

On Lock-Free Synchronization for Dynamic Embedded Real-Time Software*

Hyeonjoong Cho
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
hjcho@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
binoy@vt.edu

E. Douglas Jensen
The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

ABSTRACT

We consider non-blocking synchronization for dynamic embedded real-time systems such as those that are subject to resource overloads and arbitrary activity arrivals. The multi-writer/multi-reader problem inherently occurs in such systems, when their activities must concurrently and mutually exclusive share data objects. We consider lock-free synchronization for this problem under the unimodal arbitrary arrival model (or UAM). UAM embodies a “stronger” adversary than most traditional arrival models. We establish the fundamental tradeoffs between lock-free and lock-based object sharing under UAM — the first such result. Our tradeoffs include analytical conditions under which activities’ accrued timeliness utility is greater under lock-free than lock-based, and the consequent upper bound on the increase in accrued utility. Our implementation experience on a POSIX RTOS reveals that the lock-free scheduling algorithm yields higher accrued utility, by as much as 65%, and critical time satisfactions, by as much as 80%, over lock-based.

1. INTRODUCTION

Embedded real-time systems that are emerging in many domains such as robotic systems in the space domain (e.g., NASA/JPL’s Mars Rover [9]) and control systems in the defense domain (e.g., airborne trackers [7]) are fundamentally distinguished by the fact that they operate in environments with dynamically uncertain properties. These uncertainties include transient and sustained resource overloads due to context-dependent activity execution times and arbitrary activity arrival patterns. Nevertheless, such systems’ desire the strongest possible assurances on activity timeliness behavior.

When resource overloads occur, meeting deadlines of all application activities is impossible as the demand exceeds the supply. The urgency of an activity is typically orthogonal to the relative importance of the activity—e.g., the most urgent activity can be the least important, and vice versa; the most urgent can be the most important, and vice versa. Hence when overloads occur, completing the most important activities irrespective of activity urgency is often desirable. Thus, a clear distinction has to be made between the urgency and the importance of activities, during overloads. (During underloads, such a distinction need not be made, because deadline-based scheduling algorithms such as EDF are optimal for those situations [12]—i.e., they can satisfy all deadlines.)

Deadlines by themselves cannot express both urgency and importance. Thus, we consider the abstraction of time/utility functions (or TUFs) [14] that express the utility of completing an application activity as a function of that activity’s completion time. We specify deadline as a binary-valued, downward “step” shaped

TUF; Figure 1(a) shows examples. Note that a TUF decouples importance and urgency—i.e., urgency is measured as a deadline on the X-axis, and importance is denoted by utility on the Y-axis.

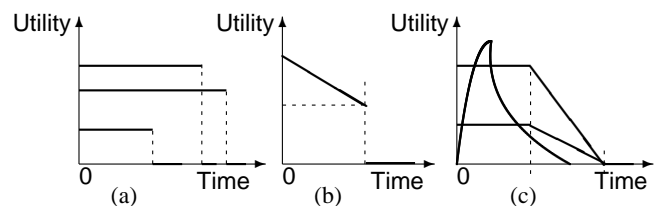


Figure 1: Example TUF Time Constraints

Many embedded real-time systems also have activities that are subject to *non-deadline* time constraints, such as those where the utility attained for activity completion *varies* (e.g., decreases, increases) with completion time. This is in contrast to deadlines, where a positive utility is accrued for completing the activity anytime before the deadline, after which zero, or infinitely negative utility is accrued. Figures 1(b)–1(c) show example such time constraints from two real applications (see [7] and references therein).

When activity time constraints are specified using TUFs, which subsume deadlines, the scheduling criteria is based on accrued utility, such as maximizing sum of the activities’ attained utilities. We call such criteria, *utility accrual* (or UA) criteria, and scheduling algorithms that optimize them, as UA scheduling algorithms.

UA algorithms that maximize summed utility under downward step TUFs (or deadlines) [8, 17, 23] default to EDF during underloads, since EDF can satisfy all deadlines during those situations. Consequently, they obtain the maximum total utility during underloads. When overloads occur, they favor activities that are more important (since more utility can be attained from them), irrespective of their urgency. Thus, UA algorithms’ timeliness behavior subsume the optimal timeliness behavior of deadline scheduling.

1.1 Shared Data and Synchronization

Most embedded real-time systems involve mutually exclusive, concurrent access to shared data objects, resulting in contention for those objects. Resolution of the contention directly affects the system’s timeliness behavior. Mechanisms that resolve such contention can be broadly classified into: (1) lock-based schemes—e.g., [8, 19, 23]; and (2) non-blocking schemes including wait-free protocols (e.g., [4, 13, 15]) and lock-free protocols (e.g., [2]).

Lock-based protocols have several disadvantages such as serialized access to shared objects, resulting in reduced concurrency [2]. Further, many lock-based protocols typically incur additional runtime overhead due to protocol (or scheduler) activations that occur when activities request previously locked shared objects, which

*This work was supported by the US Office of Naval Research under Grant N00014-00-1-0549 and The MITRE Corporation under Grant 52917.

trigger the protocol (or the scheduler) [8, 19]. Another disadvantage is the possibility of deadlocks that can occur when lock holders crash, causing indefinite starvation to blockers. Many (real-time) lock-based protocols also require a-priori knowledge of the ceilings of locks [19], which may be difficult to obtain for dynamic applications. OS data structures (e.g., semaphore control blocks) must be a-priori updated with that knowledge, resulting in reduced flexibility [2]. These drawbacks have motivated research on wait-free and lock-free object sharing in embedded real-time systems.

Wait-free protocols typically use multiple buffers for the shared object. For the single-writer/multi-reader (or SWMR) problem, the key idea is to use as many buffers as the maximum number of times the readers can be preempted by the writer. The maximum number of preemptions of a reader bounds the number of times the writer can update the object while the reader is reading. Thus, by using as many buffers as the worst-case number of preemptions of readers, the readers and the writer can continuously read and write in different buffers, respectively, and thereby avoid interference.

Lock-free protocols allow readers to concurrently read while the writer is writing (without acquiring locks), but the readers check whether their reading was interfered by the writer. If so, they read again. Thus, a reader continuously reads, checks, and retries until its read becomes successful. Since a reader's worst-case number of retries depends upon the worst-case number of times the reader is preempted by the writer, the additional execution-time overhead incurred for the retries is bounded by the number of preemptions.

Both wait-free and lock-free sharing incur additional costs with respect to lock-based sharing. Wait-free protocols generally incur additional space costs due to their multiple buffer usage, which is infeasible in many small-memory embedded real-time systems. Lock-free protocols generally incur additional time costs due to their retries, which is antagonistic to timeliness optimization.

Prior research have shown how to mitigate these space and time costs. An excellent survey of this prior research can be found in [13]. To provide context for our work, we summarize some important efforts here: In [2], Anderson *et al.* show how to bound lock-free retries through judicious scheduling. In [15], Kopetz *et al.* present one of the earliest wait-free protocols, which was later improved by Chen *et al.* in [4], and subsequently by Huang *et al.* in [13] and by Cho *et al.* in [5]. Cho *et al.* later use their protocol [5] for wait-free synchronization under UA scheduling in [6]. In [21], Sundell *et al.* describe a wait-free protocol for the multi-writer/multi-reader (or MWMR) problem.

1.2 Contributions

In this paper, we focus on *dynamic* embedded real-time systems. By dynamic systems, we mean those that are subject to resource overloads due to context-dependent activity execution times and arbitrary activity arrivals. To account for the variability in activity arrivals, we describe arrival behaviors using the unimodal arbitrary arrival model (or UAM) [11]. UAM specifies the maximum number of activity arrivals that can occur during any time interval — the arrival frequency within the time interval is arbitrary. Consequently, the model subsumes most traditional arrival models (e.g., frame-based, periodic, sporadic) as special cases.

When activities in such systems concurrently and mutually exclusively share data objects, the MWMR problem always occurs. This is because, under the UAM, multiple jobs of a single writer task can be simultaneously pending in the ready queue and be requesting the same shared object. This situation will not occur under a periodic task model during under-loads if considering the SWMR problem.

We consider lock-free sharing for the MWMR problem under the

UAM. We consider lock-free, as opposed to wait-free, due to the following reasons: Upper bounds on the number of buffers needed for wait-free are typically established under the assumption that read/write operations know the identity of tasks invoking the operations [4, 13, 15]. However, under the UAM, multiple jobs of the same task inherit the same task identity, and can access a shared object. Consequently, for the UAM, wait-free sharing must be anonymous in the sense that the read/write operations should not require knowledge of the identity of the jobs. Most wait-free solutions for the MWMR problem, however, are not anonymous [3, 10, 22].

Further, wait-free sharing for the MWMR problem is inherently space-inefficient. Theoretically, with n readers and m writers, the minimum number of needed buffers is $n + m + 1$ for the MWMR problem [20]. However, most wait-free solutions for the MWMR problem need $n \times m$ number of atomic buffers, where each atomic buffer solves the single-writer/single-reader (SWSR) problem [10, 22]. The atomicity of the buffer is ensured by using multiple buffers, which further increases the number of needed buffers. To make things worse, the UAM allows greater number of concurrent operations than more restrictive arrival models like the periodic considered in past works [13, 15], resulting in even greater number of buffers.

These problems do not arise with lock-free sharing. Several lock-free objects exist for the MWMR problem [18], and lock-free is inherently anonymous and space-efficient. Thus, lock-free sharing is attractive for dynamic real-time systems. However, past work on lock-free sharing upper bounds the retries under restrictive arrival models like periodic [2] — retry bounds under the UAM are not known. If this can be established, then the MWMR problem which occurs under the UAM can be solved using lock-free.

We precisely do so in this paper. We consider the UA criteria of maximizing the summed activity utility, while allowing most TUF shapes including step and non-step shapes, and mutually exclusive concurrent object sharing. We focus on the Resource-constrained Utility Accrual (or RUA) scheduling algorithm [23], as it is the only algorithm for that model. RUA allows arbitrarily-shaped TUFs and concurrent object sharing using locks. For the special case of step TUFs, no object sharing, and under-loads, RUA defaults to EDF.

We develop the lock-free version of RUA, and derive the upper bound on lock-free RUA's retries under the UAM — the first ever retry bound under a non-periodic model. Since lock-free sharing incurs additional time overhead due to the retries (than lock-based), we establish the conditions under which activity sojourn times are shorter under lock-free RUA than under lock-based RUA, for the UAM. From this result, we establish the maximum increase in activity utility that is possible under lock-free RUA over lock-based.

We also ask a fundamental question: If an anonymous and space-optimal wait-free solution for the MWMR problem under the UAM can be developed (to the best of our knowledge, no such solution exists), then what is the tradeoff between that solution and lock-based and lock-free solutions? Answering this is important, as wait-free will also yield utility increases, but does so by sacrificing space. We answer this question by considering a theoretically abstract wait-free solution. We then establish the maximum increase in utility with wait-free RUA over lock-based and lock-free ones.

We implement lock-free and lock-based RUA on a POSIX RTOS. Our implementation measurements reveal that lock-free RUA yields significant increase in accrued utility, by as much as 65%, and critical time satisfactions, by as much as 80%, over lock-based.

Thus, the paper's contributions include new results that facilitate, for the first time, non-blocking synchronization for embedded real-time systems subject to overloads and arbitrary activity arrivals. The results include lock-free retry upper bound under the UAM,

lock-free TUF/UA scheduling, and upper bound on utility increase under lock-free and abstract wait-free. We are not aware of any other work on non-blocking synchronization for the UAM.

The rest of the paper is organized as follows: In Section 2, we derive the upper bound on retries under lock-free RUA. We compare lock-free and lock-based RUA, and establish the tradeoffs between the two in Section 3. In Section 4, we establish the tradeoffs between lock-based, lock-free, and abstract wait-free RUA. In Section 5, we discuss our implementation experience and report our measurements. Finally, we conclude the paper in Section 6.

2. BOUNDING RETRIES UNDER UAM

To derive lock-free RUA’s retry bound under the UAM, we first describe our task model, and overview lock-based RUA. We then derive the upper bound on the number of preemptions under UA schedulers. This result is then used to develop the lock-free retry bound. These are discussed in the subsections that follow.

2.1 Task Model

Figure 2 shows the three dimensions of the task model that we consider in the paper. The first dimension is the arrival model. We consider the UAM, which is more relaxed than the periodic model, but more regular than the aperiodic model. Hence it falls in between these two ends of the (regularity) spectrum of the arrival dimension. For a task T_i , its arrival using UAM is defined as a tuple $\langle l_i, a_i, W_i \rangle$, meaning that the maximal number of job arrivals of the task during any sliding time window W_i is a_i and the minimal number is l_i [11]. Jobs may arrive simultaneously. The periodic model is a special case of UAM with $\langle 1, 1, W_i \rangle$.

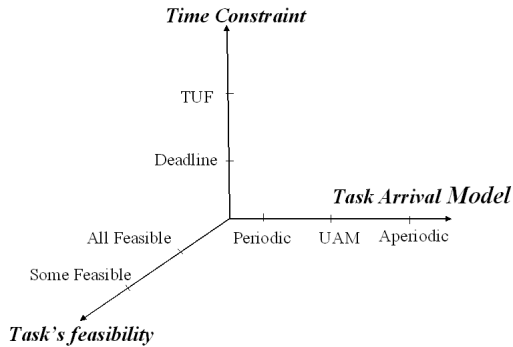


Figure 2: Three Dimensions of Task Model

We refer to the j^{th} job (or invocation) of task T_i as $J_{i,j}$. The basic scheduling entity that we consider is the job abstraction. Thus, we use J to denote a job without being task specific, as seen by the scheduler at any scheduling event.

A job’s time constraint, which forms the second dimension, is specified using a TUF. A TUF subsumes deadline as a special case—i.e., binary-valued, downward step TUF. Jobs of a task have the same TUF. We use $U_i(\cdot)$ to denote task T_i ’s TUF. Thus, completion of task T_i at a time t will yield a utility $U_i(t)$.

TUFs can take arbitrary shapes, but must have a (single) “critical time.” Critical time is the time at which the TUF has zero utility. For the special case of deadlines, critical time corresponds to the deadline time. We denote the critical time of task i ’s $U_i(\cdot)$ as C_i , and assume that $C_i \leq W_i$.

The third dimension is feasibility. Feasibility includes underload situations, during when all tasks can be completed before their critical times, and overloads, during when only a subset of the tasks (including possibly none) can be done so.

Our model includes overloads, the UAM, and TUFs.

2.2 Overview of Lock-Based RUA

RUA [23] considers activities subject to arbitrarily shaped TUF time constraints and concurrent object sharing under mutual exclusion constraints. The algorithm’s scheduling events include job arrivals, job departures, and lock, and unlock requests. When RUA is invoked, it first builds each job’s dependency list—that arises due to mutually exclusive object sharing—by following the chain of object request and ownership. The algorithm then checks for deadlocks by detecting the presence of a cycle in the object/resource graph—a necessary condition for deadlocks. Deadlocks are resolved by aborting that job in the cycle, which will likely contribute the least utility.

After handling deadlocks, RUA examines jobs in the order of non-increasing potential utility densities (or PUDs). The PUD of a job is the ratio of the expected job utility to the remaining job execution time, and thus measures the job’s return of investment. The algorithm inserts a job and its dependents into a tentative schedule, in the order of their critical times, earliest critical time first. The insertion is done by respecting the jobs’ dependencies.

After insertion, RUA checks the schedule’s feasibility with respect to satisfying all job critical times. If the schedule is infeasible, the inserted job and its dependents are rejected. The algorithm repeats the process until all jobs are examined, and selects the job at the schedule head for execution.

If a job’s critical time is reached and its execution has not been completed, an exception is raised, which is also a scheduling event, and the job is immediately aborted.

During under-loads, with step TUFs, and no object sharing, it is easy to see that RUA will never reject a job, as all jobs will be feasible. The resulting output schedule will be a critical time (or deadline)-ordered schedule, yielding the maximum possible total utility. During overloads, the algorithm will produce a feasible schedule by rejecting some (low PUD) jobs and thereby seeks to obtain as high total utility as possible.

2.3 Preemptions Under UA Schedulers

Under fixed priority schedulers such as RM, a lower priority job can be preempted at most once by each higher priority job if no resource dependency (that arises due to concurrent object sharing) exists. This is because a job does not change its execution eligibility until its completion time. However, under UA schedulers such as RUA, execution eligibility of a job dynamically changes.

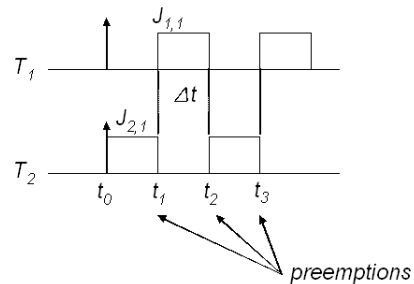


Figure 3: Mutual Preemption Under RUA

In Figure 3, assume that two jobs $J_{1,1}$ and $J_{2,1}$ arrives at time t_0 . In the figure, $J_{2,1}$ occupies CPU at time t_0 and is preempted by $J_{1,1}$ at t_1 . Subsequently, $J_{1,1}$ is preempted by $J_{2,1}$ at t_2 , which does not happen under RM scheduling. Here, we assume that some events which invoke the scheduler happens at time t_1 and t_2 , caused by other jobs in the system. Under RUA, a simple condition causing this *mutual preemption*, where a job which has preempted another job can be subsequently preempted by the other job, can be deter-

mined as follows:

The PUD of a task i at time t is given by $\frac{U_i(c_i)}{c_i - t}$, where c_i is task i 's expected completion time, $U_i(c_i)$ is task i 's utility at time c_i , and t is the current time. Suppose that $J_{1,1}$ and $J_{2,1}$ have the same critical time and only one of them is feasible. RUA will now select the higher PUD job among $J_{1,1}$ and $J_{2,1}$. Assuming that RUA is invoked at times t_0 , t_1 , and t_2 by the arrival of other low PUD jobs, PUDs of jobs $J_{1,1}$ and $J_{2,1}$ can be computed as follows:

$$\begin{aligned} t_0: PUD_1(t_0) &= \frac{U_1(c_1)}{c_1 - t_0} < PUD_2(t_0) = \frac{U_2(c_2)}{c_2 - t_0} \\ t_1: PUD_1(t_1) &= \frac{U_1(c_1 + \Delta t)}{c_1 - t_0} > PUD_2(t_1) = \frac{U_2(c_2)}{c_2 - (t_0 + \Delta t)} \\ t_2: PUD_1(t_2) &= \frac{U_1(c_1 + \Delta t)}{c_1 - (t_0 + \Delta t)} < PUD_2(t_2) = \frac{U_2(c_2 + \Delta t)}{c_2 - (t_0 + \Delta t)} \end{aligned}$$

Since $PUD_2(t_0) < PUD_2(t_1)$, $PUD_1(t_1)$ should be greater than $PUD_1(t_0)$, which means that $U_1(c_1 + \Delta t)$ should be greater than $U_1(c_1)$. In the same way, since $PUD_1(t_1) < PUD_1(t_2)$, $U_2(c_2 + \Delta t)$ should be greater than $U_2(c_2)$.

As indicated previously, one of the simple cases for RUA to allow the mutual preemption is when TUFs of two jobs are increasing. In other words, two jobs scheduled by RUA may preempt each other repeatedly if the jobs have increasing TUFs. This potential mutual preemption distinguishes RUA from traditional priority schedulers such as RM, where a job can be preempted by another job at most once. Hence, the maximum number of preemptions under RM that a job may suffer can be bounded by the number of releases of other jobs during a given time interval [1].

However, this is not true with RUA, as one job can be preempted by another job more than once. Therefore, the maximum number of preemptions that a job may experience under RUA is bounded by the number of events that invoke the scheduler. This is also true with other UA schedulers such as [8, 17], because it is impossible for more preemptions to occur than scheduling events.

LEMMA 1 (PREEMPTIONS UNDER UA SCHEDULER). *During a given time interval, a job scheduled by a UA scheduler can experience preemptions by other jobs at most the number of the scheduling events that invoke the scheduler.*

PROOF. Assume that preemptions happen more than the scheduling events during a given time interval. It means that preemptions can occur without invoking the scheduler, which is not true. \square

Lemma 1 helps compute the upper bound of the number of retries on lock-free objects for our task model.

2.4 Bounded Retry Under UAM

Under our task model, jobs arrive under the UAM, are subject to TUF time constraints, and sufficient CPU time may not always be available to complete all jobs before their critical times. We now bound lock-free RUA's retries under this model.

THEOREM 2 (LOCK-FREE RETRY BOUND UNDER UAM). *Suppose that jobs arrive under the UAM < 1 , $a_i, W_i >$ and are scheduled by RUA. When a job J_i accesses more than one lock-free object, the total number of retries f_i of J_i is bounded as:*

$$f_i \leq 3a_i + \sum_{j=1, j \neq i}^N 2a_j \left(\left\lceil \frac{C_i}{W_j} \right\rceil + 1 \right)$$

where N is the number of tasks.

PROOF. In Figure 4, J_i is released at time t_0 and has the absolute critical time $(t_0 + C_i)$. After the critical time, J_i will not exist in the ready queue, because it will be either completed by that time or aborted by RUA. Thus, by Lemma 1, the number of retries

of J_i is bounded by the maximum number of all scheduling events that occur within the time interval $[t_0, t_0 + C_i]$. The scheduling events that J_i suffers can be categorized into those occurring from other tasks, $T_j, j \neq i$ and those occurring from T_i . We consider these two cases:

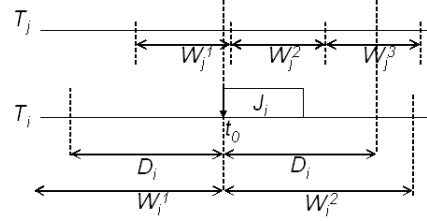


Figure 4: Interferences Under UAM

Case 1 (Scheduling events from other tasks): To account for the worst-case event occurrence, we assume that all instances of T_j in the window W_j^1 are released right after time t_0 , and all instances of T_j in the window W_j^3 are released before time $(t_0 + C_i)$. Thus, the maximum number of releases of T_j within $[t_0, t_0 + C_i]$ is $\lceil C_i/W_j \rceil + 1$. It also holds when $W_j > C_i$ as $\lceil C_i/W_j \rceil + 1 = 2$. All jobs of T_j before the first window W_j must depart either by completion or by abortion before t_0 . All released jobs must be completed or aborted, so that the number of scheduling events that a job can create is at most 2. Hence, $a_j (\lceil C_i/W_j \rceil + 1)$ is multiplied by 2.

Case 2 (Scheduling events from the same task): In the worst-case, all jobs of T_i are released and completed within $[t_0, t_0 + C_i]$, which results in at most $2a_i$ events. T_i 's jobs, which are released during $[t_0 - C_i, t_0]$ also cause events within $[t_0, t_0 + C_i]$ by completion. Thus, the total number of events that are possible is $3a_i$.

The sum of case 1 and case 2 is the upper bound on the number of events. It is also the maximum number of total retries of J_i 's objects. \square

Note that f has nothing to do with the number of lock-free objects in J_i in Theorem 2, even when J_i accesses more than one lock-free object. This is because no matter how many objects J_i accesses, f cannot exceed the number of events. Further, even if J_i accesses a single object, the retry can occur only as many times as the number of events.

Theorem 2 also implies that the sojourn time of J_i is bounded. The sojourn time of J_i is computed as the sum of J_i 's execution time, the interference time by other jobs, the lock-free object accessing time, and f retry time.

3. LOCK-BASED VERSUS LOCK-FREE

We now formally compare lock-based and lock-free sharing by comparing job sojourn times. We do so, as sojourn times directly determine critical time-misses and accrued utility. The comparison will establish the tradeoffs between lock-based and lock-free sharing: Lock-free is free from blocking times on shared object accesses and scheduler activations for resolving those blockings, while lock-based suffers from these. However, lock-free suffers from retries, while lock-based does not.

We introduce some notations, which are the same as that in [1]. We assume that all accesses to lock-based objects require r time units, and to lock-free objects require s time units. The computation time c_i of a job J_i can be written as $c_i = u_i + m_i \times t_{acc}$, where u_i is the computation time not involving accesses to shared objects; m_i is the number of shared object accesses by J_i ; and t_{acc} is the maximum computation time for any object access—i.e., r for lock-based objects and s for lock-free objects.

The worst-case sojourn time of a job with lock-based objects is $u_i + I + r \cdot m_i + B$, where B is the worst-case blocking time and I is the worst-case interference time. In [23], it is shown that a job J_i under RUA can be blocked for at most $\min(m, n)$ times, where n is the number of jobs that could block J_i and m is the number of resources (or objects) that can be used to block J_i . Therefore, B can be computed as $r \cdot \min(m, n)$.

On the other hand, the worst-case sojourn time of a job with lock-free objects is $u_i + I + s \cdot m_i + R$, where R is the worst-case retry time. R can be computed as $s \cdot f$ by Theorem 2. Thus, the difference between $r \cdot m_i + B$ and $s \cdot m_i + R$ is the sojourn time difference between lock-based and lock-free sharing.

THEOREM 3 (LOCK-BASED VERSUS LOCK-FREE SOJOURN). *Let jobs arrive under the UAM and be scheduled by RUA. Now, if*

$$\begin{cases} \left(\frac{s}{r} < \frac{2}{3} \right) \wedge \left(\frac{1}{2r-1} (3a_i + 2x) < m_i < 2a_i + x \right), & m_i \leq n \\ \left(\frac{s}{r} < 1 \right) \wedge \left(\frac{s}{r} (3a_i + 2x) < (1 - \frac{s}{r}) m_i + n \right), & m_i > n, \end{cases}$$

then the sojourn time of job J_i with lock-free objects is shorter than with lock-based objects, where $x = \sum_{j=1, j \neq i}^N a_j \left(\left\lceil \frac{C_j}{W_j} \right\rceil + 1 \right)$.

PROOF. Let X denote $r \cdot m_i + r \cdot \min(m_i, n)$ and Y denote $s \cdot m_i + s \cdot f$ to compare sojourn times. n is bounded by the total number of jobs that may happen during executing J_i so that n is less than $2a_i + \sum_{j=1, j \neq i}^N a_j \left(\left\lceil C_j/W_j \right\rceil + 1 \right) = 2a_i + x$. We now search for the condition when lock-free sharing yields shorter sojourn times than lock-based, which means $X > Y$. There are two cases:

Case 1: When $m_i \leq n$, X becomes $2r \cdot m_i$. Y is $s(m_i + 3a_i + 2x)$. Therefore:

$$Y = \frac{s}{2r} X + s(3a_i + 2x), X \leq X_1 = 2r(2a_i + x).$$

The condition that leads to $X > Y$ can be derived only when $\frac{s}{2r} < 1$. Otherwise, Y is always greater than X , which means that the sojourn time with lock-free objects is greater than that with lock-based objects. Assuming $\frac{s}{2r} < 1$, X and Y become the same when $X_0 = \frac{2rs(3a_i + 2x)}{2r-s}$. Hence, when $X_0 < X < X_1$, X is greater than Y . This leads to:

$$\frac{1}{\frac{2r}{s} - 1} (3a_i + 2x) < m_i < (2a_i + x).$$

Case 2: When $m_i > n$, X becomes $r(m_i + n)$ and we can obtain $Y = \frac{s}{r} X + s(3a_i + 2x - n)$. To make $X > Y$, $\frac{s}{r}$ should be less than 1. Otherwise, the sojourn time of jobs with lock-based objects is always less than that with lock-free objects. X converges to Y at $X_0 = \frac{rs(3a_i + 2x - n)}{r-s}$. Therefore,

$$\frac{rs(3a_i + 2x - n)}{r-s} < r(m_i + n).$$

This inequality leads to:

$$\frac{s}{r} (3a_i + 2x) < \left(1 - \frac{s}{r}\right) m_i + n.$$

□

Theorem 3 shows that at least $\frac{s}{r} < \frac{2}{3}$ for jobs to obtain a shorter sojourn time under lock-free. While r includes the time of the scheduler invocations, s does not. In [1], Anderson *et al.* show that s is typically much smaller than r in comparison with various lock-free objects. Especially, RUA's asymptotic time complexity is $O(n^2 \log n)$, where n is the number of jobs [23]. Moreover, when a deadlock occurs, its time complexity increases to $O(n^4)$

due to deadlock detection and resolution. This high time complexity makes r much longer. However, for the most simple lock-free objects, such as buffers, queues, and stacks, s is much smaller.

COROLLARY 4 (SPECIAL CASE SOJOURN). *Let jobs arrive under the UAM and be scheduled by RUA. Let r be significantly larger than s . Now, if:*

$$\begin{cases} 0 < m_i < 2a_i + x, & m_i \leq n \\ \text{always,} & m_i > n, \end{cases}$$

then the sojourn time of job J_i with lock-free objects is shorter than that with lock-based objects, where $x = \sum_{j=1, j \neq i}^N a_j \left(\left\lceil \frac{C_j}{W_j} \right\rceil + 1 \right)$.

PROOF. When r is significantly larger than s , $\frac{s}{r}$ converges to 0. Thus, Theorem 3 simplifies. □

Shorter sojourn times always yields higher utility with non-increasing TUFs, but not always with increasing TUFs. However, it is likely to improve performance at system level because each job can save more CPU resources for other jobs to execute.

As r is larger than s , lock-free sharing is more likely to perform better than lock-based sharing, according to Corollary 4. Further, when m_i is larger than n , the sojourn time of a job with lock-free object always outperforms its lock-based counterpart. Thus, lock-free sharing is very attractive for UA scheduling as most UA schedulers' object sharing mechanisms have higher time complexity.

Shorter sojourn times for a job under lock-free sharing will only increase the job's accrued utility under non-increasing TUFs. However, this does not directly imply that lock-free sharing will yield higher *total* accrued utility than lock-based sharing. This is because, Theorem 3 does not reveal anything regarding aggregate system-level performance, but only job-level performance. Since RUA's objective is to maximize total utility, we now compare lock-based and lock-free sharing in terms of accrued utility ratio (or AUR). AUR is the ratio of the actual accrued total utility to the maximum possible total utility.

LEMMA 5 (LOCK-BASED VERSUS LOCK-FREE AUR). *Let jobs arrive under the UAM and be scheduled by RUA. If all jobs are feasible and their TUFs are non-increasing, then the difference in AUR between lock-free and lock-based sharing, $\Delta AUR = AUR_{lf} - AUR_{lb}$ is:*

$$\sum_{i=1}^N \frac{U_{i,lf}(s_{0,lf} + R) - U_{i,lb}(s_{0,lb})}{U_i(0)} \leq \Delta AUR \leq \sum_{i=1}^N \frac{U_{i,lf}(s_{0,lf}) - U_{i,lb}(s_{0,lb} + B)}{U_i(0)}$$

where $U_i(t)$ is task i 's utility at time t , $s_{0,lf} = u_i + I + s \cdot m_i$, $s_{0,lb} = u_i + I + r \cdot m_i$, and N is the number of tasks.

PROOF. The proof is straightforward. Both the blocking time B and the retry time R are computed for the worst-case. In the best-case, both are zero. With non-increasing TUFs, job sojourn times can be directly mapped to accrued utility. Since all jobs are assumed to be feasible and hence will not be aborted, the AUR can be calculated from each job of the task. □

The fact that B and R are the worst-case values make it highly likely for lock-free sharing to outperform lock-based sharing. Even when jobs access several shared objects, it is highly likely that there may be no or a few interferences or retries. Such few interferences are very likely to actually occur because r and s are significantly smaller than job execution times. Lock-based sharing, on the contrary, will invoke the scheduler for these cases, since object access is an automatic scheduling event, resulting in increased overhead.

4. ABSTRACT WAIT-FREE SHARING

As mentioned previously, to the best of our knowledge, no anonymous and space-optimal wait-free solution for the MWMR problem (i.e., one that requires $n + m + 1$ buffers) exists. But if such a solution can be developed in the future, then that solution will also yield increases in activity utility, but will do so by sacrificing space through its multiple buffer usage. To understand the utility increase that can be achieved with wait-free and establish the consequent tradeoffs with respect to lock-based and lock-free, we consider an *abstract* anonymous and space-optimal wait-free solution for the MWMR problem. The solution is only a theoretical abstraction.

We regard that this abstract wait-free solution only requires the optimal $n + m + 1$ number of buffers, where n is the number of readers and m is the number of writers. Further, the solution allows readers/writers to access the object anonymously. The computation time for accessing the wait-free buffer is denoted as w and job i 's sojourn time with the wait-free buffer is given by $u_i + I + w \cdot m_i$.

THEOREM 6 (WAIT-FREE BUFFERS UNDER UAM). *Suppose that jobs arrive under the UAM $< 1, a_i, W_i >$ and are scheduled by RUA. Then, the abstract wait-free solution requires $\sum_{i=1}^N a_i + 1$ number of buffers, where N is the number of tasks.*

PROOF. At any given time, at most a_i number of jobs can exist for task i , as all jobs are released according to the UAM, and RUA will either complete a job or abort the job by W_i since $C_i \leq W_i$. Each job can be a reader or a writer. For a single wait-free object, a job can be reading or writing the object at any given time during its execution. Thus, task i can have at most a_i readers or writers at any time. The minimum buffer size is $n + m + 1$, where the summation of readers and writers, $n + m$ is $\sum_{i=1}^N a_i$. \square

Although wait-free sharing avoids lock-based sharing's blockings' and lock-free's retries, it does not mean that sojourn times under wait-free is always shorter than that under lock-based and lock-free. This is because, sojourn times under wait-free also depend on w , which can be long, as is in many wait-free approaches [21, 22]. Similar to Section 3, we now compare wait-free sharing with its lock-based and lock-free counterparts, in terms of AUR.

LEMMA 7 (LOCK-BASED VERSUS WAIT-FREE AUR). *Let jobs arrive under the UAM and be scheduled by RUA. If all jobs are feasible and their TUFs are non-increasing, then the difference in AUR between wait-free and lock-based sharing, $\Delta AUR = AUR_{wf} - AUR_{lb}$ is:*

$$\sum_{i=1}^N \frac{U_{i,wf}(s_{0,wf}) - U_{i,lb}(s_{0,lb})}{U_i(0)} \leq \Delta AUR \leq \sum_{i=1}^N \frac{U_{i,wf}(s_{0,wf}) - U_{i,lb}(s_{0,lb} + B)}{U_i(0)}$$

where $U_i(t)$ is task i 's utility at time t , $s_{0,wf}$ is $u_i + I + w \cdot m_i$, $s_{0,lb}$ is $u_i + I + r \cdot m_i$, and N is the number of tasks.

PROOF. The blocking time B is computed for the worst-case, so it is zero in the best-case. With non-increasing TUFs, job sojourn times can be directly mapped to accrued utility. Since all jobs are assumed to be feasible and hence will not be aborted, the AUR can be calculated from each job of the task. \square

LEMMA 8 (LOCK-FREE VERSUS WAIT-FREE AUR). *Let jobs arrive under the UAM and be scheduled by RUA. If all jobs are feasible and their TUFs are non-increasing, then the difference in AUR*

between wait-free and lock-free sharing, $\Delta AUR = AUR_{wf} - AUR_{lf}$ is:

$$\sum_{i=1}^N \frac{U_{i,wf}(s_{0,wf}) - U_{i,lf}(s_{0,lf})}{U_i(0)} \leq \Delta AUR \leq \sum_{i=1}^N \frac{U_{i,wf}(s_{0,wf}) - U_{i,lf}(s_{0,lf} + R)}{U_i(0)}$$

where $U_i(t)$ is task i 's utility at time t , $s_{0,wf}$ is $u_i + I + w \cdot m_i$, $s_{0,lf}$ is $u_i + I + s \cdot m_i$, and N is the number of tasks.

PROOF. The proof is similar to that of Lemma 7, except that B is replaced with R .

\square

5. IMPLEMENTATION EXPERIENCE

We implemented lock-based RUA and lock-free RUA in the *meta-scheduler* scheduling framework [16], which allows middleware-level real-time scheduling atop POSIX RTOSes. We used QNX Neutrino 6.3 RTOS running on a 500MHz, Pentium-III processor in our implementation. In our study, we used the well-known lock-free queue and the lock-based queue in [18].

5.1 Object Access Times

One of the critical elements that affect the tradeoff between lock-based and lock-free sharing is the time needed for object access—i.e., lock-based object access time r , and lock-free object access time s . We measure r and s for lock-based and lock-free RUA, respectively, with a 10 task set. Each measurement is an average of approximately 2000 samples.

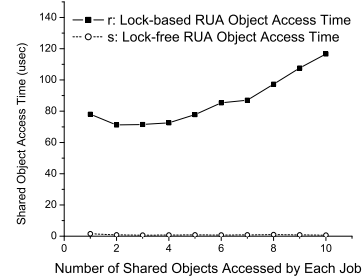


Figure 5: Lock-Based and Lock-Free Shared Object Access Time

Figure 5 shows r and s under increasing number of shared objects accessed by jobs. From the figure, we observe that r is significantly larger than s . Note that r includes the time for lock-based RUA's resource sharing mechanism.

When $r \gg s$, Theorem 3 implies that lock-free sharing is likely to perform better than lock-based sharing. Furthermore, when the number of objects m_i increases, it increases the likelihood for satisfying Corollary 4's Case 2 and lock-free sharing becomes increasingly advantageous than lock-based consequently.

5.2 Critical Time-Miss Load

The impact of r and s on lock-based RUA's and lock-free RUA's performance, respectively, can be measured by evaluating the load at which the schedulers miss task critical times. We measure this using a metric called *Critical time-Miss Load* (or CML). The CML of a scheduler is defined as the approximate load *after* which the scheduler begins to miss task critical times. We define approximate

load as $AL = \sum_{i=1}^n u_i/C_i$, where u_i is the task computation time excluding shared object access time, and C_i is the task critical time.

We exclude object access time in this load definition, because an ideal implementation of objects for synchronization must have negligibly small – almost zero – object access time. If so, the implementation is ideal in the sense that the scheduler’s performance with the (ideal) implementation is the same as that under no object sharing. For no object sharing, RUA’s CML is 1 (or 100%) if RUA’s overhead is almost zero, as the algorithm defaults to EDF for that case, and thereby will miss critical times only during overloads.

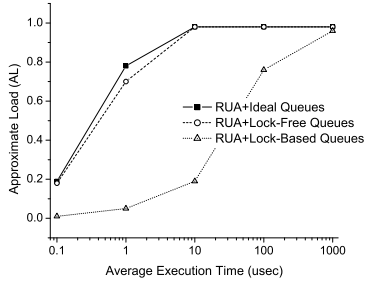


Figure 6: Critical Time Miss Load

We consider a task set of 10 tasks, accessing 10 shared queues, and measure the CML of lock-free RUA, lock-based RUA, and ideal RUA under increasing average job execution times. Figure 6 shows the results. We observe that lock-free RUA yields almost the same CML as that of ideal RUA, as it exploits the eliminated blocking times and achieves almost the same performance of RUA without object sharing. Note that the ideal queue and RUA achieve the CML of 1, only at $\approx 10usec$ of average execution time. This is because of the algorithm’s overhead for scheduling. (RUA’s and EDF’s CML of 1 is valid at zero job execution times under the assumption of no system overheads, which is not true in practice).

On the other hand, lock-based RUA’s CML converges to 1, only at ≈ 1 millisecond. This is precisely because of lock-based RUA’s complex operations for resolving jobs’ contention for object locks and consequent higher overhead, as manifested by its higher asymptotic complexity and higher object access times in Figure 5.

5.3 Accrued Utility and Critical Time-Meets

We now measure the AUR and the critical time-meet ratio (CMR) of lock-free RUA and lock-based RUA for average job execution times in the range of $30usec - 1000usec$. CMR is the ratio of the number of tasks that meet their critical times to the total number of task releases. We consider a task set of 10 tasks, accessing 10 shared queues. Each experiment is repeated to obtain AUR and CMR averages from more than 5000 task arrivals. We consider two classes of TUF shapes in this study: step and a heterogenous class including step, parabolic, and linearly-decreasing shapes.

Figures 7 and 8 show the AUR and CMR of lock-based and lock-free RUA for step TUFs and heterogenous TUFs, respectively, during under-loads ($AL \approx 0.4$), under increasing number of shared objects. Figures 9 and 10 show similar results during overloads ($AL \approx 1.1$).

As expected, the figures show that the AUR and CMR of lock-based RUA sharply decreases, eventually reaching 0% during overloads, as the number of objects increases. This is because, as the number of objects increases, greater number of task blockings’ occurs, due to the large r , resulting in increased sojourn times, critical time-misses, and consequent abortions.

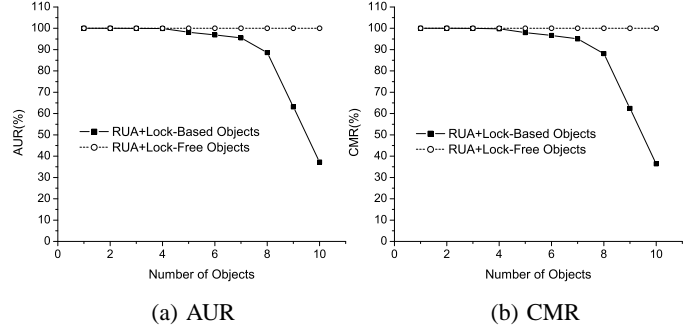


Figure 7: AUR/CMR During Underload, Step TUFs

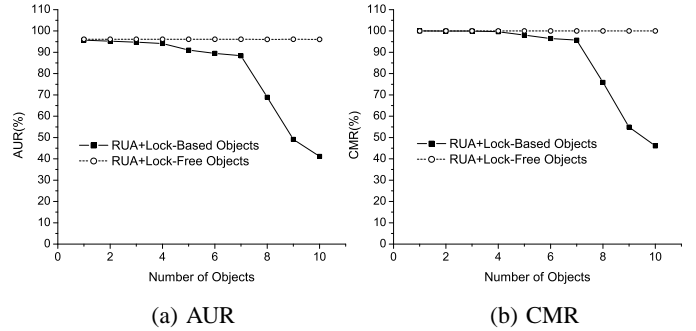


Figure 8: AUR/CMR During Underload, Heterogeneous TUFs

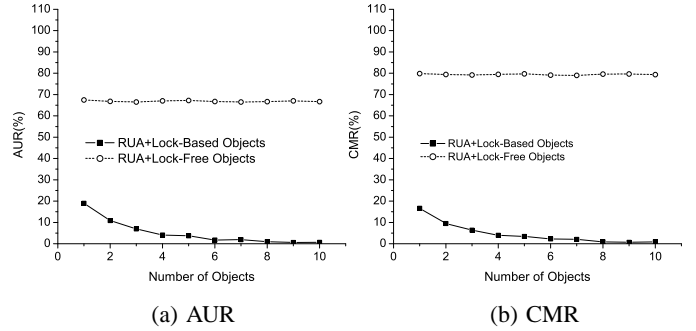


Figure 9: AUR/CMR During Overload, Step TUFs

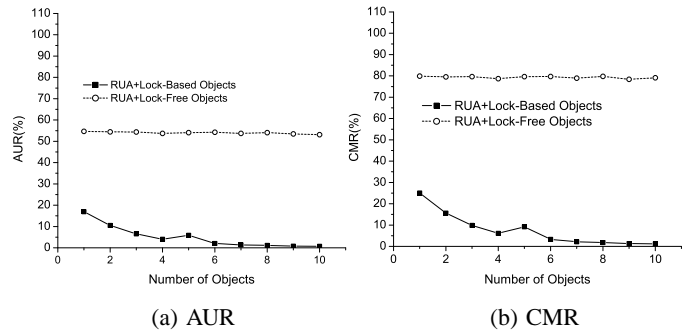


Figure 10: AUR/CMR During Overload, Heterogeneous TUFs

The performance of lock-free RUA, on the contrary, does not degrade as the number of objects increases. During under-loads, lock-free RUA achieves almost 100% AUR and CMR, whereas during overloads, the algorithm achieves higher AUR by as much as $\approx 65\%$ and higher CMR, by as much as $\approx 80\%$ than lock-based. This better performance is directly due to the short s of lock-free objects, which results in few retries and thus reduced interferences.

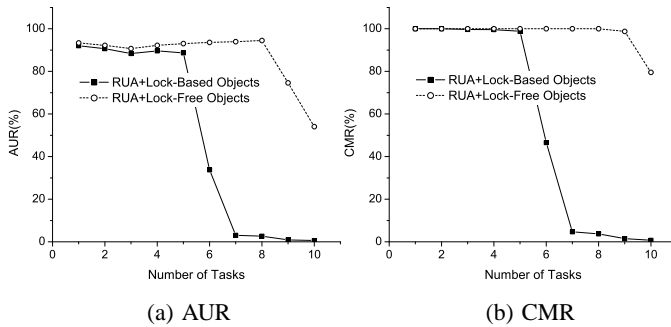


Figure 11: AUR/CMR During Increasing Readers, Heterogeneous TUFs

We repeated similar experiments as in Figures 7–10 for increasing number of reader tasks (instead of increasing shared objects) and observed exact similar trends and consistent results. Figure 11 shows a snapshot of these results (Heterogeneous TUFs, $AL=0.1-1.1$), further illustrating lock-free RUA’s superior performance over lock-based. Lock-based RUA starts to lose AUR/CMR earlier than Lock-free because its heavy overhead makes the earlier overload. We omit more results as they show the same trend and consistency. We emphasize again that the lock-free RUA’s superiority over lock-based RUA is mainly because s is much smaller than r , which is highly likely to happen in most systems, and lock-free approaches are optimistic.

6. CONCLUSION

In this paper, we consider non-blocking synchronization for embedded real-time systems that are subject to resource overloads and arbitrary activity arrivals. We consider lock-free synchronization for the multi-writer/multi-reader problem that occurs in such systems. We establish the tradeoffs between lock-free and lock-based object sharing under the UAM, including the conditions under which activity timeliness utility is greater under lock-free than under lock-based, and the upper bound on this utility increase — the first such result. Our implementation experience on a POSIX RTOS strongly validates our analytical results: Lock-free sharing yields higher accrued utility, by as much as 65%, and critical time satisfactions, by as much as 80%, over lock-based.

Several aspects of the work are directions for further research. Examples include extending the results to scheduling algorithms that provide individual activity timeliness assurances, higher-level abstractions such as snapshots, and other architectures such as multiprocessor systems and distributed systems.

7. REFERENCES

- [1] J. H. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *IEEE RTSS*, pages 111 – 122, Dec. 1997.
- [2] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM TOCS*, 15(2):134–165, 1997.
- [3] J. Chen. A loop-free asynchronous data sharing mechanism in multiprocessor real-time systems based on timing properties. In *ICDCSW 2003*, 2003.
- [4] J. Chen and A. Burns. A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-288, CS Dept., University of York, May 1997.
- [5] H. Cho, B. Ravindran, and E. D. Jensen. A space-optimal, wait-free real-time synchronization protocol. In *IEEE ECRTS*, 2005 (to appear).
- [6] H. Cho, B. Ravindran, and E. D. Jensen. On utility accrual processor scheduling with wait-free synchronization for embedded real-time software, 2005 (under review).
- [7] R. Clark, E. D. Jensen, et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, April 1999.
- [8] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990.
- [9] R. K. Clark, E. D. Jensen, and N. F. Rouquette. Software organization to facilitate dynamic processor scheduling. In *IEEE WPDRTS*, April 2004.
- [10] P. T. Hakan Sundell. Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information. In *RTCSA 2000*, pages 433–440, 2000.
- [11] J.-F. Hermant and G. L. Lann. A protocol and correctness proofs for real-time high-performance broadcast networks. In *IEEE ICDCS*, pages 360–369, 1998.
- [12] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.
- [13] H. Huang, P. Pillai, and K. G. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *USENIX Annual Technical Conference*, pages 303–316, 2002.
- [14] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *IEEE RTSS*, pages 112–122, December 1985.
- [15] H. Kopetz and J. Reisinger. The non-blocking write protocol nbw: A solution to a real-time synchronisation problem. In *IEEE RTSS*, pages 131–137, 1993.
- [16] P. Li, B. Ravindran, et al. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Trans. Software Engineering*, 30(9):613 – 629, Sept. 2004.
- [17] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986.
- [18] M. M. Michael and M. L. Scott. Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, May 1998.
- [19] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.
- [20] P. Sorensen and V. Hemacher. A real-time system design methodology. In *INFOR 13*, pages 1–18, 1975.
- [21] H. Sundell and P. Tsigas. Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information. In *IEEE RTCSA*, pages 433 – 440, 2000.
- [22] P. M. B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *IEEE Annual FOCS*, pages 233–243, 1986.
- [23] H. Wu, B. Ravindran, et al. Utility accrual scheduling under arbitrary time/utility functions and multiunit resource constraints. In *IEEE RTCSA*, August 2004.