs. Overheads/delays other than spinning are omitted to avoid clutter.

Managing CS-budgeting timers. To prevent CS execution budget overruns, OS invocations are needed, contrary to G1, to manage CS-budgeting timers. It is perhaps theoretically possible to avoid using such timers by having the CS itself repeatedly monitor the CS execution budget remaining, but such an approach would have very high overhead.

Satisfying P1.1. We satisfy P1.1 by employing FZs, as mentioned above. When a job is allocated its task execution budget, a portion of that budget at the end constitutes its FZ. A job is not allowed to issue a resource request during its FZ. The length of J_i's FZ, denoted by f_i, is the maximum task execution budget of J_i that can be consumed when a request of J_i is active. Under the OR-FMLP, this task execution budget consumption includes both its CS length and spinning time. The OR-FMLP adds an additional "FZ check" prior to performing the locking logic of the FMLP. This check, which is assumed to be part of the locking overhead of the OR-FMLP, can be implemented entirely in user space by having the OS record the current time as given by the local timestamp counter (TSC) in a shared control page whenever a job begins or resumes execution. Using this recorded value and the current local TSC value, a job can determine whether it is in its FZ.

▶ **Example 4.** Fig. 5(a) depicts two jobs that issue requests to the OR-FMLP for resource ℓ_1. Assume that J_2's task execution budget is 7.0 units and its CS execution budget is 2.0 units. J_2 could potentially be blocked by J_1 for 3.0 time units, the length of J_1's CS execution. J_2 enters its FZ at time 2 as it does not have sufficient task execution budget to spin for 3.0 time units and then execute its CS for 2.0 time units. Thus, its request is denied at time 4.

Satisfying P3.1. To satisfy P3.1, we derive task and CS execution budgets by accounting for all lock- and timer-related overheads/delays, given in Tbl. 2, using their ETBs. Before deriving these terms, we first give the rules of the OR-FMLP.

OR-FMLP Rules. We assume the following properties, which we justify later.

B1 The execution and analytical budgets of all tasks and CSs have been determined.

B2 A job's FZ length can be derived from task/CS base, execution, and analytical budgets.

When a job J_i attempts to issue a request $\mathcal{R}$ for a resource ℓ_k, it proceeds according to the following rules (J_i is non-preemptive while executing according to these rules).

F1 $\mathcal{R}$ is issued only if J_i's remaining task execution budget is at least f_i; otherwise, $\mathcal{R}$ is denied. Issued requests spin (if necessary) in per-resource FIFO queues until satisfied. (Policies for handling denied or aborted requests are an application-level concern.)

F2 When $\mathcal{R}$ is satisfied, J_i's CS-budgeting timer is set to expire L_i^e time units in the future.

F3 When J_i's CS completes, J_i's CS-budgeting timer is stopped and J_i releases ℓ_k. If the CS-budgeting timer expires prior to CS completion, then ℓ_k is released (*i.e.*, J_i's CS is aborted, as allowed by Assumption A4).

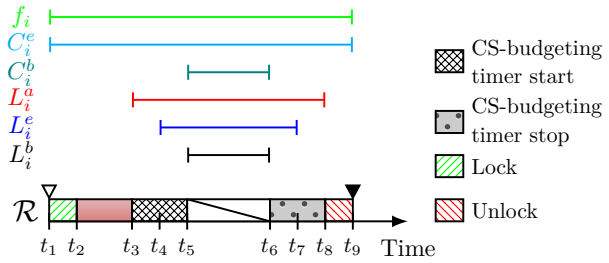


Figure 6 OR-FMLP request timeline with overheads included.

Addressing B1 and B2. We now address Properties B1 and B2. Fig. 6 depicts the execution of a request $\mathcal{R}$ of a job J_i, with overheads included, during the time interval from $\mathcal{R}$'s issuance until its completion. During this interval, J_i issues $\mathcal{R}$ by inserting $\mathcal{R}$ into the FIFO spin-queue during $[t_1, t_2)$, spins (if required) during $[t_2, t_3)$, starts its CS-budgeting timer during $[t_3, t_5)$, executes its CS during $[t_5, t_6)$, stops its CS-budgeting timer during $[t_6, t_8)$, and unlocks its acquired resource during $[t_8, t_9)$. Using this figure as a reference, we now derive the various terms mentioned in B1 and B2.

CS execution budget. We derive L_i^e by inflating L_i^b to account for its CS execution budget consumption due to CS-budgeting timer overheads. Since J_i's CS executes non-preemptively under the OR-FMLP, J_i incurs CS-budgeting timer overheads only when its CS starts and completes execution. However, the CS-budgeting timer can start or stop at an arbitrary time point within the OS's timer-handling code, as shown by times t_4 and t_7 in Fig. 6, respectively. Since $(t_5 - t_4) \leq \Delta_{tb}$ and $(t_7 - t_6) \leq \Delta_{te}$, J_i's CS execution budget is consumed by at most $\Delta_{tb} + \Delta_{te}$ units due to starting and stopping its CS-budgeting timer. Expiring the CS-budgeting timer does not consume any CS execution budget because it occurs only when the CS execution budget is fully consumed. Thus, we have

$$L_i^e = L_i^b + \Delta_{tb} + \Delta_{te}. \tag{1}$$

Analytical CS budget. The above derivation of L_i^e pessimistically assumes that t_4 (resp., t_7) is close to t_3 (resp., t_8). In reality, t_4 (resp., t_7) could instead be close to t_5 (resp., t_6), implying that we must inflate again for timer overheads in determining the analytical CS budget. With this in mind, we derive L_i^a, represented by $[t_3, t_8)$ in Fig. 6, by inflating L_i^e to account for its task execution budget consumption due to CS-budgeting timer overheads. J_i's task execution budget is consumed by at most L_i^e units during $[t_4, t_7)$. Before (resp., after) the CS-budgeting timer actually starts (resp., stops), the timer-handling code may execute for at most Δ_{tb} (resp., Δ_{te}) time units during $[t_3, t_4)$ (resp., $[t_7, t_8)$). If the CS-budgeting timer of J_i expires, then the expiration and CS abort take at most Δ_{tt} time units (by Assumption A2). Since the timer stop and expiration cannot both occur for a CS, we have

$$L_i^a = L_i^e + \Delta_{tb} + \max(\Delta_{te}, \Delta_{tt}). \tag{2}$$

Request blocking time. Under the FMLP, a request $\mathcal{R}$ for a resource ℓ_k by a job J_i can be blocked by at most $m-1$ requests for ℓ_k by other jobs. A request $\mathcal{R}'$ by a job J_j that blocks $\mathcal{R}$ can do so for the entire duration when $\mathcal{R}'$ is satisfied. This duration includes the time needed for $\mathcal{R}'$ to **(i)** start its the CS-budgeting timer, **(ii)** execute its CS, **(ii)** stop/expire its CS-budgeting timer, and then **(iv)** unlock ℓ_k. This time interval is analogous to $[t_3, t_9)$ for J_j in Fig. 6. L_j^a upper bounds the total time for (i)–(iii) and Δ_{unlock} upper bounds the time for (iv). It follows that

$$B_i = \sum_{m-1 \text{ largest in } \Gamma_k} (L_j^a + \Delta_{unlock}). \tag{3}$$

FZ length. J_i's FZ length, f_i, is the maximum task execution budget of J_i that can be consumed during the time interval when its request $\mathcal{R}$ is active. This time interval corresponds to $[t_1, t_9)$ in Fig. 6. J_i issues $\mathcal{R}$ during $[t_1, t_2)$, which takes at most Δ_{lock} time units. It then busy-waits for at most B_i time units during $[t_2, t_3)$. It subsequently executes its CS and timer-handling code for at most L_i^a time units during $[t_3, t_8)$ and then unlocks ℓ_k during $[t_8, t_9)$, which requires at most Δ_{unlock} time units. Therefore,

$$f_i = B_i + \Delta_{lock} + L_i^a + \Delta_{unlock}. \tag{4}$$

Task execution budget. We derive J_i's task execution budget by inflating its base task budget to account for spinning time, locking and unlocking overheads, and task- and CS-budgeting timer overheads incurred when its request $\mathcal{R}$ is active (see Tbl. 2). These overheads/delays correspond to all of $[t_1, t_9)$ except $[t_5, t_6)$ in Fig. 6. Since f_i (resp., L_i^b) corresponds to $[t_1, t_9)$ (resp., $[t_5, t_6)$), J_i's task execution budget is

$$C_i^e = C_i^b + f_i - L_i^b.$$

Analytical task budget. J_i's analytical task budget is obtained by inflating its task execution budget to account for non-preemptive blocking and a potential task-budgeting timer expiration. Expiring the task-budgeting timer takes at most Δ_{tt} time units. Because jobs invoke the OR-FMLP non-preemptively, a newly released job may be blocked by lower-priority jobs for the duration of m CSs (inflated to include overheads). Reasoning similarly to (3), we have

$$NPB_i = \sum_{m \text{ largest}} (L_j^a + \Delta_{unlock}).$$

Therefore, J_i's analytical task budget is

$$C_i^a = C_i^e + NPB_i + \Delta_{tt}.$$

Fine-grained FZs and why they are problematic. FZs were originally proposed to be policed upon CS entry [27], but here we have policed them in a more coarse-grained way by also including spinning time. We made this choice to avoid interactions with the OS, per Goal G1, to maintain the use of simple user-level synchronization code, and to reduce the length of code sequences that require ETBs. Here we briefly explore the fine-grained choice of defining FZ lengths based on CS execution times only.

The main advantage of the fine-grained approach is that FZ lengths are shorter. However, now a job may exhaust its task execution budget *while executing within the locking protocol*. In this case, its request must be extracted from the FIFO spin-queue. Letting Δ_{abort} denote that time required to do this, it can be shown that (4) can be replaced by

$$f_i = \max(L_{max}^a + \Delta_{unlock}, \Delta_{abort}). \tag{5}$$

▶ **Example 5.** Fig. 5(b) depicts two jobs that issue requests for resource ℓ_1. At time 3, J_2 issues a request $\mathcal{R}$ for ℓ_1. Assume that J_2's task and CS execution budgets are 7.0 and 2.0 units, respectively, and extracting $\mathcal{R}$ from the FIFO spin-queue requires 3.0 time units. Then, J_2's fine-grained FZ length is $\max\{2, 3\} = 3$. Thus, J_2 reaches its FZ at time 4 after consuming 4.0 units of its task execution budget.

Significant prior research has been directed at *abortable spinlocks* that allow requests to be aborted [1, 2, 29, 33, 38, 44]. Two approaches have been investigated in designing such locks.

The first approach aborts requests "lazily" by setting a removal flag [2, 33, 38, 44]. Proper request removal is performed later by another job whose resource request is pending or satisfied. This removal requires $O(m^2)$ time [33], which can significantly increase FZ lengths. In the second approach, an aborted request is removed immediately [1, 29]. In existing algorithms, such a removal requires $O(\min(m, \log n))$ time complexity or worse. Moreover, abortable spinlocks require complicated lock/unlock/abort logic. This would significantly increase the ETBs associated with that logic. In contrast, a simple ticket lock can be used in the course-grained variant, thus reducing the length of code sequences requiring ETBs.

5 OR-OMLP

In this section, we introduce the *overrun-resilient $O(m)$ locking protocol (OR-OMLP)*. The original OMLP executes CSs preemptively and uses *priority inheritance* [37] as a progress mechanism to ensure that if a job waiting to access a resource ℓ_k is among the m highest-priority jobs, then the currently satisfied request for ℓ_k is scheduled. The OR-OMLP executes CSs non-preemptively (for the timing-analysis-related reasons discussed in Sec. 3) but retains priority inheritance as a progress mechanism. Priority inheritance is still needed to ensure that when a (non-preemptive) CS ends, the next queued request will be satisfied if it is blocking a job whose priority is among the top m. Non-preemptive CSs do not alter the OMLP's request-blocking bounds, but introduce non-preemptive blocking, which the OMLP avoids.

Similar to the OR-FMLP, the OR-OMLP uses FZs and ETBs of lock- and timer-related overheads to satisfy P1.1 and P3.1, respectively, but here, FZs are fine-grained (*i.e.*, policed on CS entry). Having already seen how these basic mechanisms work in the context of the OR-FMLP, we proceed directly to defining the rules of the OR-OMLP.

OR-OMLP rules. Our description of the OR-OMLP focuses on a single resource ℓ_k, for which there are two queues, an m-element FIFO queue FQ and a priority queue PQ, as shown in Fig. 1. When a job J_i attempts to issue a request $\mathcal{R}$ for ℓ_k, it follows the rules below, which are specified assuming that B1 and B2 in Sec. 4 hold, and that a job's task-budgeting timer starts (resp., stops) when it begins (resp., ceases) to execute.

O1 $\mathcal{R}$ is issued only if J_i's remaining task execution budget is at least f_i; otherwise, $\mathcal{R}$ is denied. If not denied, $\mathcal{R}$ is enqueued in FQ if fewer than m requests for ℓ_k are already active; otherwise, it is added to PQ.

O2 All queued jobs except the job at the head of FQ are suspended. The job at the head of FQ inherits the priority of the highest-priority job in FQ or PQ.

O3 If $\mathcal{R}$ becomes the head of FQ at time t, then it is satisfied and J_i becomes eligible to be scheduled at time t (this depends on its perhaps-inherited priority). If J_j's request was the head of FQ before time t and J_i is among the m highest-priority jobs at time t but cannot preempt the lowest-priority scheduled job due to non-preemptivity, then J_i preempts J_j (even if J_j is one of the top m priority jobs). Once scheduled, J_i's CS-budgeting timer is set to expire L_i^e time units in the future, and J_i executes its CS non-preemptively.

O4 When J_i's CS completes, its CS-budgeting timer is stopped, J_i releases ℓ_k, and J_i becomes preemptive. If instead its CS-budgeting timer expires prior to its CS completion, then ℓ_k is released (*i.e.*, $\mathcal{R}$ is aborted, as allowed by Assumption A4). In either case, $\mathcal{R}$ is dequeued from FQ and the highest-priority request from PQ is moved to the tail of FQ.

Lazy preemptions. In addition to the above rules, we enact preemptions (except the preemption mentioned by Rule O3) *lazily* by delaying any preemption until the lowest-priority scheduled job becomes preemptable instead of preempting the first-available lower-priority job [10, §3.3.3]. Lazy preemptions prevent a job from incurring repeated pi-blocking each time

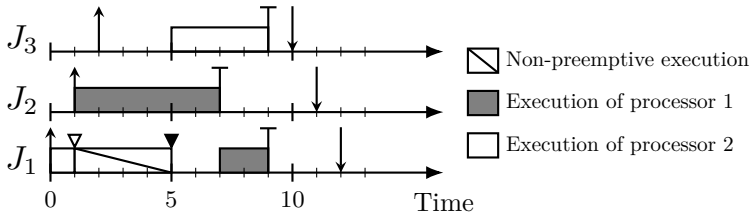


Figure 7 Example illustrating lazy preemptions and link-based scheduling.

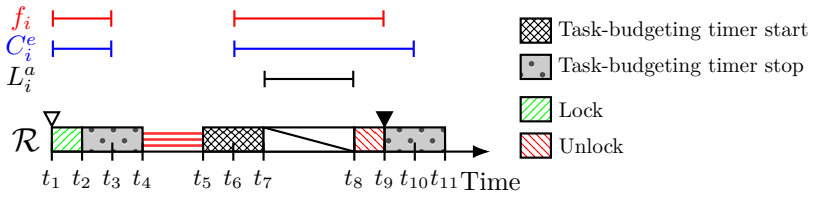


Figure 8 Timeline of an active request under the OR-OMLP.

a higher-priority job is released when the scheduler implementation is *link-based* [10, §3.3.3], which we assume.

▶ **Example 6.** Lazy preemption is depicted in Fig. 7. Three jobs are scheduled on two processors under link-based G-EDF. When J_3 is released at time 2, the scheduler links J_3 to processor 1, which is executing the lowest-priority job among both processors. Since J_1 is scheduled on processor 1 and is nonpreemptive at time 2, J_3 is not scheduled. At time 5, J_1 becomes preemptive again, and J_3, the job linked to processor 1, is scheduled on processor 1. Further details concerning lazy preemptions and link-based scheduling are given in an online appendix [41].

Addressing B1 and B2. We deal with these properties similarly as for the OR-FMLP. Fig. 8 depicts the execution of a request $\mathcal{R}$ of a job J_i, with overheads included. We omit CS-budgeting timers in Fig. 8 as their accounting is the same as for the OR-FMLP, but we include task-budgeting timers as they are relevant to suspension-based locks. J_i's lock request is issued by enqueueing it in the relevant queue during $[t_1, t_2)$. If ℓ_k is held by another job, then J_i suspends. Before suspending J_i at t_4, J_i's task-budgeting timer is stopped during $[t_2, t_4)$. J_i's task-budgeting timer is started during $[t_5, t_6)$ after it is scheduled again upon satisfaction of $\mathcal{R}$ at t_5. During $[t_7, t_8)$, J_i's CS-budgeting timer starts, its CS executes, and then its CS-budgeting timer stops. J_i unlocks ℓ_k during $[t_8, t_9)$. If J_i is preempted due to Rule O3 after it unlocks ℓ_k, then its task-budgeting timer is stopped during $[t_9, t_{11})$. Using this figure as reference, we now derive the various terms mentioned in B1 and B2.

CS execution and analytical budgets. Since CSs execute non-preemptively, the analysis of CS execution and analytical budgets is the same as under the OR-FMLP. Thus, J_i's CS execution budget and analytical CS budgets are determined by (1) and (2), respectively.

Task execution budget. Locking and unlocking overheads during $[t_1, t_2)$ and $[t_8, t_9)$, respectively, occur when J_i's task-budgeting timer is active and thus consume at most Δ_{lock} and Δ_{unlock} units, respectively, of J_i's task execution budget. J_i's task-budgeting timer actually starts (resp., stops) at an arbitrary time point t_6 (resp., t_3 and t_{10}) within the OS's timer-handling code. Thus, $(t_3 - t_2) + (t_{10} - t_9) \leq 2\Delta_{te}$ (resp., $(t_7 - t_6) \leq \Delta_{tb}$) units are consumed from J_i's task execution budget when J_i is suspended/preempted (resp., resumed). By the FZ check in Rule O1, J_i's task-budgeting timer cannot expire when $\mathcal{R}$ is active. The CS-budgeting timer overheads are accounted for in L_i^a, and are at most $L_i^a - L_i^b$. Thus, J_i's

task execution budget is

$$C_i^e = C_i^b + \Delta_{lock} + \Delta_{unlock} + \Delta_{tb} + 2\Delta_{te} + L_i^a - L_i^b.$$

FZ length. f_i is given by the maximum task execution budget of J_i that can be consumed during the time interval when $\mathcal{R}$ is active. J_i consumes its task execution budget throughout all of $[t_1, t_9)$ except within $[t_3, t_6)$. Therefore,

$$f_i = \Delta_{lock} + \Delta_{unlock} + \Delta_{tb} + \Delta_{te} + L_i^a.$$

Request blocking. Under the OMLP, a job J_i can be pi-blocked by the length of at most $2(m-1)$ requests for ℓ_k [12]. This result hinges on a progress mechanism, which ensures the progress of the job J_j holding ℓ_k whenever J_i is request-blocked. Under the OR-OMLP, Rule O3 and priority inheritance ensure the same progress property. When J_j becomes the head of FQ, Rule O3 ensures that it is scheduled if its (perhaps-inherited) priority is one of the top m despite any non-preemptive execution of lower-priority jobs. This may cause non-preemptive blocking for the previous resource holder (if any), which we discuss later. Priority inheritance ensures that J_j can be scheduled when its priority is raised because of J_i's request issuance. (We give a formal proof in an online appendix [41].)

By preserving the same progress property as the OMLP, the OR-OMLP has the same request-blocking bounds as the OMLP. A request $\mathcal{R}'$ by J_j can pi-block J_i for the duration in which $\mathcal{R}'$ is satisfied, which is analogous to the time interval $[t_5, t_9)$ in Fig. 8. This duration includes the time needed for $\mathcal{R}'$ to **(i)** start its task-budgeting timer, **(ii)** start its CS-budgeting timer, **(iii)** execute its CS, **(iv)** stop its CS-budgeting timer, and **(v)** unlock ℓ_k. Δ_{tb} upper bounds (i), L_i^a upper bounds (ii)–(iv), and Δ_{unlock} upper bounds (v). Additionally, J_j can be preempted before the next job holding ℓ_k can be scheduled. This causes J_j's task-budgeting timer to stop during $[t_9, t_{11})$, which takes at most Δ_{te} time. Therefore,

$$B_i = 2 \cdot (m-1) \cdot (\Delta_{tb} + \Delta_{unlock} + \Delta_{te} + \max_{\tau_j \in \Gamma_k} \{L_j^a\}). \tag{6}$$

Non-preemptive blocking. With lazy preemptions, J_i can incur non-preemptive blocking when it releases ℓ_k (due to Rule O3). (We give an example of the latter case in an online appendix [41].) Note that a job can be pi-blocked when a resource is released even under the OMLP if there is a task with non-preemptive sections [10, §3.3.3]. However, such pi-blocking can be analytically treated the same as pi-blocking incurred upon job release by considering the remaining portion of the job as a new job. Each job release can cause pi-blocking for the length of at most one CS [10, §3.3.3]. Reasoning as above for (6), we have

$$NPB_i = 2 \cdot (\Delta_{tb} + \Delta_{unlock} + \Delta_{te} + \max\{L_j^a\}).$$

Analytical task budget. We derive C_i^a by inflating C_i^e to account for request blocking time B_i, non-preemptive blocking time NPB_i, task-budgeting timer expiration overhead Δ_{tt}, and task-budgeting timer starting/stopping overheads during $[t_3, t_4)$, $[t_5, t_6)$, and $[t_{10}, t_{11})$. Since $(t_4 - t_3) + (t_{11} - t_{10}) \leq 2\Delta_{te}$ and $(t_6 - t_5) \leq \Delta_{tb}$, we have

$$C_i^a = C_i^e + B_i + NPB_i + \Delta_{tb} + 2\Delta_{te} + \Delta_{tt}.$$

Coarse-grained FZs. Is it possible to have a OR-OMLP variant with coarse-grained FZs like the OR-FMLP? Such a variant would actually be quite tricky to implement due to the need to track task budget consumption by waiting jobs. A waiting job's task budget should be consumed only when it is pi-blocked, and under s-oblivious analysis, not all suspension time "counts" as pi-blocking time [12]. This nuance greatly complicates budget tracking.

6 Abortable Critical Sections

In this section, we introduce *abortable* CSs, which enable operations on shared data structures to be aborted without undo code. Abortable CSs are inspired by word-based obstruction-free STM, which *linearizes* multiple operations to a single instruction, but can only ensure progress in the presence of a contention manager. By executing instructions in CSs, the locking protocol serves as a strong contention manager, allowing us to simplify and address issues in prior obstruction-free techniques.

Undo code problem. The following example shows the necessity of undo code when an ordinary CS is aborted.

▶ **Example 7.** Consider the MODIFY procedure in Alg. 1, which updates a two-word buffer $M[1..2]$ by incrementing each $M[i]$ by $M[1]$'s value. If the procedure is aborted after completing the first **for**-loop iteration, then the buffer is left in an inconstent state. In order to restore M to a valid state, undo code would need to set $M[1]$ to its old value.

While the undo code above is simple, such code can be much more complicated for operations that make many changes to object state. Undo code also needs to be provisioned using its ETB to satisfy P3.1, which can be as pessimistic as the ETB of the CS itself.

Prior work on versioning techniques. Prior work on versioning techniques attempt to obviate undo code through various means, but they all have unfortunate limitations in our context. Interruptible CSs (ICSs [30]) use the idea of a *continuation* [42] to eliminate undo code by appending memory modifications to a log, which will be applied before the next CS entry. Unfortunately, ICSs can force short CSs to apply the memory modifications of long CSs. When CSs only modify memory, each CS length may increase by the length of the longest CS in the system. Object-based obstruction-free STMs [21, 26, 35] do not face this issue, but may require coarse-grained copies of an entire data structure when only small modifications are performed. Word-based variants [22–24] eliminate the need for coarse-grained copying, but require a garbage collector. Other protocols such as TL2 [19] fix both the problems of word-based STMs and continuations, but can require a lengthy clean-up process on abort.

Our abortable CS concept leverages locks as a strong contention manager. It also addresses the issue present in ICSs, and word-based STMs' reliance on garbage collectors without requiring a lengthy clean-up process like TL2. We now explain this concept by showing how to convert MODIFY into an abortable version, ABORTABLEMODIFY, also given in Alg. 1. We first describe the data structures involved.

Data structures. We represent each $M[i]$ using the structure shown in Fig. 9(a) and associate a CS, *e.g.*, an invocation of ABORTABLEMODIFY, with a *transaction record* as defined in Fig. 9(b). The fields $M[i].old$ and $M[i].new$ contain the *valid* value of $M[i]$—*i.e.*, the value written by the last unaborted request involving a write to $M[i]$—before and after reaching the linearization point, respectively.[3] We modify only $M[i].new$ within a CS before it reaches its linearization point. The *txn* field of $M[i]$ is a pointer to a transaction record, which is set when $M[i]$ is updated in ABORTABLEMODIFY.

The *rc1* and *rc2* fields in a transaction record count the number of $M[i]$ structures that point to that record, which are used to determine when the record is no longer in use, as discussed later. The *done* field indicates whether the CS corresponding to the record has

[3] $M[i]$ is a pointer, so technically we should use notation like $M[i]\rightarrow old$ to indicate that it must be dereferenced before accessing the *old* field. We have opted for simpler notation that is more readable.

Algorithm 1 Example buffer data structure.

Variables:
$M[1..2]$: A shared array of words

```
1: procedure Modify(M)
2:     x := M[1]
3:     for i ∈ {1, 2} do
4:         M[i] := M[i] + x
5:     end for
6: end procedure
```

Variables:
$M[1..2]$: A shared array of type in Fig. 9(a)
data : Ptr of data structure in Fig. 9(a)
txn : Ptr to txn_record in Fig. 9(b)
new_txn : Ptr to txn_record in Fig. 9(b)
free_stack : Stack of free transaction records
free_stack_top : Ptr to top of *free_stack*

```
1: procedure AbortableModify(M)
2:     new_txn := NULL
3:     if M[1].txn ≠ NULL ∧ M[1].txn.done
       then
4:         x := M[1].new
5:     else
6:         x := M[1].old
7:     end if
8:     for i ∈ {1, 2} do
9:         txn := M[i].txn
10:        data := &(M[i])
11:        if txn ≠ NULL then
12:            if txn.done then
13:                data.old := data.new
14:            end if
15:            txn.rc1 := txn.rc1 − 1
16:            if txn.rc1 = 0 then
17:                txn.done := false
18:                txn.next := free_stack_top
19:                free_stack_top := txn
20:            end if
21:            data.txn := NULL
22:            txn.rc2 := txn.rc2 − 1
23:        end if
24:        txn := new_txn
25:        if new_txn = NULL then
26:            txn := free_stack_top
27:            txn.rc2 := txn.rc2 + 1
28:            data.txn := txn
29:            free_stack_top := free_stack_top.next
30:            txn.rc1 := txn.rc1 + 1
31:        else
32:            txn.rc2 := txn.rc2 + 1
33:            data.txn := txn
34:            txn.rc1 := txn.rc1 + 1
35:        end if
36:        new_txn := txn
37:        data.new := data.old + x
38:    end for
39:    new_txn.done := true
40: end procedure
```

successfully been completed or not. The computation of a CS linearizes to a single write that sets its transaction record's *done* field to *true*. Thus, we maintain the following invariant.

I $M[i].old$ contains $M[i]$'s valid value if $M[i].txn.done$ is *false* or $M[i].txn$ is NULL. $M[i].new$ contains $M[i]$'s valid value if $M[i].txn.done$ is *true*.

The *next* field in a transaction record is used to maintain a stack *free_stack* of *free* transaction records that are not pointed to by any $M[i]$. This *free_stack* is used to reuse a transaction record for future CSs. We now describe the code in AbortableModify.

Reads of shared variables. Lines 3–6 in AbortableModify replace line 2 of Modify. These lines read $M[1]$'s valid value from either the *new* or *old* field of $M[1]$ based on Invariant I.

Writes of shared variables. Lines 9–39 in AbortableModify replace the write to $M[i]$ in line 4 of Modify. We note that, while these lines reflect our general transformation

```
struct {
    old: word
    new: word
    txn: Ptr to txn_record
} data
```

(a) Data element structure

```
struct {
    rc1: int
    rc2: int
    done: boolean
    next: Ptr to txn_record
} txn_record
```

(b) Transaction record structure

Figure 9 Data structures for abortable CS.

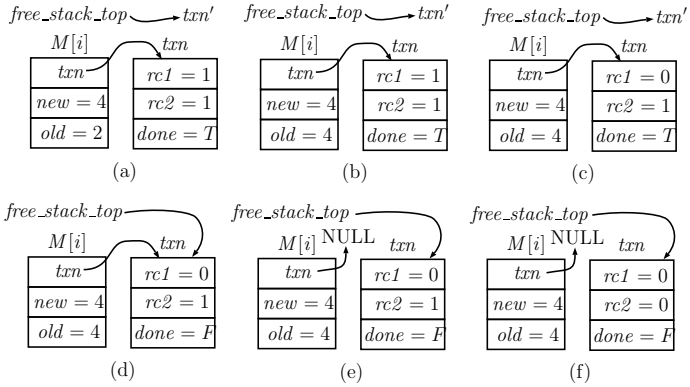


Figure 10 Unlinking process. The field *txn.next* is not shown.

process for making a CS abortable, it is possible to shorten this code for this simple example. A write to $M[i]$ in AbortableModify occurs in three steps: **(i)** unlink $M[i]$ from its old transaction record to enable future reuse of that record; **(ii)** link $M[i]$ with the transaction record corresponding to this CS invocation; and **(iii)** commit that invocation, *i.e.*, make it take effect atomically. We now explain these steps.

Step (i): Unlinking. Lines 12–22 in AbortableModify unlink $M[i]$ from its old transaction record stored in *txn* by line 9. We depict the steps of unlinking in Fig. 10, where inset (a) shows an initial state prior to executing lines 12–22. To maintain Invariant I, lines 12 and 13 copy $M[i].new$ to $M[i].old$ if *txn.done* is *true*, as shown in Fig. 10(b). Lines 15, 21, and 22 decrement *rc1* and *rc2* of *txn* before and after setting $M[i].txn$ to NULL, respectively, as shown in insets (c)–(f) of Fig. 10. If *rc1* becomes 0 after line 15, as shown in Fig. 10(c), then lines 17–19 push *txn* onto *free_stack* and unset its *done* field, as shown in Fig. 10(d).

Step (ii): Linking. Lines 24–35 link $M[i]$ to a transaction record. Line 24 assigns *new_txn* to *txn*, which is the transaction record corresponding to this invocation of AbortableModify. We illustrate the linking process when *new_txn* is NULL by considering insets (b)–(f) of Fig. 10 in reverse order. Fig. 10(f) shows an initial state after executing line 26 of the linking process when *new_txn* is NULL. Insets (e), (d), (c), and (b) of Fig. 10 illustrate incrementing *rc2* (line 27), linking *txn* to $M[i]$ (line 28), removing *txn* from *free_stack* (line 29), and incrementing *rc1* (line 30), respectively. After this linking process, line 36 sets *new_txn* to *txn* to ensure that future loop iterations use this same transaction record. Also, line 37 performs the write operation (from line 4 of Modify) by updating $M[i].new$.

Step (iii): Committing. The CS is committed by simply setting *new_txn.done* to *true* in line 39. It is this one-line commit at the end that obviates the need for any undo code.

Why two reference counters? To see why using only one counter is problematic, consider again the unlinking process shown in Fig. 10. If only *rc1* is used and the CS is aborted after executing the step in Fig. 10(c), then *txn* is left in an inconsistent state, *i.e.*, its *rc1* field indicates no structure points to *txn* yet one does. Using both *rc1* and *rc2* enables this "inopportune" CS abort to be detected by simply checking whether *rc1* is smaller than *rc2*. To fix the inconsistent transaction record in this case, we add a small code sequence to the OS timer-handling code that deals with CS-budgeting timer expirations. This code completes the remaining steps of unlinking by executing lines 15–21 of AbortableModify if *txn.rc1* is smaller than *txn.rc2* (with an additional check of *txn* = *free_stack_top* in line 15 to prevent inserting *txn* to *free_stack* twice). The timer-handling code can access the CS-specific variables involved in these actions via a control page shared with the CS's task. Note that *the added timer-handling code is the same for all abortable CSs and deriving the EBT of*

expiring the CS-budgeting timer after adding this code supports Assumption A2. Similar inopportune CS aborts can affect the linking process, and they are dealt with similarly.

Generalizing to arbitrary shared data structures. For clarity, we presented the idea of an abortable CS via a simple example. However, in an online appendix [41], we present a set of routines for performing necessary actions (reading, writing, linking, unlinking), a set of rules to transform any ordinary CS into an abortable one, and an invariant-based proof that shows that these rules are correct. In our simple example, reading and writing occurred at the granularity of words. However, in the general scheme, any granularity can be assumed.

Have we really eliminated undo code? One could argue that abortable CSs merely intertwine undo-related actions with ordinary CS code. If this is so, do they offer any real advantages over simply following ordinary CS code with potential undo code? The answer is yes. With separate undo code, the "chicken and egg" problem mentioned in Sec. 1 arises: the undo code would have to be budgeted, and to avoid exhausting that budget, an ETB would have to be assumed for it, which could be very costly. Perhaps one could take the same "intertwined view" and inflate the cost of any CS by its undo cost, but then how is the (separate) undo code ever triggered? Presumably, the combined budget would have to be factored into two parts, one for the ordinary CS code and one for the undo code, bringing us back to the chicken-and-egg problem. While abortable CSs are immune from these problems, further research into supporting real-time undo code would certainly be valuable.

7 Experimental Evaluation

To assess the costs and benefits of overrun-resilient locking, we conducted two sets of experiments under LITMUS$^{\mathrm{RT}}$ [10, 15] on an eight-core 2.1GHz Intel Xeon Silver processor. To increase timing predictability, we disabled hyperthreading, low CPU power states, and CPU frequency scaling.

Experiment 1. We first assessed the costs of using abortable CSs that satisfy P2.1 vs. using CS ETBs to provision CS execution budgets. As a baseline, we measured the execution times of ordinary (non-abortable) operations on buffers, queues, and binary heaps. To assess the cost of abortable CSs, we compared the baseline to the execution times of corresponding abortable CS implementations. To assess CS ETB budget provisioning (which assumes unrealistically pessimistic conditions), we compared the baseline to the execution times of cacheless runs of the ordinary operations. Two metrics were considered: the *worst-case* (resp., *average-case*) *inflation factor* of an operation is the ratio between the observed maximum (resp., average) execution times of abortable/cacheless CS vs. that of the baseline. As this is not a paper on timing analysis, we note that the ETBs assumed here are provided to give a plausible sense of the pessimism they may entail; we make no claim that they are in fact upper bounds, or if they are, that they cannot be safely tightened.

For each implementation, we determined the maximum and average duration of each operation (measured using the timestamp counter) through 10,000 trials, running alongside contention-generating tasks that contend for the memory bus. We separately measured the duration of our timing code and subtracted it from our results. For the read/write buffer, we used a one-word buffer with single-word reads and writes. We initialized the queue and heap to contain 1,000 items. Our results, shown in Tbl. 4, support the following observations.

▶ **Observation 1.** *The worst-case inflation factor of abortable CSs was around two to five.*

▶ **Observation 2.** *The inflation factors for running cacheless was in the hundreds.*

Table 4 Comparison between abortable and ordinary CSs.

Data structure	Buffer Write	Buffer Read	Queue Enqueue	Queue Dequeue	Heap Insert	Heap Extract
WC Baseline	39.0 ns	33.3 ns	46.7 ns	48.6 ns	89.5 ns	203.8 ns
WC Abortable	161.9 ns	76.2 ns	97.1 ns	252.4 ns	300.9 ns	981.9 ns
WC Cacheless	13.0 μs	11.0 μs	22.7 μs	20.6 μs	32.3 μs	15.9 μs
WC Abortable Inflation	4.14×	2.28×	2.08×	5.19×	2.95×	4.08×
AC Abortable Inflation	6.91×	2.55×	2.47×	5.53×	5.88×	3.36×
WC Cacheless Inflation	332.7×	329.8×	486.9×	424.4×	360.7×	784.2×
AC Cacheless Inflation	93.5×	113.5×	304.8×	200.8×	245.6×	882.4×

The extremely high inflation factors for running cacheless were due to both instructions and data being accessed from main memory instead of mainly the L1 cache. While the effects of disabling caches on other processor mechanisms such as branch prediction, pipelining, and prefetching are not well documented, we suspect that these factors also contributed to the slowdown, especially in the case of heaps where branching code is common.

Experiment 2. We assessed the impacts of CS execution budget overruns under the OR-FMLP, OR-OMLP, FMLP, and OMLP by executing a task system consisting of an equal number of synthetic non-overrunning *correct* tasks and overrunning *faulty* tasks. (We did not examine task execution budget overruns because they require application-dependent mitigation; note, however, that common mitigations such as aborting the overrunning job can break the FMLP and OMLP.) Each task τ_i had $(C_i^e, T_i) = (1\text{ms}, 40\text{ms})$. We generated enough tasks so that the sum of all analytical task utilizations (C_i^a/T_i) was $0.8m$.

The task system had a single shared resource for which each job issued a single request. For each synthetic task τ_i, we generated its *CS execution budget* L_i^e and an *actual CS execution time*, denoted L_i, via three steps. When jobs of these synthetic tasks execute CSs, they acquire a lock and spin for the duration of the actual CS execution time. First, for each correct task τ_i, we set L_i^e to 0.2ms and L_i to 0.19ms (which was sufficient to preclude budget overruns due to overheads). Second, for each faulty task τ_i, we set L_i^e to 0.2ms under the FMLP and OMLP. Since abortable CSs require more execution budget, we inflated L_i^e for each faulty task τ_i under the OR-FMLP and OR-OMLP by considering three different *inflation scenarios* based on the data given for buffers, queues, and heaps in Tbl. 4. For each scenario, we inflated each such L_i^e by the worst-case abortable-CS inflation factor of that scenario's data structure in Tbl. 4. Third, for each faulty task τ_i, we determined L_i by a type-1 Gumbel distribution.[4] Under the FMLP and OMLP, the mean of this Gumbel distribution was set at 0.05ms. Under the OR-FMLP and OR-OMLP, this mean was inflated according to the average-case abortable-CS inflation factor given in Tbl. 4 for the corresponding scenario. We varied the probability of a job of a faulty task ovrerrunning its CS execution budget from 0.0 to 0.3 with a step size of 0.05. The variance of the Gumbel distribution was determined by this probability value.

For each inflation scenario and CS execution budget overrun probability, we executed the task system for 10 minutes and measured the maximum response time among all correct tasks. Figs. 11 and 12 plot the recorded response times that supports following observations.

▶ **Observation 3.** *The worst-case response times of correct tasks under the OR-FMLP (resp., OR-OMLP) stayed relatively constant as the overrun probability increased.*

[4] The Gumbel distribution is often used to represent measurement-based probabilistic WCETs [18].

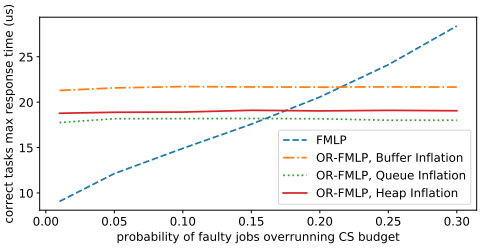


Figure 11 FMLP vs. OR-FMLP results.

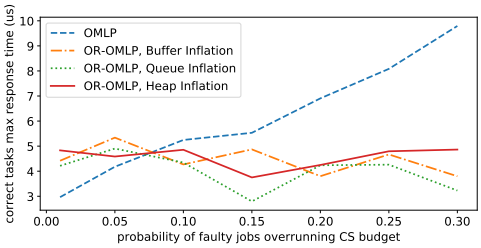


Figure 12 OMLP vs. OR-OMLP results.

▶ **Observation 4.** *Cost/benefit tradeoffs are evident in these curves. For example, for buffers in Fig. 12, overrun-resilient protocols increased response times for overrun probabilities less than 0.10 and decreased them for greater probabilities.*

8 Revisiting Assumptions A1 and A3

We now return to Assumptions A1 and A3.

Assumption A1. As we have seen, user-level budgets have a fundamental dependency on the execution times of certain OS code paths. If one defines budgets for those code paths, then what entity would enforce them? The only alternative to budgeting is to require ETBs for these code paths, but this has major implications for real-time OS (RTOS) designs. For example, modern OSs tend to be highly preemptive, but preemptions greatly complicate measurement-based timing analysis. To avoid "chicken and egg" problems, RTOS designs need to be rethought, with enabling reliable timing analysis for critical code paths being a first-class concern. These code paths should be simple and non-preemptive and should have reasonable ETBs. Techniques like cache locking may help in this regard.

Assumption A3. Standard techniques [10] can account for the overheads/delays considered negligible by A3. From a timing-analysis point of view, the overhead of most concern is CPMD, as it is incurred on preemptions, which as noted already, are hard to deal with in timing analysis due to difficulties in predicting cache state. These difficulties have important implications for synchronization and scheduling: CSs that execute on a CPU (as opposed to, *e.g.*, an I/O device) should be non-preemptive, and tasks should either be non-preemptive or only preemptive at certain "preemption points" [13]. While both task-scheduling options have been studied for simple task models [7, 14, 36], they warrant further attention in more complex models relevant today, such those based on processing graphs [4, 40, 45].

9 Related Work

Locking protocols that consider budget overruns (shown in Tbl. 1) have been explored in the past. However, none satisfy properties P1–P3, which define our notion of overrun resiliency.

Satisfying P1. To satisfy P1, a protocol must deal with jobs overrunning their budgets while in a CS. Prior work such as M-BWI and vMPCP satisfy P1 by allowing task budget overruns to occur inside of CSs and account for them analytically. In contrast, SIRAP and M-BROE satisfy P1 using FZs, introduced in [27], to avoid task budget overruns inside of CSs. Tradeoffs between satisfying P1 using FZs and overrun accounting are detailed in [5].

Satisfying P2. M-BWI and vMPCP both require accurate CS WCETs to produce correct overrun accounting. SIRAP and M-BROE also require accurate CS WCETs to correctly provision FZs. Since CS WCETs may exceed CS PETs, protocols that rely on forbidden

zones and overrun accounting do not satisfy P2. In the absence of correct CS WCETs, ICSs and protocols such as RRP and RACPwP satisfy both P1 and P2 by aborting CSs when their budgets overrun. To maintain consistent state for shared data structures, ICSs, RRP, and RACPwP use versioning techniques.

Satisfying P3. No prior work considers P3. However, all protocols that satisfy P1 *can* satisfy P3 when ETBs are used to account for corresponding budget-enforcement mechanisms. Unfortunately, all protocols that satisfy P2 use versioning techniques that make copies of modified data. Thus, when CSs only modify data, the ETBs of versioning techniques can be as large as the CS ETBs, nullifying the benefits of ensuring P2. Only ICSs avoid the problem by allowing shared-resource state to remain consistent even when a job is aborted while executing the versioning technique, satisfying P3. However, ICSs are intended for uniprocessors, thus the ICS versioning technique cannot handle concurrent resource accesses.

In this work, we proposed the OR-FMLP and OR-OMLP, which in conjunction with abortable CSs, yield overrun-resilient locking protocols. In fact, prior work on versioning techniques can also be used with the OR-FMLP and OR-OMLP to yield overrun-resilient protocols. However, those prior works have unfortunate tradeoffs as discussed in Sec. 6.

10 Conclusion

We have presented the OR-FMLP and the OR-OMLP, which are overrun-resilient variants of the FMLP and the OMLP, respectively. Both the OR-FMLP and the OR-OMLP utilize FZs and the ability to abort CSs to circumvent problems associated with overrunning task and CS budgets. As our designs of these two protocols suggest, it is better to apply FZs in a coarse-grained way in a spinlock but in a fine-grained way in a suspension-based lock. For both protocols, we have carefully worked out how execution budgets should be defined. To easily apply these protocols to support operations on shared data structures, we have also presented abortable CSs, which enable such operations to be aborted with no undo code.

In future work, we plan to consider aborting CSs in other contexts, such as when using locks to access shared hardware resources. We also intend to investigate other types of locks, such as k-exclusion locks and reader/writer locks, as well as mechansims for allocating "slack" generated by underrunning CSs to overrunning CSs. Additionally, full budgeting support enables the possibility of freeing system capacity by intentionally under-budgeting certain computations at the expense of aborting work more often; we plan to explore this possibility as well. This paper also establishes a need for timing-analysis techniques that can guarantee safe ETBs without exorbitant pessimism. The various "chicken and egg" problems we have pointed out warrant scrutiny as well.

References

1 A. Alon and A. Morrison. Deterministic abortable mutual exclusion with sublogarithmic adaptive rmr complexity. In *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing*, pages 27–36, 2018.

2 J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355, 1998.

3 M. Asberg, T. Nolte, and M. Behnam. Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 129–140, 2013.

4 S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 222–231, 2015.

5 M. Behnam, T. Nolte, M. Asberg, and R. Bril. Overrun and skipping in hierarchically scheduled real-time systems. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 519–526, 2009.

6 M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharingin real-time open systems. In *Proceedings of the International Conference on Embedded Software*, page 279–288, 2007.

7 M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 251–260, 2010.

8 A. Biondi, G. Buttazzo, and M. Bertogna. Supporting component-based development in partitioned multiprocessor real-time systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 269–280, 2015.

9 A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56, 2007.

10 B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2011.

11 B. Brandenburg. *Multiprocessor Real-Time Locking Protocols*, pages 1–99. Springer, 2020.

12 B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60, 2010.

13 A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. *Advances in Real-Time Systems*, pages 225–248, 1994.

14 G. Buttazzo, M. Bertogna, and G. Yao. Limited preemptive scheduling for real-time systems. A survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15, 2013.

15 J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–126, 2006.

16 F. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Computing Surveys*, 52(1):14:1–14:35, 2019.

17 R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, 2011.

18 R. Davis and L. Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1):03:1–03:60, 2019.

19 D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, volume 4167, pages 194–208, 2006.

20 D. Faggioli, G. Lipari, and T. Cucinotta. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real Time Systems*, 48(6):789–825, 2012.

21 K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, UK, 2003.

22 T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, page 388–402, 2003.

23 T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. *Commununications of the ACM*, 51(8):91–100, 2008.

24 T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 14–25, 2006.

25 M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003.

26 M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, page 92–101, 2003.

27 P. Holman and J. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 149–158, 2002.

28 P. Holman and J. Anderson. Locking under Pfair scheduling. *ACM Transactions on Computer Systems*, 24(2):140–174, 2006.

29 P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, page 295–304, 2003.

30 T. Johnson and K. Harathi. Interruptible critical sections. Technical Report TR94-007, University of Florida, 1994.

31 H. Kim, S. Wang, and R. Rajkumar. vMPCP: A synchronization framework for multi-core virtual machines. In *Proceedings of the 35th IEEE Real-Time Systems Symposium*, pages 86–95, 2014.

32 N. Kim, B. Ward, M. Chisholm, C.-Y. Fu, J. Anderson, and F.D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *Proceedings of the 22nd IEEE Real-Time Embedded Technology and Applications Symposium*, pages 49–160, April 2016.

33 H. Lee. Fast local-spin abortable mutual exclusion with bounded space. In *Proceedings of the 14th International Conference on Principles of Distributed Systems*, pages 364–379, 2010.

34 C. Maiza, H. Rihani, J. Rivas, J. Goossens, S. Altmeyer, and R. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys*, 52(3):56:1–56:38, 2019.

35 V. Marathe, M. Spear, A. Acharya C. Heriot, D. Eisenstat, W. Scherer, and M. Scott. Lowering the overhead of nonblocking software transactional memory. Technical report, University of Rochester, 11 2006.

36 M. Nasri, G. Nelissen, and B. Brandenburg. A response-time analysis for non-preemptive job sets under global scheduling. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, volume 106, pages 9:1–9:23, 2018.

37 R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.

38 M. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, page 31–40, 2002.

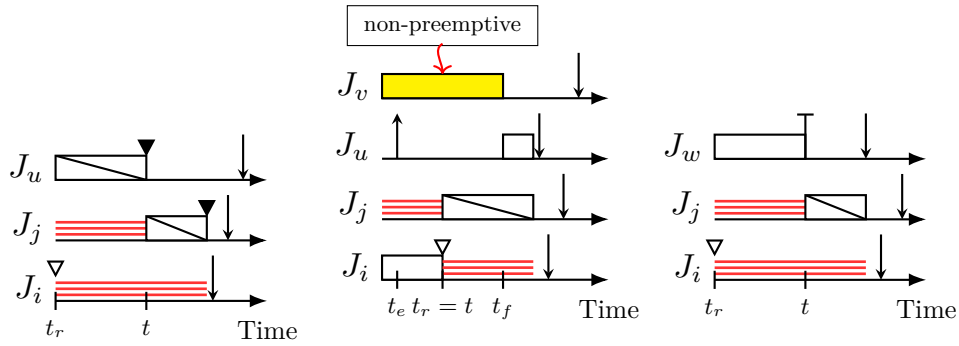
39 T. Springer, S. Peter, and T. Givargis. Resource synchronization in hierarchically scheduled real-time systems using preemptive critical sections. In *Proceedings of the 17th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 293–300, 2014.

40 M. Stigge, P. Ekberg, N. Guan, and W. Yi. The digraph real-time task model. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80, 2011.

41 Z. Tong, S. Ahmed, and J. Anderson. Overrun-resilient multiprocessor real-time locking (longer version), 2022. URL: `http://jamesanderson.web.unc.edu/papers/`.

42 J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, page 212–222, 1992.

43 R. Wilhelm. Real time spent on real time (invited talk). In *Proceedings of the 41st IEEE Real-Time Systems Symposium*, pages 1–2, 2020.



(a) Case 1 **(b)** Subcase 2a **(c)** Subcase 2a

Figure 13 Illustration of the proof of Thm. 8.

44 R. Wisniewski, L. Kontothanassis, and M. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 199–206, 1995.

45 K. Yang, G. Elliott, and J. Anderson. Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 23th International Conference on Real-Time Networks and Systems*, pages 77–86, 2015.

A Proofs and Examples Regarding the OR-OMLP

This appendix contains proofs and examples regarding request and non-preemptive blocking under the OR-OMLP.

Request blocking. We now prove the progress property of the OR-OMLP. We consider a job J_i that issues a request $\mathcal{R}$ for resource ℓ_k at time t_r, where $\mathcal{R}$ completes at time t_s. We refer to the priority of a job as assigned by the scheduling policy, *i.e.*, G-EDF, as its *base* priority; a job's *effective priority* can exceed its base priority due to priority inheritance. Note that whether or not J_i is pi-blocked depends on its base priority. The following theorem ensures progress whenever J_i is pi-blocked during $[t_r, t_s)$.

▶ **Theorem 8.** *If J_i is pi-blocked at time $t \in [t_r, t_s)$, then the job holding ℓ_k is scheduled at time t.*

Proof. We give a proof by contradiction. Let t be the first time instant such that J_i is pi-blocked at time t (and hence is not scheduled at t), but the job J_j holding ℓ_k is not scheduled at time t. We consider two cases.

Case 1. There exists $\varepsilon > 0$ such that J_i is pi-blocked during $[t - \varepsilon, t)$. Fig. 13(a) illustrates this case. Assuming ε is small enough, by the definition of t, the job J_u at the head of FQ prior to t executes its CS during $[t - \varepsilon, t)$. Since CS execution is non-preemptive, J_u must release ℓ_k at time t, allowing J_j to become the job at the head of FQ at time t. Because of priority inheritance, J_j is one of the m highest-priority jobs at time t. By Rule O3, J_j is scheduled at time t, a contradiction.

Case 2. Otherwise. In this case, J_i is not pi-blocked immediately before t and becomes pi-blocked at time t. We first prove the following claim.

▶ **Claim 1.** *If a processor becomes available at time t, then J_j is scheduled at time t.*

Proof. Assume that a processor π_q becomes available at time t. We use Fig. 13(b) for illustration. The base priority of J_i (which is pi-blocked) is among the top m at time t.

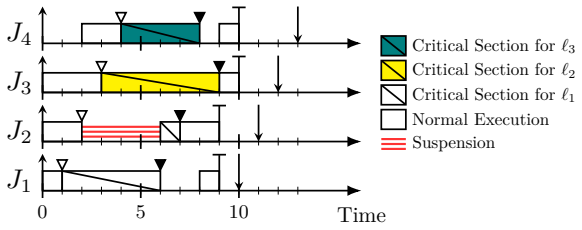


Figure 14 Example of non-preemptive blocking.

By Rule O2, the effective priority of J_j is among the top m at time t. At time t, any ready job with higher effective priority than J_j is either scheduled or unscheduled. Therefore, one of the following two cases holds: (i) at time t, each ready job with higher effective priority than J_j is scheduled; (ii) at time t, there is a ready but unscheduled job with higher effective priority than J_j.

For (i), since no job with higher effective priority than J_j is ready but not scheduled at time t, J_j is scheduled on π_q or some other available processor.

For (ii), let J_u be a job that at time t has higher effective priority than J_i, is ready, but is not scheduled (see Fig. 13(b)). We show that J_u cannot be scheduled on π_q under the lazy preemption policy. In order for J_u to be ready but unscheduled at time t, a job J_v must exist with effective priority not in the top m that executes non-preemptively at time t and cannot be preempted by J_u at time $t_e \leq t$ when J_u becomes ready and one of the top-m-priority jobs (see Fig. 13(b)). With lazy preemptions, J_u is not scheduled until J_v completes its non-preemptive execution at time t_f or its priority is raised during $[t, t_f)$ [10, §3.3.3]. Thus, J_u cannot be scheduled on π_q. Thus, J_j is scheduled on π_q or some other available processor. ◁

We now address Case 2 by showing that a processor becomes available at time t. Since $t \geq t_r$ holds, we have the following two subcases.

Subcase 2a. $t = t_r$ holds, *i.e.*, J_i issues $\mathcal{R}$ at time t. Fig. 13(b) illustrates this subcase. In this subcase, J_i must be executing on a processor π_q during time interval $[t - \varepsilon, t)$ for some $\varepsilon > 0$. Since J_j holds resource ℓ_k at time t, by Rule O2, J_i suspends at time t and π_q becomes available. By Claim 1, J_j is scheduled at time t.

Subcase 2b. $t > t_r$ holds, *i.e.*, J_i issues $\mathcal{R}$ before time t. Fig. 13(c) illustrates this subcase. In this subcase, J_i does not execute during time interval $[t - \varepsilon, t)$ for some $\varepsilon > 0$, otherwise $J_j = J_i$ holds and J_i executes non-preemptively (holding ℓ_k) at time t (and thus is not pi-blocked at time t). Therefore, because J_i is not pi-blocked immediately before t but is pi-blocked at time t, its base priority must become one of the top m at time t. Since a job's base priority does not change under G-EDF, J_i's base priority can become one of the top m only if a job J_w with higher base priority (which is among the top m base priorities) than J_i completes at time t. Since J_w completes execution at time t, a processor becomes available at time t. By Claim 1, J_j is scheduled at time t. ◀

Non-preemptive blocking. As discussed in Sec. 5, a job can incur non-preemptive blocking when a resource is released due to Rule O3. The following example illustrates this.

▶ **Example 9.** Fig. 14 depicts four jobs that are scheduled on three processors using G-EDF. J_2 suspends at time 2 as J_1 holds resource ℓ_1 at time 2. At times 3 and 4, J_3 and J_4 request resources ℓ_2 and ℓ_3, respectively, and acquire those resources. When J_1 completes its CS at time 6, J_2's request is satisfied. Although J_2's base priority is among the top three base

Algorithm 2 Procedures for an abortable CS.

Variables for transactions for resource ℓ_k:
txn : Ptr to the transaction record being currently modified
new_txn : Ptr to the transaction record used in the current transaction
free_stack_top : Ptr to the top transaction record in the *free_stack*
data : Ptr of data structure in Fig. 9a

```
1: procedure Read(data)
2:     if data.txn ≠ NULL ∧ data.txn.done then
3:         return data.new
4:     else
5:         return data.old
6:     end if
7: end procedure
8: procedure UnassignData(data)
9:     if txn ≠ NULL then
10:        if txn.done then
11:            data.old := data.new
12:        else
13:            data.new := data.old
14:        end if
15:        txn.rc1 := txn.rc1 − 1
16:        if txn.rc2 = 1 then
17:            txn.done := false
18:            txn.next := free_stack_top
19:            free_stack_top := txn
20:        end if
21:        data.txn := NULL
22:        txn.rc2 := txn.rc2 − 1
23:    end if
24: end procedure
25: procedure AssignData(data)
26:    if txn = NULL then
27:        txn := free_stack_top
28:        txn.rc2 := txn.rc2 + 1
29:        data.txn := txn
30:        free_stack_top := free_stack_top.next
31:        txn.rc1 := txn.rc1 + 1
32:    else
33:        txn.rc2 := txn.rc2 + 1
34:        data.txn := txn
35:        txn.rc1 := txn.rc1 + 1
36:    end if
37: end procedure
38: procedure FixState(data)
39:    if txn.rc1 < txn.rc2 then
40:        if txn.rc2 = 1 ∧ txn ≠ free_stack_top then
41:            txn.done := false
42:            txn.next := free_stack_top
43:            free_stack_top := txn
44:        end if
45:        data.txn := NULL
46:        txn.rc2 := txn.rc2 − 1
47:    end if
48: end procedure
```

priorities at time 6, it cannot preempt job J_4 with the lowest effective priority due to its non-preemptive execution. By Rule O3, J_2 preempts J_1. Thus, J_1 becomes pi-blocked at time 6. J_1 preempts J_4 *lazily* when it becomes preemptive at time 8.

B Details of Abortable Critical Sections

This appendix contains further details concerning abortable CSs.

The Read, UnassignData, and AssignData routines in Algorithm 2 correspond to lines 3–6, 11-22, and 24–37 of AbortableModify. The FixState routine corresponds to the code we add in the OS timer-handling code for fixing inconsistent transaction records.

Transformation rules. We begin by giving a set of rules for transforming ordinary CS code for a shared data structure ℓ_k into abortable CS code. We assume that ℓ_k consists of n_k elements in memory denoted by $M[1..n_k]$. We denote an arbitrary transaction record by tr.

R1 Each $M[i]$ is represented by the structure shown in Fig. 9(a). For each $M[i]$, $M[i].txn$ is set to NULL and $M[i].old$ contains the initial value of $M[i]$ before any CS invocation. Initially, for each tr, $tr.rc1$, $tr.rc2$, and $tr.done$ are 0, 0, and *false*, respectively. Each such tr is also in *free_stack* initially. The variable *new_txn* is declared in the CS code and is initially NULL (line 2 of AbortableModify).

R2 $M[i]$ is read by the READ procedure if it is never modified in the CS.
R3 If $M[i]$ is modified anywhere in the CS, then UNASSIGNDATA and ASSIGNDATA are called in sequence when $M[i]$ is first accessed. We refer to this as *initializing* $M[i]$.
R4 Before calling UNASSIGNDATA for $M[i]$, *txn* and *data* are set to $M[i].txn$ and $M[i]$, respectively (lines 9 and 10 of ABORTABLEMODIFY).
R5 Before calling ASSIGNDATA for $M[i]$, *txn* is set to *new_txn* (line 24 of ABORTABLEMODIFY).
R6 After ASSIGNDATA returns, *new_txn* is set to *txn* (line 36 of ABORTABLEMODIFY).
R7 After $M[i]$ is initialized, it is accessed by $M[i].new$ (line 37 of ABORTABLEMODIFY).
R8 In the end of the CS, $new\_txn.done$ is set to *true* (line 39 of ABORTABLEMODIFY).
R9 $txn, data, new\_txn$, and fields of any $M[i]$ and tr are not updated anywhere except the procedures in Alg. 2 and the places mentioned in Rules R1–R8.

B.1 Proof

We denote $M[i]$'s valid value by $M[i].valid$. $tr.nr$ denotes the number of elements referenced by tr, *i.e.*, $tr.nr = |i : M[i].txn = tr|$. We consider a task that executes a CS and *pc* denotes its program counter. We prove Invariant I and the following theorem.

▶ **Theorem 10.** *The memory requirement of resource ℓ_k's transaction records is $O(n_k)$.*

We prove Invariant I and Thm. 10 below by proving that each of (I1)–(I13) is an invariant. We first define the following predicates.

$$P_1 \equiv \forall tr : tr.rc1 = tr.rc2 = tr.nr.$$

$$P_2 \equiv txn.rc2 = txn.rc1 + 1.$$

$$P_3 \equiv txn.rc1 = txn.nr.$$

$$P_4 \equiv txn.rc2 = txn.nr.$$

$$P_5 \equiv \forall tr \neq txn : tr.rc1 = tr.rc2.$$

$$P_6 \equiv (data.txn = \text{NULL} \wedge P_3) \vee (data.txn \neq \text{NULL} \wedge P_4).$$

Invariant (I1) shows that the value stored in $tr.rc1$ and $tr.rc2$ is correct after CS is aborted/-completed. Invariants (I7) and (I8) show that tr is available for future use if $tr.nr$ is 0 and it is not overwritten, otherwise. Finally, Invariants (I11)–(I13) show that Invariant I is correct.

invariant $pc \notin \{16..22, 29..31, 34, 35, 38..46\} \implies P_1.$ (I1)

invariant $pc \in \{16..21, 30, 31, 35\} \implies P_2 \wedge P_4 \wedge P_5.$ (I2)

invariant $pc \in \{22, 29, 34, 46\} \implies P_2 \wedge P_3 \wedge P_5.$ (I3)

invariant $pc \in \{38..39\} \implies P_1 \vee (P_2 \wedge P_6 \wedge P_5).$ (I4)

invariant $pc \in \{40..45\} \implies P_2 \wedge P_6 \wedge P_5.$ (I5)

invariant $pc \in \{22..24, 25..29, 32..34\} \implies data.txn = \text{NULL}.$ (I6)

invariant $\forall tr : tr.nr = 0 \implies tr \in free\_stack.$ (I7)

invariant $\forall tr : tr \in free\_stack \wedge pc \notin \{21, 30, 39..45\} \implies tr.nr = 0.$ (I8)

invariant $pc \in \{15..37\} \implies data.old = data.valid.$ (I9)

invariant $pc \in \{40..46\} \implies data.old = data.valid.$ (I10)

invariant $\forall i : M[i].txn.done \implies M[i].new = M[i].valid.$ (I11)

invariant $\forall i : \neg M[i].txn.done \implies M[i].old = M[i].valid.$ (I12)

invariant $\forall i : M[i].txn = \text{NULL} \implies M[i].old = M[i].valid.$ (I13)

We now prove that each of (I1)–(I13) is an invariant. (I1)-(I13) hold initially by Rule R1.

For each invariant I' and any pair of consecutive states u and v, we show that if all the invariants hold at state u, then I' holds at state v. For invariants in the form of an implication, it suffices to only check those statements that may either establish the antecedent or falsify the consequent if executed while the antecedent holds.

Proof of (I1). Statements 22, 31, 35, and 46 establish the antecedent by incrementing pc. By (I3), $pc \in \{22, 46\}$ implies $P_2 \wedge P_3 \wedge P_5$. Statements 22 and 46 decrement *txn.rc2* and establish P_1. By (I2), $pc \in \{31, 35\}$ implies $P_2 \wedge P_4 \wedge P_5$. Statements 31 and 35 establish P_1 by incrementing *txn.rc1*. By (I4), statement 39 establishes the antecedent only if *txn.rc1* = *txn.rc2* holds (thus, $\neg P_2$ holds). By (I4), P_1 also holds before executing statement 39.

Statements 15, 28, and 33 can falsify the consequent, but also falsify the antecedent. ◀

Proof of (I2). The statements that may establish the antecedent are 15, 29, and 34. By (I1), $pc = 15$ implies P_1. Since statement 15 decrements *txn.rc1*, it establishes the consequent. By (I6), $pc \in \{29, 34\}$ implies $data.txn = \text{NULL}$. Thus, each of statements 29 and 34 increments *txn.nr*. Therefore, by (I3), the consequent is established.

Statements 21, 31, and 35 can falsify the consequent, but also falsify the antecedent. ◀

Proof of (I3). Statements 21, 28, 33, and 45 establish the antecedent. By (I2), Rule R4, and statement 9, $pc = 21$ implies $P_2 \wedge P_4 \wedge P_5 \wedge data.txn \neq \text{NULL}$. Thus, statement 21 decrements *txn.nr* and establishes the consequent. $pc \in \{28, 33\}$ implies P_1. Statements 28 and 33 increment *txn.rc2* and establish the consequent. We consider two cases for statement 45.

Case 1. $pc = 45 \wedge data.txn = NULL$ *holds.* By (I5), $P_2 \wedge P_6 \wedge P_5$ holds. Since $data.txn = \text{NULL}$, by the definition of P_6, P_3 holds. Thus, the consequent holds before executing statement 45. Since $data.txn = \text{NULL}$ holds before executing statement 45, it does not alter *txn.nr*.

Case 2. $pc = 45 \wedge data.txn \neq NULL$ *holds.* By (I5) and the definition of P_6, $P_2 \wedge P_4 \wedge P_5$ holds. In this case, statement 45 establishes the consequent by decrementing *txn.nr*.

Statements that can falsify the consequent also falsify the antecedent. ◀

Procedure FixState is invoked after a CS is aborted. We let *last* to denote the value of pc when the CS is aborted. Thus, *last* can be any statement in the CS.

Proof of (I4). Any statement in the CS can establish the antecedent. If $last \notin \{16..22, 29..31, 34, 35\}$, then by (I1), the consequent holds. Otherwise, by (I2) and (I3), $P_2 \wedge P_5$ holds. We now prove that P_6 also holds in this case. If $last \in \{22, 29, 34\}$, then $data.txn = \text{NULL} \wedge P_3$, thus P_6, holds by (I3) and (I6). By Rules R4 and R5 and statements 9, 27, 26, 29 and 32, $data.txn = txn \neq \text{NULL}$ holds if $last \in \{16..21, 30, 31, 35\}$. By (I2), P_4 holds if $last \in \{16..21, 30, 31, 35\}$. Therefore, $last \in \{16..21, 30, 31, 35\}$ implies $data.txn \neq NULL \wedge P_4$. Thus, $last \in \{16..22, 29..31, 34, 35\} \implies P_6$.

No statement for which the antecedent holds falsifies the consequent. ◀

Proof of (I5). Statement 39 may establish the antecedent. By (I4) and statement 39, the consequent holds. No statement for which the antecedent holds falsifies the consequent. ◀

Proof of (I6). Statement 21 establishes the antecedent, which also establishes the consequent. By Rules R3 and R9, the consequent also holds when $pc = 25$ is established, *i.e.*, AssignData is invoked. Statements 29 and 34 falsify both antecedent and consequent. ◀

Proof of (I7). By Rule R9, only statement 21 and 45 can establish the antecedent only if executed when $txn.nr = 1 \wedge data.txn = txn$. If $pc \in \{21, 45\} \wedge txn.nr = 1$, then by (I2), $pc \in \{21, 45\} \wedge txn.rc2 = 1$ holds. Since statements in {16..20} do not alter $txn.rc2$, statements in {17..19} must execute before $pc = 21 \wedge txn.rc2 = 1$ holds. Similarly, either statements {41..43} must execute or $txn = free\_stack\_top$ holds before $pc = 45 \wedge txn.rc2 = 1$ holds. These statements insert txn into the *free_stack* and statements 21 and 45 do not alter it. Therefore, the consequent is true after executing statements 21 and 45.

Statement 30 may falsify the consequent. By statement 29 and the definition of $txn.nr$, $pc = 30$ implies $txn.nr \geq 1$. Therefore, the antecedent is false for txn. ◀

Proof of (I8). Statements 21, 30, and 45 can establish the antecedent only if $txn \in free\_stack$. Assume that $txn \in free\_stack \wedge txn.nr > 0$ holds at $pc \in \{22, 46\}$ after executing statements 21 and 45. By (I7), $txn \notin free\_stack$ holds, a contradiction. Statement 30 removes txn from the *free_stack* and falsifies the antecedent.

By (I7), statements that falsify the consequent also falsify the antecedent. ◀

Proof of (I9). Statement 11 or 13 establishes the antecedent. By (I11) (resp., (I12)), statement 11 (resp. 13), establishes the consequent. By Rules R3 and R9, the consequent also holds when $pc = 25$ is established. None of the statements in {9..37} falsifies the consequent. ◀

Proof of (I10). Statement 39 can establish the antecedent only when $txn.rc1 < txn.rc2$. By (I1), $txn.rc1 \neq txn.rc2$ implies $last \in \{16..22, 29..31, 34, 35, 38..46\}$, which by (I9), implies $data.old = data.valid$. None of the statements in {40..46} falsifies the consequent. ◀

Proof of (I11). Rule R8 establishes the antecedent for each $M[i]$ associated with new_txn. By Rules R3, R5, and R6, and statements 29 and 30, $M[i].txn = new\_txn$ holds for each $M[i]$ corresponding to that CS before applying Rule R8. By Rule R7, each such $M[i]$'s *new* field is updated before applying Rule R8. Thus, the consequent holds after applying Rule R8. Statement 13 may falsify the consequent. By statement 10 and Rule R4, the antecedent is false when $pc = 13$ holds. ◀

Proof of (I12). Statements 17 and 41 establish the antecedent for each $M[i]$ associated with txn. By (I2) and statements 16 and 40, $pc \in \{17, 41\}$ implies $txn.nr = 1$. Since $txn = data.txn$ holds at $pc = 8$ (by Rule R4) and statements in {9..15} do not alter txn, $txn = data.txn$ holds before executing statement 17. Since $pc = 17$ implies $txn.nr = 1$, $pc = 17 \implies \forall M[i] \neq data : M[i].txn \neq txn$ holds. By (I9), $data.old = data.valid$ holds before and after executing statement 17. Thus, the consequent holds for txn.

data and *txn* in FixState are the same as when the transaction aborts. If $pc = 41 \wedge data.txn = \text{NULL}$ holds, then by (I5) and the definition of P_6, $txn.nr = 0$ holds. Thus, the consequent holds. If $pc = 41 \wedge data.txn \neq \text{NULL}$ holds, then by an argument similar to the above using (I5) and (I10), the consequent holds.

Statement 11 may falsify the consequent. By statement 10, the antecedent is also false. ◀

Proof of (I13). Statements 21 and 45 establish the antecedent for *data*. By (I9) and (I10), $pc \in \{21, 45\}$ implies $data.old = data.valid$. Therefore, the consequent holds.

By statement 9 and Rule R4, Statement 11 cannot falsify the consequent. ◀

Proof of Invariant I. Follows from invariants (I11)–(I13). ◀

▶ **Lemma 11.** *If free_stack initially contains n_k+1 transaction records, then it is never empty.*

Proof. Each element $M[i]$ can be associated with at most one transaction record. Thus, at most n_k transaction records are reference by some elements. Hence, at any point in time, there exists a transaction record tr with $tr.nr = 0$. By (I7), we have $tr \in free\_stack$. ◀

Proof of Thm. 10. Follows from (I7), Lemma 11, and tr's constant memory requirement. ◀