Lock-Free Synchronization for Dynamic Embedded Real-Time Systems*

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 lock-free synchronization for dynamic embedded real-time systems that are subject to resource overloads and arbitrary activity arrivals. We model activity arrival behaviors using the unimodal arbitrary arrival model (or UAM). UAM embodies a stronger "adversary" than most traditional arrival models. We derive the upper bound on lock-free retries under the UAM with utility accrual scheduling — the first such result. We establish the tradeoffs between lock-free and lock-based sharing under UAM. These include 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 that is possible with lock-free. We confirm our analytical results with a POSIX RTOS implementation.

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 [7]) and control systems in the defense domain (e.g., airborne trackers [6]) 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. Another important distinguishing feature of these systems is their relatively long execution time magnitudes—e.g., in the order of milliseconds to minutes.

When resource overloads occur, meeting deadlines of all 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 urgency and importance, during overloads. (During under-loads, such a distinction need not be made, because deadline-based scheduling algorithms such as EDF are optimal.)

Deadlines by themselves cannot express both urgency and importance. Thus, we consider the abstraction of time/utility functions (or TUFs) [10] that express the utility of completing an 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 on the X-axis, and importance is denoted (by utility) on the Y-axis.

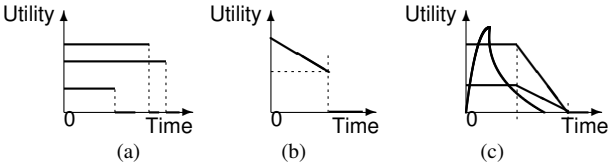


Figure 1. Example TUF Time Constraints

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 infinitively negative utility is accrued. Figures 1 show example such time constraints from two real applications [6]. When activity time constraints are specified using TUFs, the scheduling criteria are based on accrued utility, such as maximizing the total activity attained utility. We call such criteria, *utility accrual* (or UA) criteria, and scheduling algorithms that optimize them, as UA scheduling algorithms.

UA algorithms that maximize total utility under step TUFs (see algorithms in [15]) default to EDF during underloads, since EDF satisfies all deadlines during under-loads. Consequently, they obtain the maximum total utility during under-loads. During overloads, they favor more important activities (since more utility can be attained from them), irrespective of urgency. Thus, deadline scheduling's optimal timeliness behavior is a special-case of UA scheduling.

* This work was supported by the Office of Naval Research under Grant N00014-05-1-0179 and The MITRE Corporation under Grant 52917.

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., [16], see algorithms in [15]; and (2) non-blocking schemes including wait-free protocols (e.g., [11, 4, 9]) and lock-free protocols (e.g., [3]).

Lock-based protocols have several disadvantages such as serialized access to shared objects, resulting in reduced concurrency [3]. Further, many lock-based protocols typically incur additional run-time overhead due to scheduler activations that occur when activities request locked objects [16, 15]. Also, deadlocks 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 [16], which may be difficult to obtain for dynamic applications, resulting in reduced flexibility [3]. These drawbacks have motivated research on non-blocking objects in embedded real-time systems.

Especially, lock-free objects guarantee that some object operations will complete in a finite number of steps. It implies other operation may have to retry a potentially infinite number of time. Instead of acquiring locks, lock-free operation continuously accesses the object, checks, and retries until it becomes successful. Inevitably, lock-free protocols incur additional time costs due to their retries, which is antagonistic to timeliness optimization. Prior research has shown how to mitigate these time or space costs. On the other hand, wait-free objects guarantee any operation on the objects will complete in a bounded number of steps, regardless of interferences. Wait-free protocols may incur additional time or space costs for their intrinsic mechanisms.

An excellent survey of the prior research can be found in [9]. We summarize some important efforts in the context of our work here: In [3], Anderson *et al.* show how to bound the retry loops of lock-free protocols through judicious scheduling. Lock-free objects on single- and multi-processors under quantum-based scheduling is presented in [1]. This work assumes that each task can be preempted at most once during a single quantum; thus each object access needs to be retried at most once. In [2], Anderson *et al.* present wait-free schemes for single- and multi-processors, where a task which announces its intention to share objects also helps other tasks to complete their access.

Efficient lock-free objects, such as queues and stacks, have been presented in [14, 13]. In [18], Valois presents lock-free linked lists. Treiber presents lock-free stack algorithms in [17]. In [11], Kopetz *et al.* present an analysis on non-blocking synchronization in real-time systems. This work was later improved by Chen *et al.* in [4], and subsequently by Huang *et al.* in [9] and by Cho *et al.* in [5].

1.2. Contributions

In this paper, we focus on *dynamic* embedded real-time systems on a single processor. By dynamic systems, we mean those 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 (UAM) [8]. UAM specifies the maximum number of activity arrivals that can occur during any time interval. Consequently, the model subsumes most traditional arrival models (e.g., periodic, sporadic) as special cases.

We consider lock-free sharing under the UAM. The past work on lock-free sharing upper bounds the retries under restrictive arrival models like periodic [3] — retry bounds under the UAM are not known. Moreover, we consider the UA criteria of maximizing the total 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 (RUA) scheduling algorithm [19], 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 derive the upper bound on lock-free RUA's retries under the UAM — the first ever retry bound under a non-periodic arrival model. Since lock-free sharing incurs additional time overhead due to the retries (as compared to lock-based), we establish the conditions under which activity sojourn times are shorter under lock-free RUA than under lock-based, for the UAM. From this result, we establish the maximum increase in activity utility under lock-free RUA over lock-based. Further, we implement lock-free and lock-based RUA on a POSIX RTOS. Our implementation measurements strongly validate our analytical results.

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. Section 4 discusses our implementation experience. We conclude the paper in Section 5. In describing our work, we skip some proofs for brevity; they can be found in [20].

2. Bounding Retries Under UAM

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 [8]. Jobs may arrive simultaneously. The periodic model is a special case of UAM with $\langle 1, 1, W_i \rangle$.

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

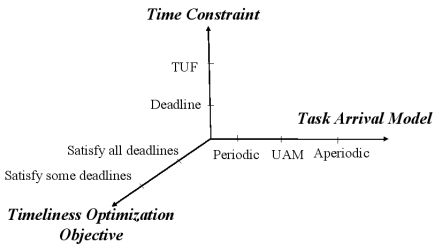


Figure 2. Three Dimensions of Task Model

a task have the same TUF. $U_i(\cdot)$ denotes task T_i's TUF; thus, completion of T_i at time t will yield $U_i(t)$ utility.

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 under-load situations, during which all tasks can be completed before their critical times, and overloads, where only a subset of the tasks (including possibly none) can be done so. Our model includes overloads, the UAM, and TUFs.

Finally, the resource model we consider here does not allow the nested critical section which may cause deadlock in the lock-based synchronization.

2.1. Overview of Lock-Based RUA

RUA [19] targets dynamic applications, where activities are subject to arbitrary arrivals and resource overloads. Further, activities may access (logical and physical) resources arbitrarily—e.g., the resources that will be needed, the length of time for which they will be needed, and the order of accessing them are all statically unknown. Thus, the set of activities to be scheduled and their resource dependencies may change over time. Consequently, RUA performs scheduling entirely online. RUA considers activities subject to arbitrarily shaped TUF time constraints, concurrent object sharing under mutual exclusion constraints, and the scheduling objective of maximizing the total utility.

With n jobs, RUA's asymptotic cost is $O(n^2 logn)$. This is higher than that of many traditional real-time scheduling algorithms. However, this high cost is justified for application with longer execution time magnitudes such as those that we focus here. (Of course, this high cost cannot be justified for every application.) Nevertheless, it is desirable to reduce the cost so that the resources utilized by the scheduling algorithm can yield greater benefit in terms of improved scheduling from the application's point of view.

2.2. Preemptions Under UA Schedulers

Under fixed priority schedulers such as rate-monotonic (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.

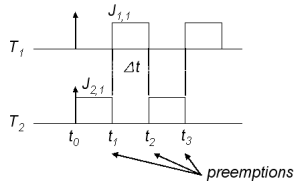


Figure 3. Mutual Preemption Under RUA

In Figure 3, assume that two jobs $J_{1,1}$ and $J_{2,1}$ arrive at time t_0, and scheduling events occur at t_1, t_2, and t_3. $J_{2,1}$ occupies CPU at 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. 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, is when TUFs of the two jobs are increasing.

This potential mutual preemption distinguishes RUA from traditional 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 (see [2]). However, this is not true with RUA, as one job can be preempted by another job more than once. Thus, the maximum number of preemptions that a job may experience under RUA is bounded by the number of events that invoke the scheduler.

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.*

Lemma 1 helps compute the upper bound on the number of retries on lock-free objects for our model. This is also true with other UA schedulers [15], because it is impossible for more preemptions to occur than scheduling events.

2.3. Bounded Retry Under UAM

Under our model, jobs with TUF constraints arrive under the UAM, and there may not always be enough CPU time 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)**.** *Let 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,* J_i*'s total number of retries,* f_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:

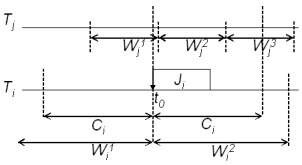


Figure 4. Interferences Under UAM

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. □

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 those in [2]. 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 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 [19], 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 objects that can be used to block J_i. Thus, 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 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.

Theorem 3 (Lock-Based versus Lock-Free Sojourn)**.** *Let jobs arrive under the UAM and be scheduled by RUA. If*

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

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

Theorem 3 shows that at least $\frac{s}{r} < \frac{2}{3}$ for jobs to obtain a shorter sojourn time under lock-free. In [2], Anderson *et al.* show that s is often much smaller than r in comparison with the vast majority of lock-free objects in most systems, such as buffers, queues, and stacks.

Corollary 4 (Special Case Sojourn)**.** *Let jobs arrive under the UAM and be scheduled by RUA. Let r be much larger than s enough for $\frac{s}{r}$ to converge to zero. Now, if:*

$$((0 < m_i < 2a_i + x) \wedge (m_i \leq n)) \vee (m_i > n),$$

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

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 time for other jobs.

When 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:*

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

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.

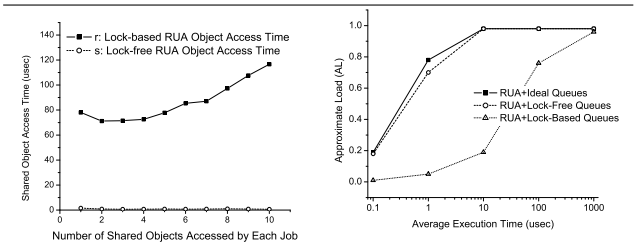
4. Implementation Experience

We implemented lock-based and lock-free objects with RUA in the *meta-scheduler* scheduling framework [12], which allows middleware-level real-time scheduling atop POSIX RTOSes. Our motivation for implementation is to verify our analytical results. We used QNX Neutrino 6.3 RTOS running on a 500MHz, Pentium-III processor in our implementation, which provides an atomic memory-modification operation, the CAS (Compare-And-Swap) instruction. In our study, we used queues, one of the common shared objects, to validate our theorems. We used the lock-free queues introduced in [14] in our implementation.

4.1. Object Access Times, Critical Load

As Theorem 3 shows, the tradeoff between lock-based and lock-free sharing under RUA depends upon 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 with a 10 task set. Each measurement is an average of approximately 2000 samples.

Figure 5(a) shows r and s under increasing number of shared objects accessed by jobs, not allowing any nested critical section. From the figure, we observe that r is significantly larger than s. As the number of shared objects accessed by each job, r is increasing. Note that r includes the time for lock-based RUA's resource sharing mechanism.



(a) Object Access Time (b) Critical Time Miss Load

Figure 5. Object Access Times and Critical Time Miss Load

When $r >> s$, Theorem 3 implies that lock-free is likely to perform better than lock-based. Furthermore, when the number of objects m_i increases, it increases the likelihood of satisfying Corollary 4 and lock-free becomes increasingly advantageous over lock-based.

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 it using a metric called *Critical time-Miss Load* (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 object access time, and C_i is the task critical time.

We exclude object access time in CML, 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. We call it, the ideal RUA.

We consider a task set of 10 tasks, accessing 10 shared queues, and measure the CML of lock-free, lock-based, and ideal RUA under increasing average job execution times. Figure 5(b) 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 CML of 1 is valid at zero job execution times when assuming no system overheads, which is not true in practice.)

On the other hand, lock-based RUA's CML converges to 1, only at $\approx$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(a).

4.2. Accrued Utility, Critical Time-Meets

We now measure the AUR and the 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 a heterogenous class including step, parabolic, and linearly-decreasing shapes.

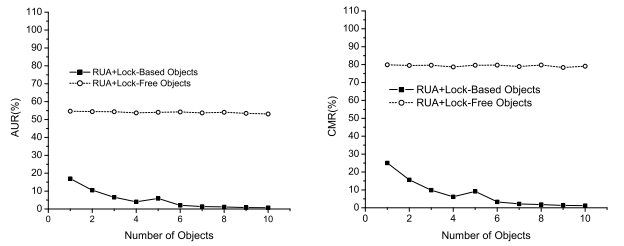


Figure 6. AUR/CMR During Overload, Heterogeneous TUFs

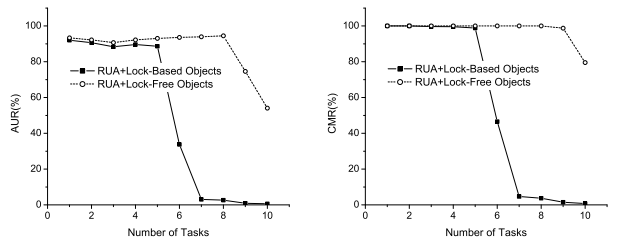


Figure 7. AUR/CMR During Increasing Readers, Heterogeneous TUFs

Figures 6 and 7 show lock-based and lock-free RUA's performance, respectively, under heterogenous TUFs, AL = $\approx$1.1, and increasing number of shared objects. As expected, the figures show that lock-based RUA's AUR and CMR 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. The performance of lock-free RUA, on the contrary, does not degrade as the number of objects increases. This is directly due to the short s of lock-free objects, which results in few retries and thus reduced interferences.

We repeated similar experiments for increasing number of reader tasks (instead of increasing shared objects) and observed exact similar trends and consistent results in Figure 7 (Heterogeneous TUFs, AL=0.1-1.1), further illustrating lock-free RUA's superiority over lock-based. We omit more results as they show the same trend and consistency.

5. Conclusions

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.

Future work includes extending the results to algorithms that provide activity-level timing assurances, the snapshot abstraction, and multiprocessor and distributed systems.

References

[1] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *RTSS*, 1998.

[2] 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.

[3] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM TOCS*, 15(2):134–165, 1997.

[4] J. Chen and A. Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *10th Euromicro Workshop on Real-Time System*, 1998.

[5] H. Cho et al. A space-optimal, wait-free real-time synchronization protocol. In *IEEE ECRTS*, 2005.

[6] R. Clark, E. D. Jensen, et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, April 1999.

[7] R. K. Clark, E. D. Jensen, and N. F. Rouquette. Software organization to facilitate dynamic processor scheduling. In *IEEE WPDRTS*, April 2004.

[8] 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.

[9] H. Huang, P. Pillai, and K. G. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *USENIX Annual*, pages 303–316, 2002.

[10] 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.

[11] 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.

[12] P. Li, B. Ravindran, et al. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE TSE*, 30(9):613 – 629, Sept. 2004.

[13] M.Herlihy. A methodology for implementing highly concurrent data objects. *ACM TOPLAS*, 15(5):745–770, 1993.

[14] M. M. Michael and M. L. Scott. Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *JPDC*, 51(1):1–26, May 1998.

[15] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *IEEE ISORC*, pages 55 – 60, May 2005.

[16] 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.

[17] R. K. Treiber. System programming: Copying with parallelism. Technical report, IBM Almaden Research Center, April 1986. RJ 5118.

[18] J. Valois. Lock-free linked list using compare-and-swap. In *ACM PODC*, pages 214–222, 1995.

[19] H. Wu, B. Ravindran, et al. Utility accrual scheduling under arbitrary time/utility functions and multiunit resource constraints. In *IEEE RTCSA*, August 2004.

[20] H. Cho. Utility Accrual Scheduling with Non-Blocking Synchronization on Uniprocessors and Multiprocessors. In *PhD Dissertation Proposal*, ECE Dept., Virginia Tech, 2005, http://www.ee.vt.edu/~realtime/cho_proposal05.pdf.