On Utility Accrual Processor Scheduling with Wait-Free Synchronization for Embedded Real-Time Software

ABSTRACT

We present the first *wait-free* utility accrual (UA) real-time scheduling algorithms for embedded real-time systems. UA scheduling algorithms allow application activities to be subject to time/utility function (TUF) time constraints, and optimize criteria such as maximizing the sum of the activities' attained utilities. We present UA algorithms that use wait-free synchronization for mutually exclusively accessing shared data objects. We derive upper bounds on the maximum possible increase in activity utility with wait-free over their lock-based counterparts, while incurring the minimum possible additional space costs, and thereby establish the tradeoffs between wait-free and lock-based object sharing for UA scheduling. Our implementation measurements on a POSIX RTOS reveal that (during under-loads), the wait-free algorithms yield optimal utility for step TUFs and significantly higher utility (than lock-based) for non-step TUFs.

1. INTRODUCTION

Emerging real-time embedded systems such as robotic systems in the space domain (e.g., NASA's Mars Rover [6]) and control systems in the defense domain (e.g., airborne tracking systems [4]) are subject to time constraints that are "soft" in the sense that completing an activity at any time will result in some (positive or negative) utility to the system, and that utility depends on the activity's completion time. With such soft time constraints, the optimality criteria is often to complete all time-constrained activities as close as possible to their *optimal* completion times—so as to yield maximal collective utility.

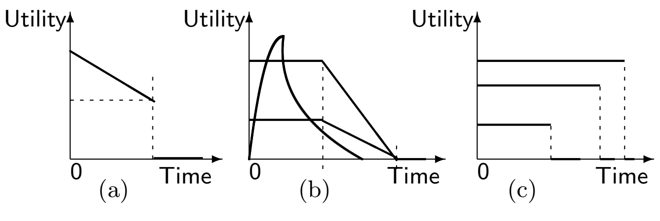


Figure 1: Example TUF Time Constraints

Jensen's time/utility functions (or TUFs) [5] allow the semantics of soft time constraints to be precisely specified. A TUF, which generalizes the deadline constraint, specifies the utility to the system resulting from the completion of an activity as a function of its completion time. Figures 1(a)–1(b) show time constraints specified using TUFs of two real applications in the defense domain (see [4] and the references therein). The classical deadline is a binary-valued, downward "step" shaped TUF; Figure 1(c) shows examples.

When activities are subject to TUF time constraints, the scheduling criteria are based on accrued utility—e.g., maximizing the summed attained utility. Such criteria are called *utility accrual* (or UA) criteria, and scheduling algorithms that optimize them are called UA scheduling algorithms.

Note that UA algorithms that maximize summed utility under downward step TUFs (or deadlines), meet all activity deadlines during under-loads [5, 11]. Thus, UA scheduling 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, and thus the system's behavior. Mechanisms that resolve such contention can be broadly classified into: (1) lock-based schemes—e.g., [10, 5]; and (2) non-blocking schemes including wait-free protocols (e.g., [8, 2, 7]) and lock-free protocols (e.g., [1]).

Lock-based protocols have several disadvantages such as serialized access to shared objects, resulting in reduced concurrency and thus reduced resource utilization [1]. Further, many lock-based protocols typically incur additional run-time overhead due to protocol (or scheduler) activations that occur when activities request previously locked shared objects, which are scheduling events [10, 5]. Another disadvantage is the possibility of deadlocks that can occur when lock holders crash, causing indefinite starvation to blockers. Further, many (real-time) lock-based protocols require a-priori knowledge of the ceilings of locks [10], which may be sometimes difficult to obtain. Furthermore, OS data structures (e.g., semaphore control blocks) must be a-priori updated with that knowledge, resulting in reduced flexibility [1].

These drawbacks have motivated research on wait-free object sharing in real-time systems. Wait-free protocols use multiple buffers (e.g., a circular buffer). For the single-writer/multiple-reader problem, wait-free protocols typically use buffers that is proportional to the maximum number of times the readers can be preempted by the writer, during when the readers are reading. The maximum number of preemptions of a reader by the writer 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 by the writer, the readers and the writer can continuously read and write in different buffers, respectively, and thereby avoid interference.

However, wait-free protocols incur additional space costs due to their usage of multiple buffers, which is infeasible for many small-memory embedded real-time systems. Prior research have shown how to mitigate such costs. In [8], Kopetz *et al.* present one of the earliest wait-free protocols. Chen *et al.* build upon [8] and present an efficient wait-free protocol in [2]. Huang *et al.* improve the time costs of Chen's protocol in [7]. Chen's protocol is further improved by Cho *et al.* to develop the space-optimal wait-free protocol for the single-writer/multiple-reader problem in [3].

1.2 Synchronization Under UA Scheduling

In this paper, we consider wait-free synchronization for the single-writer/multiple-reader problem in embedded real-time systems that are subject to TUF time constraints and UA optimality criteria. Our motivation to consider wait-free synchronization for UA scheduling is to reap its advantages (e.g., reduced object access time, greater concurrency, reduced run-time overhead, fault-tolerance) toward better optimization of UA criteria. In particular, we hypothesize that the reduced shared object access time under wait-free synchronization will result in increased activity attained utility. Of course, this will come with the additional buffer cost.

Thus, our goal is to develop wait-free UA scheduling algorithms that use the absolute minimum buffer size, and verify whether such algorithms can yield greater activity utility than lock-based ones. This will allow us to establish the fundamental tradeoffs between wait-free and lock-based object sharing for UA scheduling. We precisely do so in this paper.

We focus on wait-free synchronization, as opposed to lock-free, as lock-free incurs additional time costs (due to their retry loops), which can potentially reduce attained utility. We consider the single-writer/multiple-reader problem, as it occurs in most embedded real-time systems [7]. Further, we consider the UA optimality criteria of maximizing the sum of the activities' attained utilities, while yielding optimal total utility for step TUFs during under-loads, and ensuring the integrity of shared data objects under concurrent access.

UA scheduling under wait-free synchronization has never been studied in the past. Thus, we consider the two lock-based UA algorithms that match our exact UA criteria: (1) the Resource-constrained Utility Accrual (or RUA) scheduling algorithm [11] and (2) the Dependent Activity Scheduling Algorithm (or DASA) [5]. We develop wait-free versions of RUA and DASA using the space-optimal protocol in [3].

We analytically compare RUA's and DASA's wait-free and lock-based versions. We establish upper bounds on the maximum increase in utility that is possible with wait-free over lock-based. The upper bounds — the first such — verify our hypothesis. Our measurements from a POSIX RTOS implementation reveal that during under-loads, wait-free algorithms yield optimal utility for step TUFs and significantly higher utility (than lock-based) for non-step TUFs.

Thus, the paper's central contribution is wait-free UA scheduling algorithms, the upper bounds on their maximum possible increase in activity utility over lock-based, for the minimum additional space cost, and the resulting tradeoff. We are not aware of any other wait-free UA algorithm.

The rest of the paper is organized as follows: In Section 2, we overview the wait-free protocol in [3] for completeness. We describe RUA, develop its wait-free version, and derive the upper bound on the increase in utility in Section 3. Similarly, we present results for DASA in Section 4. In Section 5, we discuss our implementation experience and report our measurements. Finally, we conclude the paper in Section 6.

2. AN OPTIMAL WAIT-FREE PROTOCOL

The wait-free synchronization requires the protocol to hold two properties: safety and orderliness. The safety property ensures that the shared object does not become corrupted during reading and writing—i.e., ensures mutual exclusion. The orderliness property ensures that all readers always read the latest data that is completely written by the writer. In [3], Cho *et al.* present a wait-free protocol that achieves these two properties, and uses the minimum possible number of buffers. We summarize this protocol.

2.1 Number of Buffers in Use

The single-writer/multiple-reader problem is considered under the periodic task model, for under-load situations. For convenience, the total number of readers are denoted by M and the i^{th} reader by R_i. The reader R_i's j^{th} instance of reading is denoted by $R_i^{[j]}$. The writer's k^{th} writing instance is denoted by $W^{[k]}$. $R_i^{[j]}(op)$ stands for a specific operation op of $R_i^{[j]}$—e.g., $R_i^{[j]}(READING[i] = 0)$ implies the execution of one operatoin in Chen's algorithm [2]. $W^{[k]}(op)$ also stands for the operation in $W^{[k]}$. If $R_i^{[j]}$ reads what $W^{[k]}$ writes, we denote it as $w(R_i^{[j]}) = W^{[k]}$.

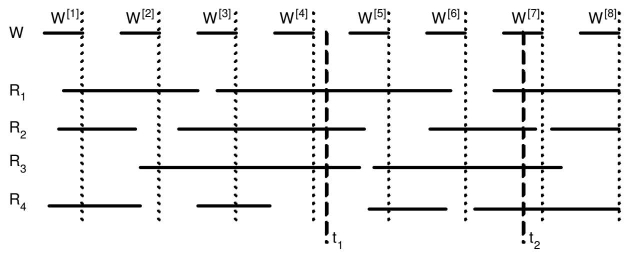


Figure 2: Number of Buffers in Use

Suppose we have 4 readers and 1 writer, as shown in Figure 2. At time t_1, $w(R_1)$=$W^{[2]}$, $w(R_2)$=$W^{[2]}$, and $w(R_3)$=$W^{[1]}$. This implies that two buffers are being accessed by the readers. Additionally, to achieve the safety and orderliness properties, one buffer is required to store and save the latest completely written data by $W^{[4]}$, and another is needed for the next writing operation by $W^{[5]}$. Thus, four buffers are being used in total, at time t_1.

The basic intuition for determining the minimum number of buffers is to construct a worst-case where the required number of buffers are as large as possible, when the maximum possible number of interferences of all readers with the writer occurs. This problem can be mapped into the *Diverse Selection Problem* (or DSP).

2.2 Diverse Selection Problem

The DSP, $D(R, \vec{R}(\vec{x}))$, is defined with the problem range R and the range vector $\vec{R}(\vec{x})$ of all elements in the vector $\vec{x}$. Each element x_i in the vector $\vec{x}$ has the range $r_i = [l_i, u_i]$. The solution to the problem D is represented as a vector $\vec{x} =< x_1, ..., x_M >$, where the vector size $n(\vec{x})$ is M. Every x_i must satisfy its range constraint r_i and the problem range constraint R simultaneously. $\{\vec{x}\}$ is defined as a set including all elements of $\vec{x}$, but without duplicates. Thus, the size of $\{\vec{x}\}, n(\{\vec{x}\})$, is less than or equal to $n(\vec{x})$. The objective of DSP is to determine the maximum $n(\{\vec{x}\})$ by selecting $\vec{x}$, satisfying all range constraints as diversely as possible. In [3], Cho *et al.* present an efficient approach to solve DSP by an inductive strategy.

Given a vector, $\vec{v} =< v_1, ..., v_i, ... >$, the number of v_i's having k value is denoted by $H(\vec{v}, k)$, and the maximum value among all v_i's as $Top(\vec{v})$. Given $D(R, \vec{R_{\vec{x}}})$, the optimal solution of D when $R = [t_1, t_2]$ is denoted as $n^{max}_{[t_1,t_2]}(\{\vec{x}\})$. Here, the lower bounds of all ranges are assumed to be 1 to make it easily applicable to the wait-free problem.

Theorem 1 (DSP for the Wait-Free Protocol). *In the DSP $D(R,\vec{u})$ with $R = [1, N], n^{max}_{[t+1]}(\{\vec{x}\}) =$*

$$\begin{cases} n^{max}_{[t]}(\{\vec{x}\}) + 1, & \text{if } \sum_{k=0}^{t+1} H(\vec{r}, N-k) > n^{max}_{[t]}(\{\vec{x}\}) \\ n^{max}_{[t]}(\{\vec{x}\}), & \text{otherwise} \end{cases}$$

where $N = Top(\vec{u})$, $[t]$=$[N-t, N]$, and $0 \le t < N$. When $t = 0$, $n^{max}_{[0]}(\{\vec{x}\}) = 1$.

Proof. See [3]. □

As shown in Theorem 1, the solution to $D([1,N],\vec{r})$ can be decomposed and build up from $D([i,N],\vec{r})$, where $1 \le i \le N$, inductively.

2.3 DSP and WFBP

The DSP has similarity with the *Wait-Free Buffer size decision Problem* (or WFBP). Here, there are M readers and their maximum interferences are $< N^{max}_1, ..., N^{max}_M >$. WFBP's objective is to determine the worst-case maximum number of buffers.

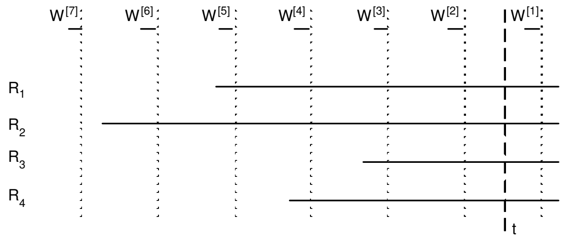


Figure 3: A Worst-Case of the WFBP

Figure 3 illustrates with an example, how to construct the worst-case where the required number of buffers are as large as possible. In this example, R_1's maximum interference is 5, which is illustrated in a line. It means that $w(R_1)$ may belong to the set $\{W^{[1]},...,W^{[6]}\}$. The worst-case is assumed to happen at time t between $W^{[2]}$ and $W^{[1]}$, where $W^{[2]}$ writes the latest completely written data, and $W^{[1]}$ is the next writing operation for which another buffer is needed. For this reason, WFBP is restated as determining $\vec{x} =< w(R_1), ..., w(R_M) >$ that will maximize $n(\{\vec{x}\} \cup \{W^{[1]}, W^{[2]}\})$, where $w(R_i) \in \{W^{[1]}, ..., W^{[N^{max}_i+1]}\}$. If $W^{[j]}$ is abbreviated as j, the problem is redefined as determining $\vec{x} =< x_1, ..., x_M >$ that will maximize $n(\{\vec{x}\} \cup \{1, 2\})$, where $x_i \in \{1, ..., N^{max}_i + 1\}$. Therefore, the final solution $\{\vec{x}\}$ of a given WFBP is obtained with a sum of the solution from a mapped DSP and a set $\{1, 2\}$.

Corollary 2 (The Optimality). *If a solution to the WFBP can be obtained, then it must be the minimum and space-optimal buffer size that satisfies the two properties, safety and orderliness.*

Proof. See [3]. □

Algorithm 1 solves WFBP based on Theorem 1. The *sum* and the function *doesExist*(t) correspond to $\sum H(\vec{u}, ...)$ and $H(\vec{u}, t)$ in Theorem 1. Note that Algorithm 1 is independent of the scheduling algorithm. Instead, it depends on N^{max} analysis.

Algorithm 1: Algorithm for WFBP

```
input : number of readers M; maximum
        interference $N^{max}$[M]
output: required buffer size $n$
$sum$=$n$=0;
$on_1$=$on_2$=false;
for $i = 0$ to M-1 do $N^{max}[i]$++;
sort( $N^{max}$[1,...,M] );
for $t$=$N^{max}$[0] to 1 do
    $sum$ += doesExist( $t$, $N^{max}$[1,...,M] );
    if $sum>n$ then
        $n$++;
        if $t$=2 then $on_2$ = true;
        if $t$=1 then $on_1$ = true;
if $on_2$=false then $n$++;
if $on_1$=false then $n$++;
```

3. SYNCHRONIZATION IN RUA

RUA [11] considers activities subject to arbitrarily shaped TUF time constraints and concurrent sharing of non-CPU resources including logical (e.g., data objects) and physical (e.g., disks) resources. The algorithm allows concurrent resource sharing under mutual exclusion constraints.

RUA's objective is to maximize the total utility accrued by all activities. To develop RUA's wait-free version, we first overview the original lock-based algorithm.

3.1 Lock-Based RUA

RUA's scheduling events include task arrivals, task departures, and lock, and unlock requests. When RUA is invoked, it first builds each task'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 task in the cycle, which will likely contribute the least utility.

After handling deadlocks, RUA examines tasks in the order of non-increasing potential utility densities (or PUDs). The PUD of a task is the ratio of the expected task utility to the remaining task execution time, and thus measures the task's return of investment. The algorithm inserts a task and its dependents into a tentative schedule, in the order of their critical times, earliest critical time first. Critical time is the time at which the task TUF has zero utility–i.e., the "deadline" time (all TUFs are assumed to have such a time). The insertion is done by respecting the tasks' dependencies.

After insertion, RUA checks the schedule's feasibility. If infeasible, the inserted task and its dependents are rejected. The algorithm repeats the process until all tasks are examined, and selects the task at the schedule head for execution.

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

In [11], Wu *et al.* bound the maximum blocking time that a task may experience under RUA for the special-case of the single-unit resource model:

Theorem 3 (RUA's Blocking Time). *Under RUA with the single-unit resource model, a task T_i can be blocked for at most the duration of $min(n, m)$ critical sections, where n is the number of tasks that can block T_i and have longer critical times than T_i, and m is the number of resources that can be used to block T_i.*

PROOF. See [11]. □

3.2 Wait-Free RUA

We focus on the single-unit resource model for developing RUA's wait-free version. With wait-free synchronization, RUA is significantly simplified: Scheduling events now include only task arrivals and task departures — lock and unlock requests are not needed anymore. Further, no dependency list arises and need to be built. Moreover, deadlocks will not occur due to the absence of object/resource dependencies. All these reduce the algorithm's complexity. RUA's wait-free version is described in Algorithm 2.

Algorithm 2: Wait-Free RUA

```
1  input  : $\mathcal{T}_r$, task set in the ready queue
2  output: selected thread $T_{exe}$
3  Initialization: $t$=$t_{cur}$, $\sigma = \phi$;
4  for $\forall T_i \in \mathcal{T}_r$ do
5      if feasible($T_i$) then
6          $T_i.PUD = \frac{U_i(t+T_i.ExecTime)}{T_i.ExecTime}$;
7      else
8          abort($T_i$);
9  $\sigma_{tmp1}$=sortByPUD( $\mathcal{T}_r$ );
10 for $\forall T_i \in \sigma_{tmp1}$ from head to tail do
11     if $T_i.PUD > 0$ then
12         $\sigma_{tmp2} = \sigma$;
13         InsertByDeadline($T_i$,$\sigma_{tmp2}$);
14         if feasible($\sigma_{tmp2}$) then
15             $\sigma = \sigma_{tmp2}$;
16     else
17         break;
18 $T_{exe}$=headOf($\sigma$);
19 return $T_{exe}$;
```

Lock-based RUA's time complexity grows as a function of the number of tasks, while that for wait-free RUA, it grows as a function of the number of readers and number of shared objects. (The cost of lock-based RUA is independent of the number of objects, as the dependency list built by RUA may contain all tasks in the worst-case, irrespective of the number of objects.) The number of readers can exceed the number of tasks under nested critical sections—e.g., a single task can make multiple (nested) object requests, resulting in as many readers as there are requests.

With n tasks and m readers, the time complexity of lock-based RUA is $O(n^2 \log n)$ [11], while that of wait-free RUA is $O(n^2+m)$, as Algorithm 2 shows. Especially if there is no nested object request, the number of readers m, is bounded by n. When that happens, wait-free RUA improves upon lock-based RUA from $O(n^2 \log n)$ to $O(n^2)$.

We now formally compare task sojourn times under wait-free and lock-based versions of RUA. We assume that all accesses to lock-based shared objects require r units of time, and that to wait-free shared objects require w units. The computation time c_i of a task T_i can be written as $c_i = u_i + m_i \times t_{acc}$, where u_i is the computation time excluding accesses to shared objects; m_i is the number of shared object accesses by T_i; and t_{acc} is the maximum computation time for any object access—i.e., r for lock-based objects and w for wait-free objects.

THEOREM 4 (COMPARISON OF RUA'S SOJOURN TIMES). *Under RUA, as the critical section t_{acc} of a task T_i becomes longer, the difference between the sojourn time with lock-based synchronization, s_{lb}, and that with wait-free protocol, s_{wf}, converges to the range:*

$$0 \leq s_{lb} - s_{wf} \leq r \cdot min(n_i, m_i),$$

where n_i is the number of tasks that can block T_i and have longer critical times than T_i, and m_i is the number of shared data objects that can be used to block T_i.

PROOF. The sojourn time of a task T_i under RUA with lock-based synchronization includes the execution time $c_i = u_i + m_i \times r$, the preemption time I_i, and the blocking time B_i. Based on Theorem 3, the blocking time B_i is at most $m_i \times min(n_i, m_i)$. On the other hand, the sojourn time of task T_i under RUA with wait-free protocol does not include any blocking time. Therefore, the difference between lock-based synchronization and wait-free is at most $m_i \times (r-w) + r \times min(m_i, n_i)$. Assuming that the data for synchronization is large enough such that the execution time difference in the algorithm between lock-based synchronization and wait-free becomes negligible (i.e., the time for reading and writing the data object becomes dominant in the time for synchronization), then r and w converge and become equal. In this case, the sojourn time of T_i under lock-based synchronization is longer than that under wait-free by $r \times min(n_i, m_i)$. □

The reduced sojourn time of a task under wait-free increases the accrued utility of the task, for non-increasing TUFs. Further, this potentially allows greater number of tasks to be scheduled and completed before their critical times, yielding more total accrued utility (for such TUFs).

Based on Theorem 4, we can now estimate the difference in the Accrual Utility Ratio (or AUR) between wait-free and lock-based, for non-increasing TUFs. AUR is the ratio of the actual accrued utility to the maximum possible utility.

COROLLARY 5. *Under RUA, with non-increasing TUFs, as the critical section t_{acc} of a task T_i becomes longer, the difference in AUR, $\Delta AUR = AUR_{wf} - AUR_{lb}$, between lock-based synchronization and wait-free converges to:*

$$0 \leq \Delta AUR \leq \sum_{i=1}^{N} \frac{U_i(s_{wf}) - U_i(s_{wf} + r \cdot min(m_i, n_i))}{U_i(0)},$$

where $U_i(t)$ denotes the utility accrued by task T_i when it completes at time t, and N is the number of tasks.

PROOF. It directly follows from Theorem 4. □

When the data for synchronization is small and the time for reading and writing the shared data object is not dominant, r and w are more dependent on the object sharing algorithm. The execution time of lock-based synchronization, r, includes the time for the locking and unlocking procedure (needed for mutual exclusion), executing the scheduler, and accessing the shared object, while w includes the time for controlling the wait-free protocol's variables and accessing the shared object. The wait-free protocol does not activate the scheduler, and hence avoids significant system overhead. Consequently, w's of many wait-free protocols are known to be shorter than r [3, 7]. Thus, no matter what the size of the data to communicate between tasks is, wait-free synchronization conclusively accrues more utility compared with lock-based synchronization.

4. SYNCHRONIZATION IN DASA

DASA [5] considers activities subject to step-shaped TUFs and concurrent sharing of non-CPU resources (e.g., data objects, disks) under mutual exclusion constraints and under the single-unit resource request model. Thus, DASA's model is a proper subset of RUA: DASA is restricted to step TUFs and the single-unit model, while RUA allows arbitrarily shaped TUFs and the multi-unit model (we focus on RUA's single-unit model here).

Like RUA, DASA's objective is to maximize the total utility accrued by all activities. DASA's basic operation is identical to that of RUA for the single-unit model (see Section 3.1). Thus, for the single-unit model, RUA's behavior subsumes DASA's. Therefore, in [11], Wu *et al.* also show that the maximum blocking time that a task may suffer under DASA is identical to that under RUA (for the single-unit model), which is stated in Theorem 3.

4.1 Wait-Free DASA

Since TUFs under DASA are step-shaped, shorter sojourn time does not increase accrued utility, as tasks accrue the same utility irrespective of when they complete before their critical times, as long as they do so. However, shorter sojourn time is still beneficial, as it reduces the likelihood of missing task critical times. Hence, lesser the number of tasks missing their critical times, higher will be the total accrued utility.

Wait-free synchronization prevents DASA from loosing utility no matter how many dependencies arise. Similar to wait-free RUA, wait-free DASA is also simplified: Scheduling events now include only arrivals and departures, no dependency list needs to be built, and no deadlocks will occur.

The time complexities of lock-based and wait-free DASA are identical to that of lock-based and wait-free RUA: $O(n^2)$ [5] and $O(n^2 + m)$, respectively. Similar to wait-free RUA, when m is bounded by n, wait-free DASA improves upon lock-based DASA from $O(n^2 \log n)$ to $O(n^2)$.

The comparison of DASA's sojourn times under lock-based and that under wait-free is similar to Theorem 4:

THEOREM 6 (COMPARISON OF DASA'S SOJOURN TIMES). *Under DASA, as the critical section t_{acc} of a task T_i becomes longer, the difference between the sojourn time with lock-based synchronization, s_{lb}, and that with wait-free protocol, s_{wf}, converges to:*

$$0 \leq s_{lb} - s_{wf} \leq r \times min(n_i, m_i),$$

where n_i is the number of tasks that can block T_i and have longer critical times than T_i, and m_i is the number of shared data objects that can be used to block T_i.

PROOF. See proof of Theorem 4. □

5. IMPLEMENTATION EXPERIENCE

We implemented the lock-based and wait-free algorithms in the *meta-scheduler* scheduling framework [9], 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.

Our task model for computing the wait-free buffer size follows the model in [7]. We determine the maximum number of times the writer can interfere with the reader, N^{max} as:

$$N^{max} = \max(2, \left\lceil \frac{P_R - (C - C_R)}{P_W} \right\rceil).$$

P_R and P_W denote the reader's and the writer's period, respectively. For simplicity, we set the task critical times to be the same as the task periods.

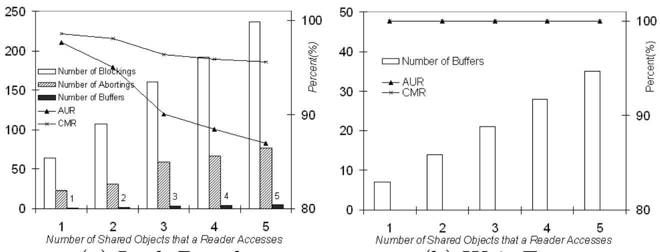
C is the reader's worst-case execution time, and C_R is the time needed to perform a read operation.

Table 1: Experimental Parameters for Tasks

Name	*P(msec)*	*C(msec)*	*Shared Objects*
Writer1	100	10	r_1
Writer2	100	10	r_2
Writer3	100	10	r_3
Writer4	100	10	r_4
Writer5	100	10	r_5
Reader1	900	100	r_1 r_2 r_3 r_4 r_5
Reader2	1000	100	r_2 r_3 r_4 r_5 r_1
Reader3	1100	100	r_3 r_4 r_5 r_1 r_2
Reader4	1200	100	r_4 r_5 r_1 r_2 r_3
Reader5	1300	100	r_5 r_1 r_2 r_3 r_4

Table 1 shows our task parameters. We consider a task set of 10 tasks, which includes 5 readers and 5 writers, mutually exclusively sharing at most 5 data objects. We set C_R for each object to be approximately 20% of C for each reader, which implies that tasks share large amounts of data. We allow a reader to read from 1 object to at most 5 objects. We also vary the number of readers from 1 to 5. The minimum buffer size needed for the wait-free protocol is calculated by Algorithm 1. The actual amount of memory needed is the buffer size multiplied by the message size in bytes.

For each experiment, we generate approximately 16,000 tasks and measure the AUR and the CMR (or critical time meet ratio) under RUA's and DASA's lock-based and wait-free versions, where CMR is the ratio of the number of tasks that meet their critical times to the total number of task releases. The performance comparison between both are shown with the varying number of shared objects and the varying number of tasks.



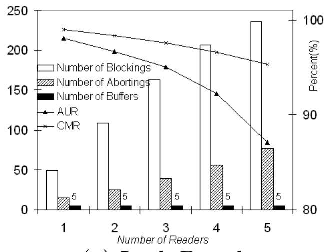
(a) Lock-Based (b) Wait-Free

Figure 4: Performance of Lock-Based and Wait-Free DASA Under Increasing Number of Shared Objects

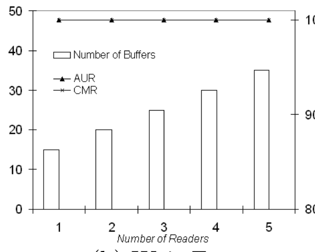
Figure 4 and Figure 5 show lock-based and wait-free DASA's AUR and CMR (on the right-side y-axes), under increasing number of shared data objects and under increasing number of reader tasks, respectively. The figures also show (on the left-side y-axes) the number of buffers used by lock-based and wait-free, as well as the number of task blockings' and task abortions that occur under lock-based.

From Figures 4(a) and 5(a), we observe that the AUR and CMR of lock-based DASA decrease as the number of shared objects and number of readers increase. This is because, as the number of shared objects and readers increase, greater number of task blockings' occurs, resulting in increased sojourn times, critical time-misses, and consequent abortions.

Wait-free DASA is not subject to this behavior, as Figures 4(b) and 5(b) show. Wait-free DASA achieves 100% AUR and CMR even as the number of shared objects and readers increase. This is because, wait-free eliminates task blockings'. Consequently, the algorithm produces a critical time (or deadline)-ordered schedule, which is optimal for under-load situations (i.e., all critical times are satisfied). Thus, all tasks complete before their critical times, and the



(a) Lock-Based

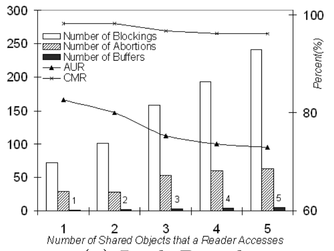


(b) Wait-Free

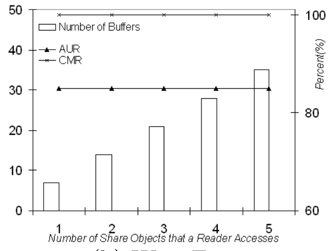
Figure 5: Performance of Lock-Based and Wait-Free DASA Under Increasing Number of Readers

algorithm achieves the maximum possible total utility.

However, the number of buffers needed for wait-free (calculated by Algorithm 1) increases as the number of objects and readers increase. But Algorithm 1 is space-optimal — the needed buffer size is the absolute minimum possible.



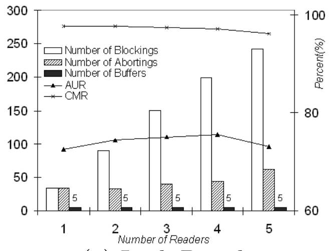
(a) Lock-Based



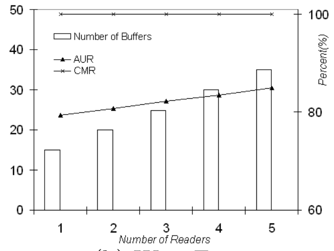
(b) Wait-Free

Figure 6: Performance of Lock-Based and Wait-Free RUA Under Increasing Number of Shared Objects

Similar to the DASA experiments, we compare the performance of RUA's lock-based and wait-free versions. Since RUA allows arbitrarily-shaped TUFs, we consider a heterogenous class of TUF shapes including step, parabolic, and linearly-decreasing. The results are shown in Figures 6 and 7. We observe similar trends as that of DASA's: Lock-based RUA's AUR and CMR decrease as the objects and readers increase, due to increased task blockings'.



(a) Lock-Based



(b) Wait-Free

Figure 7: Performance of Lock-Based and Wait-Free RUA Under Increasing Number of Readers

Similar to DASA, wait-free sharing alleviates task blockings' under RUA. Consequently, RUA produces optimal-schedules during under-load situations in terms of meeting all task critical times, and thus yields 100% CMR.

RUA with wait-free sharing yields greater AUR than that under lock-based, as objects and readers increase. However, unlike DASA, RUA does not yield 100% AUR with wait-free, because tasks accrue different utility depending upon when they complete, even when they do so before task critical times (due to non-step shapes). Further, the algorithm does not necessarily complete tasks at their optimal times as that scheduling problem is NP-hard (and RUA is a heuristic).

6. CONCLUSION

In this paper, we introduce for the first time, wait-free synchronization for UA real-time scheduling. We develop wait-free versions of two UA algorithms, RUA and DASA, using the space-optimal wait-free protocol for the single-writer/multiple-reader problem. We establish upper bounds on the maximum possible increase in activity utility that is possible with wait-free, compared to their lock-based counterparts. Our implementation measurements on a POSIX RTOS show that during under-loads, wait-free algorithms yield optimal utility for step TUFs and significantly higher utility (than lock-based) for non-step TUFs.

Several aspects of the work are directions for further research. Examples include extending the results to the multiple-writer/multiple-reader problem and overload situations.

7. REFERENCES

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

[2] 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.

[3] H. Cho, B. Ravindran, and E. D. Jensen. A space-optimal, wait-free real-time synchronization protocol. In *IEEE ECRTS*, 2005.

[4] R. Clark, E. D. Jensen, et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, pages 353–362, April 1999.

[5] R. K. Clark. *Scheduling Dependent Real-Time Activities.* PhD thesis, CMU CS Dept., 1990.

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

[7] 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.

[8] H. Kopetz and J. Reisinger. The non-blocking write protocol nbw. In *IEEE RTSS*, pages 131–137, 1993.

[9] 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.

[10] 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.

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