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

Hyeonjoong Cho[⋆], Binoy Ravindran[⋆], and E. Douglas Jensen[‡]

[⋆]ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{hjcho,binoy}@vt.edu

[‡]The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

June 15, 2007

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 an 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 lower and upper bound on the total accrued utility that is possible with lock-free and lock-based sharing. We confirm our analytical results with a POSIX RTOS implementation.

*A preliminary version of this paper appeared as "Lock-Free Synchronization for Dynamic Embedded Real-Time Systems," H. Cho, B. Ravindran, and E. D. Jensen, *ACM Design, Automation, and Test in Europe (DATE), Real-Time Systems Track*, pages 438-443, March 2006.

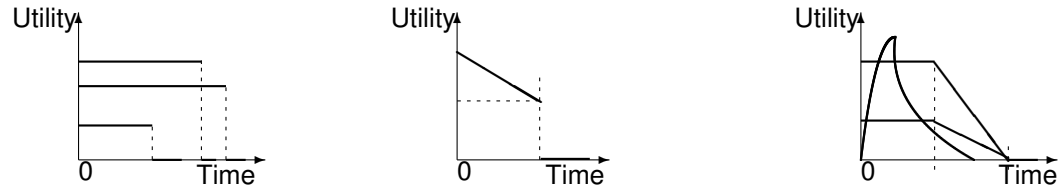
1 Introduction

Embedded real-time systems that are emerging in many domains such as robotic systems for planetary exploration (e.g., NASA/JPL's Mars Rover [10]) and battle management systems for defense (e.g., airborne trackers [8]) 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 require 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 the time constraints (at this point we speak of the special case of deadlines) of all activities is impossible, requiring a scheduling optimality criterion more sophisticated than simply meeting all deadlines. The urgency of an activity as expressed by its deadline is generally orthogonal to the relative importance of the activity—-e.g., the most urgent activity can be the least important, the most urgent can be the most important, etc. Hence when overloads occur, completing as many as possible of 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 underloads, 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) [15] that express the utility of completing an activity as a function of that activity's completion time. We specify a 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.

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,



(a) Some Step TUFs (b) MITRE/TOG AWACS TUF [8] (c) GD/CMU BM/C2 TUFs [19]

Figure 1: Example TUF Time Constraints

increases) with completion time. This is in contrast to deadlines, where a positive (conventionally, unit) utility is accrued for completing the activity anytime before the deadline, after which zero, or infinitively negative utility is accrued. Figure 1 shows examples of such time constraints from two real applications [8]. 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 for downward step TUFs (see algorithms in [22]) default to EDF during underloads, since EDF satisfies all deadlines during underloads. Consequently, they obtain the maximum total utility during underloads. 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.

1.1 Shared Data and Synchronization

Most embedded real-time systems necessarily have (logically) concurrent and conflicting needs to access shared resources that can be only sequentially shared. The means by which these resource contentions are resolved directly affect the system's timeliness properties. This is as true for shared data objects as it is for processors. Mechanisms that resolve such contention for shared data objects can be broadly classified into: (1) lock-based schemes—e.g., [23], see algorithms in [22]; and (2) non-blocking schemes including wait-free protocols (e.g., [6, 14]) and lock-free protocols (e.g., [4, 16]).

Lock-based protocols serialize accesses to sequentially shared objects by mutual exclusion, resulting in reduced concurrency [4]. Further, many lock-based protocols typically incur additional run-time overhead due to scheduler activations that occur when activities request locked objects [22, 23]. Also, deadlocks can occur when lock holders crash, causing indefinite starvation to blockers. Many priority-based (e.g., real-time) lock-based protocols also require a' priori knowledge of the priority ceilings of locks [23], which may be difficult to obtain for dynamic applications, resulting in reduced flexibility [4]. These drawbacks have motivated research on non-blocking objects in embedded real-time systems.

Lock-free objects guarantee that some object operations will complete in a finite number of steps. Consequently, other operations may have to retry an arbitrary number of times. Instead of acquiring locks, a 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 counter-productive to timeliness optimization. Prior research [4, 11, 13] has shown how to mitigate or bound these time costs.

In [4], Anderson *et al.* show how to bound the retry loops of lock-free protocols through judicious real-time scheduling. In [2], lock-free objects on single processors and multiprocessors using quantum-based scheduling are presented. That 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. Efficient lock-free objects, such as queues and stacks, have been presented in [20, 21]. In [26], Valois presents lock-free linked lists. Treiber presents lock-free stack algorithms in [25].

On the other hand, wait-free objects guarantee that 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 (e.g., some wait-free protocols require multiple buffers for their operations.)

In [3], 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 accesses. In [16], Kopetz *et al.* present an analysis on a wait-free synchronization in real-time systems. This

work was later improved by Chen *et al.* in [6], and subsequently by Huang *et al.* in [14] and by Cho *et al.* in [7]. However, wait-free synchronization sometimes requires a' priori knowledge of the maximum number of jobs that may arrive at any given time (for ensuring atomicity), because the identities of all jobs must be known to all jobs in wait-free algorithms [7, 14, 16]. Obtaining this a' priori knowledge is difficult for our motivating applications.

As shown in [4], lock-free implementations always perform better than wait-free ones when wait-free objects are implemented using Herlihy-like helping scheme. However, the other wait-free object implementations may be superior to lock-free ones. In this paper, we do not consider wait-free schemes, and focus on lock-free synchronization.

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* (or UAM) [12]. 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. Past work on lock-free sharing upper bounds the retries under restrictive arrival models like periodic [4] — 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 serializable concurrent object sharing. We focus on the *Resource-constrained Utility Accrual* (or RUA) scheduling algorithm [27] for that model. RUA allows arbitrarily-shaped TUFs and concurrent object sharing using locks. For the special case of downward step TUFs, no object sharing, and underloads, RUA defaults to EDF.

We derive an upper bound on lock-free retries of RUA 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*[1] are shorter under lock-free RUA than under lock-based, for the UAM. From this result, we establish the lower and upper bound on activities' total utility under lock-free and lock-based RUA. Further, we implement lock-free and lock-based RUA on a POSIX RTOS. Our implementation measurements strongly validate our analytical results.

The dynamic real-time application environment which this work focuses on has time frames that allow more sophisticated and thus higher cost resource management than traditional small scale static real-time applications do. As always, resource management costs must be justified by providing corresponding application and system operational effectiveness. For this work, operational effectiveness of the system is improved by reducing the cost of scheduling with TUF/UA algorithms which in turn increase application operational effectiveness. For an analogy, consider the logistics application context, such as scheduling delivery trucks. Very sophisticated but high execution time resource management algorithms are often used, but result in improved operational effectiveness in terms of lower truck fuel consumption, etc.

Our major finding is that lock-free synchronization can improve the cost of (lock-based) RUA's scheduling decisions by eliminating the computation of activity dependency chains (that arise due to lock-based object sharing conflicts), and by allowing activities to force roll-backs in their peers rather than rescheduling when encountering those conflicts. Lock-based RUA views the dependency chain of an activity (i.e., the activity together with all of its dependents, which is $O(n)$-long in the worst case) as a logically single entity to respect the dependencies, effectively constituting its scheduling unit, and thus suffers from increased scheduling cost. With lock-free synchronization, dependencies are avoided, thereby reducing an activity's dependency chain to include just the activity itself.

The rest of the paper is organized as follows: We describe the task model that we consider

[1] An activity's sojourn time is the time between the activity's arrival and its completion. We use sojourn time (often used in queuing theory), instead of "response time," as that term sometimes causes confusion—e.g., real-time practitioners sometimes use response time to mean interrupt response time, which is a subset of the sojourn time.

in Section 2. In Section 3, we overview lock-based RUA, describe the key steps of the algorithm, identify those algorithm steps that can be improved through lock-free synchronization, and thereby establish the premise for lock-free RUA. In Section 4, we derive an 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 5. Section 6 discusses our implementation experience. We conclude the paper in Section 7.

2 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 [12]. Jobs may arrive simultaneously. The periodic model is a special case of UAM with $\langle 1, 1, W_i \rangle$.

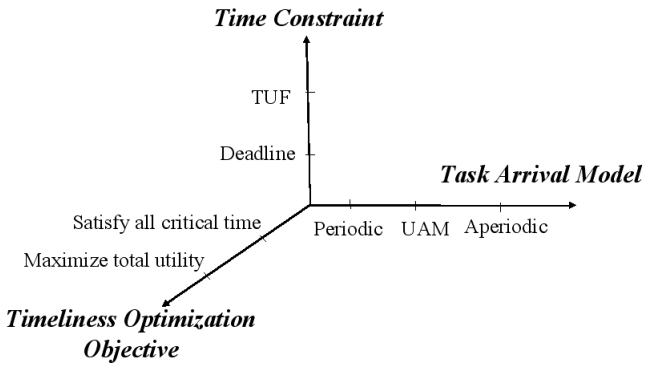


Figure 2: Three Dimensions of Task Model

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. TUFs subsume deadlines as a special case (i.e., binary-valued, downward step TUFs). All jobs of a task have the same TUF. $U_i(\cdot)$ denotes task T_i's TUF; thus, the 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 drops to zero utility. In general, the TUF has zero utility after the critical 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 the timeliness optimization objective. Example objectives include satisfying all critical times for step TUFs during underloads, and maximizing total accrued utility for arbitrarily-shaped TUFs during underloads and overloads. Our model includes the UAM, TUFs, and the objective of maximizing total utility.

Finally, the resource model we consider here does not allow nested critical sections, which may cause deadlocks in lock-based synchronization, and on the other hand, complicates implementation in lock-free synchronization.

3 Overview of Lock-Based RUA

We first overview RUA, which allows concurrent object sharing using locks [27] (the complete algorithm details can be found in [27]). Note that RUA allows nested critical sections. Thus deadlocks can occur, and the algorithm uses a deadlock detection and resolution mechanism. In describing RUA, we describe all aspects of the algorithm including its deadlock detection and resolution mechanism, for completeness and consistency with the description in [27]. Note that RUA's deadlock detection/resolution mechanism will not be triggered in the context of this paper which excludes nested critical sections. Thus, in comparing lock-based RUA with lock-free RUA, which we do later in Section 5, we disallow nested critical sections and exclude lock-based RUA's deadlock detection/resolution mechanism, for an apples-to-apples comparison. We will recall this aspect in Section 5 (so that the point is not lost).

RUA [27] targets dynamic applications, where activities are subject to arbitrary arrivals and re-

source 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.[2] 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.

The algorithm's scheduling events include job arrivals, job departures, lock and unlock requests, expiration of job critical times.[3] The major steps of the algorithm include computing job dependency chains, determining job potential utility densities, deadlock detection and resolution, and constructing feasible schedules. These are explained in the subsections that follow.

3.1 Computing Job Dependency Chains

When RUA is invoked, it first builds the *dependency chain* of each job—that arises due to mutually exclusive resource sharing—by following the chain of resource request and ownership.

For example, a job T_1 may request for a resource R_1 at time t (a scheduling event). Resource R_1 may be locked by another job T_2 at time t. Thus, T_1 is dependent upon T_2 at time t. Job T_2 may already be dependent on another job T_3 for a resource R_2 at time t. Suppose that T_3 is not dependent upon any other jobs at time t. Then, at time t, T_1's dependency chain is the sequence $\langle T_3, T_2, T_1, \rangle$, implying that T_3 must be executed before T_2, which must be executed before T_1, to respect the chained mutual exclusion dependency.

Note that the execution of T_3 and T_2 need not be till their completions, since T_3 and T_2 may release their locked resources before their completions. Further, T_3's execution need not be immediately before T_2, and T_2's execution need not be immediately before T_1. The dependency chain

[2]Note that with lock-free RUA, we exclude physical resources and only focus on logical resources—e.g., data objects.

[3]A "scheduling event" is an event that invokes the scheduling algorithm.

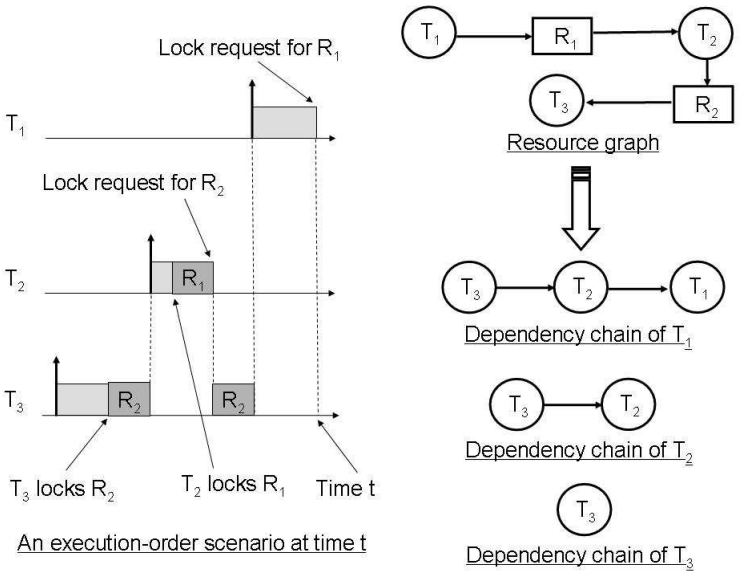


Figure 3: An Example Scenario of Computing Dependency Chains

only implies the *before-after* precedence that must be respected at time t in constructing a schedule at time t, given the current system knowledge at t. In the future, new dependencies may arise, which will cause new scheduling events, and the dependency chains must be recomputed.

Similarly, T_2's dependency chain is the sequence $\langle T_3, T_2 \rangle$, and T_3's dependency chain includes only T_3 since T_3 is not dependent upon anyone. Figure 3 illustrates this example scenario of computing dependency chains.

3.2 Computing Job Potential Utility Densities

Armed with the job dependency chains, RUA next computes the job *potential utility densities* (or PUDs). The PUD of a job measures the amount of utility that can be accrued per unit time by executing the job and the job(s) that it depends upon, given the current system knowledge.

A job T_i is said to be dependent upon another job T_j if T_i needs a resource that is held by T_j (i.e., T_i is *directly* dependent upon T_j), or if T_i needs a resource that is held by another job T_k, which in turn needs a resource that is held by T_j (i.e., T_i is *transitively* dependent upon T_j). Here, T_j and T_k are referred to as the *dependent* jobs of T_i.

To compute a job T_i's PUD at time t, RUA considers T_i's estimated completion time (denoted as t_f) and the utility that can be obtained by executing T_i and its dependent jobs. For each job T_j that is in T_i's dependency chain and needs to be completed before executing T_i, T_j's estimated completion time is denoted as t_j.[4] PUD of T_i is then computed as: $U_i(t_f) + \frac{\sum_{T_j \in T_i.Dep} U_j(t_j)}{t_f - t}$.

Thus, the PUD calculation for a job reflects the fact that executing the sequence of the job and its dependents will require a total time equal to the sum of the individual times and will yield a total utility equal to the sum of the individual utilities. A job's PUD, thus measures the job's "return on investment."

Note that the term "potential" is used in PUD. This is because, the PUD calculation reflects the utility that can possibly be obtained from the aggregate computation, given the current system knowledge.[5] It is quite possible that future situations (e.g., new dependencies, execution overruns) may negate the chance to accrue any utility from the aggregate computation.

3.3 Deadlock Detection and Resolution

The algorithm then checks for deadlocks, caused from nested critical sections, by detecting the presence of a cycle in the dependency chain — a necessary condition for deadlocks. Deadlocks are resolved by aborting that job in the cycle, which will likely contribute the least utility. (We summarize RUA's abortion model in Section 3.5.) RUA adopts a deadlock detection and resolution strategy (as opposed to deadlock avoidance or prevention) precisely due to the dynamic nature of systems of interest — which resources will be needed by which activities, for how long, and in what order is not assumed to be known to the scheduler.

[4]Completion time is only an estimate, since the job execution times presented to the scheduler, which are used to determine completion times, are only assumed to be estimates. Thus, execution overruns are quite possible.

[5]Actually, the calculated PUD of a job is the highest possible, given the current system knowledge: The PUDs are calculated assuming that the jobs are executed at the current position in the schedule, and that jobs release resources only after their executions complete.

3.4 Constructing Feasible Schedules

After handling deadlocks, RUA sorts jobs in the order of non-increasing PUDs.

The algorithm then starts the process of constructing a schedule, by starting with an empty schedule. Each job is then examined in the non-increasing PUD order, and inserted with its dependents into a copy of the schedule called the "tentative schedule." To insert a job J and its dependents into the tentative schedule, RUA proceeds from the tail of J's dependency chain (i.e., starts with J) and moves toward the head of the chain (i.e., ends with the job that is farthest from J in the chain). The insertion is done in the order of the job critical times, earliest-critical-time-first (or ECF), thereby maintaining the tentative schedule in the ECF order. (We discuss the insertion process in Section 3.4.1.)

After inserting a job and its dependents into the tentative schedule, RUA checks the feasibility of the tentative schedule with respect to satisfying all job critical times. If the schedule is feasible, the tentative schedule is updated as the new schedule. If infeasible, the tentative schedule is discarded (thereby rejecting the inserted job and its dependents).

The algorithm repeats this process until all jobs in the sorted PUD list are examined, and selects the job at the head of the schedule for execution.

3.4.1 Inserting a Job and its Dependents

When a job and its dependents are inserted into the tentative schedule in the ECF order, the ECF order of the job and its dependents can be inconsistent with the job's dependency order. This must be resolved during insertion. For example, let a job T_1's dependency chain is the sequence $\langle T_2, T_1 \rangle$. When T_1 is inserted into the tentative schedule, two cases can arise:

(1) Case 1: the ECF order is consistent with the dependency order—i.e., $C_2 < C_1$;

(2) Case 2: the ECF order is inconsistent with the dependency order—i.e., $C_2 > C_1$.

Case 1 poses no problem, since T_2 must precede T_1 to respect the dependency. For Case 2, RUA inserts T_2 before T_1 to respect the dependency, and updates C_2 as $C_2 = C_1$. The updated critical

times are used for testing schedule feasibility. Figure 4 illustrates this.

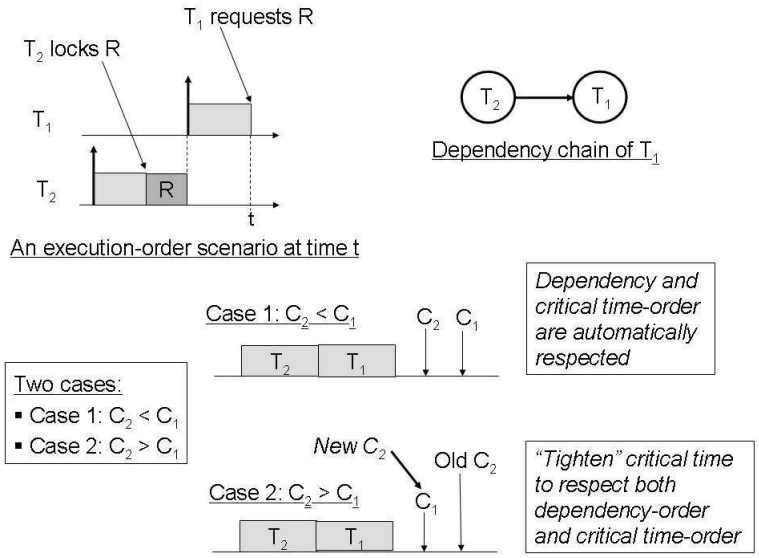


Figure 4: Resolving Conflict Between Critical-Time-Order and Dependency-Order

When a job and its dependents are inserted into the tentative schedule, it is possible that some of the dependents may already exist in the schedule, since they also may be dependents of other jobs which were previously inserted into the tentative schedule. If so, the dependency order must be reestablished to ensure that the previously inserted dependents are also inserted before the job that is currently being inserted.

For example, let the dependency chains of three jobs T_1, T_2, and T_3 are the sequences $\langle T_1 \rangle$, $\langle T_1, T_2, \rangle$, and $\langle T_1, T_3 \rangle$, respectively (see Figure 5). Let the non-increasing PUD order of the jobs is T_2, T_1, T_3. Thus, T_2 is inserted into the tentative schedule first with its dependent T_1, and tested for feasibility. Suppose that this tentative schedule is feasible and is updated as the new schedule (i.e., $\langle T_1, T_2 \rangle$).

The next job to be inserted is T_3 and its dependent T_1 (though T_1 is the next job in the PUD list, it is already inserted as part of T_2's insertion). Thus, RUA makes a copy of the tentative schedule $\langle T_1, T_2 \rangle$ and starts inserting each job in T_3's dependency chain $\langle T_1, T_3 \rangle$, from tail (T_3) to head (T_1), at their critical time positions. After inserting T_3, RUA is now ready to insert T_1. Since T_1 has

already been inserted as part of T_2's insertion and before T_2, it must now be ensured that T_1 is also inserted before T_3. There are two possible cases now:

(1) Case 1: $C_1 < C_3$;

(2) Case 2: $C_1 > C_3$.

Case 1 automatically ensures that T_1 is also inserted before T_3 (besides T_2). If Case 2 occurs, RUA removes T_1 from the tentative schedule and inserts it before T_3 and updates its critical time C_1 as $C_1 = C_3$. This scenario is illustrated in Figure 5.

If the tentative schedule is feasible, then it is updated as the new schedule (i.e., $\langle T_1, T_3, T_2 \rangle$). Otherwise, the tentative schedule is discarded and the previous schedule is retained (i.e, $\langle T_1, T_2 \rangle$).

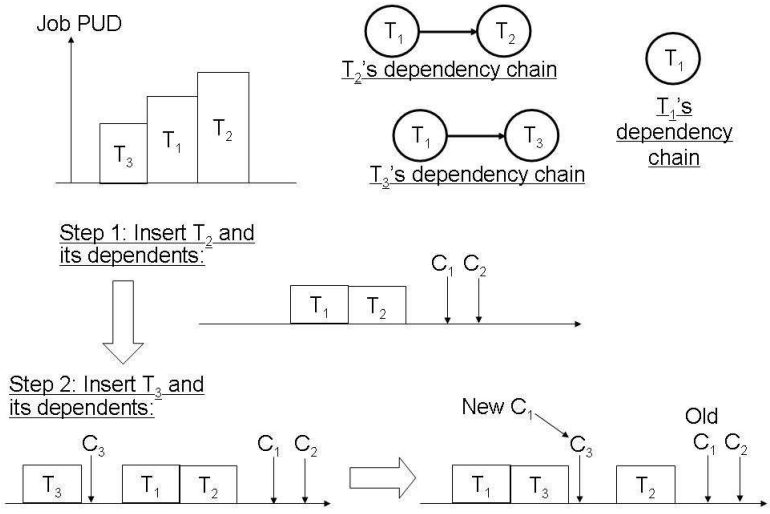


Figure 5: Removal and Reinsertion of Jobs During Schedule Construction

Note that RUA examines all jobs during the schedule construction process at each scheduling event, including those that are ready to run, as well as those that are blocked awaiting access to shared objects. This is because, many job scheduling parameters change at each scheduling event, due to the dynamic nature of the application. These parameters include dependencies (that may change due to new lock requests and lock releases) and remaining execution time estimates (that may change due to changes in application's operating environmental context). Consequently, RUA

examines all jobs to construct a schedule that will maximize the application's total utility as much as possible, given the current situation. Thus, RUA is designed as an integrated scheduler and resource manager.

If a job's critical time is reached and its execution has not been completed, the job is immediately aborted. We discuss the abortion model in Section 3.5.

During underloads, 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 an ECF (or EDF)-ordered schedule, yielding the maximum possible total utility, as that ordering is optimal for underloads and will meet all job critical times. During overloads, the algorithm will produce a feasible schedule by greedily favoring as many "high return" jobs as possible, will reject some low-return jobs, and thereby seeks to obtain as high total utility as possible.

3.5 Job Abortions

To abort a job for resolving a deadlock, RUA raises an *abort-exception* when the deadlock is detected and the job has been selected for abortion, during the scheduling process.

Abortion of a job due to the expiration of the job's critical time is handled asynchronously: When the job arrives (a scheduling event), RUA will set a timer that will expire when the job's critical time is reached, besides constructing a schedule and dispatching a job. When the critical time later expires (a scheduling event), the timer goes off, and the scheduler is awakened. If the job has not completed its execution at that time, RUA immediately raises an abort-exception (as that for deadlock resolution).

Once an abort-exception is raised, the scheduler immediately releases the job's exception handler and executes it. The handler is assumed to perform whatever compensations and recovery actions are necessary to avoid inconsistencies—e.g., rolling back logical and physical resources that are held by the job to safe states. The handler is also assumed to perform actions that are required to ensure the safety and stability of the external state.

Thus, a job is aborted by immediately executing the job's exception handler. Once the handler completes its execution, the system is assumed to be in a safe and consistent state, and other jobs can safely proceed with their execution. RUA's model of aborted executions follows that of [9], and is similar to what is typically employed in transactional paradigms (e.g., [24]).

With lock-free RUA, job abortions also follow the same model. Since deadlocks do not occur with lock-free RUA, abortions are necessary only when job critical times expire.[6] Thus, when a job's critical time expires, the timer goes off, and awakens the scheduler. If the job has not completed its execution at that time, (lock-free) RUA will immediately raise an abort-exception, and execute the job's handler.

3.6 Asymptotic Cost

With n jobs, RUA's asymptotic cost is $O(n^2 logn)$, for the single-unit resource model, and with or without nested critical sections. The steps of RUA that dominate its computational cost include:

(1) Computing the dependency chain of each job. This costs $O(n^2)$ for all jobs, since the dependency chain of a single job can contain all other jobs in the worst case. Thus, the dependency chain of a job can contain $O(n)$ jobs; so the total cost for n jobs is $O(n^2)$.

(2) Computing the PUD of each job. This costs $O(n^2)$ for all jobs, since the entire dependency chain of a job (which is $O(n)$ -long in the worst case) must be inspected to determine the PUD of the job.

(3) Testing dependency chains for a deadlock. For a single job, this costs $O(n)$, and is done while the dependency chain of the job (which is $O(n)$ -long in the worst case) is computed. This is done by checking whether the next object requested by a job in the chain is held by another job in the chain (i.e., whether the chain leads to a cycle). For n jobs, the cost is $O(n^2)$.

(4) Sorting jobs by PUD. This costs $O(nlogn)$.

[6] In this paper, abortions under lock-based RUA are also necessary only when job critical times expire, since nested sections are excluded, consequently avoiding deadlocks.

(5) Examining each job in decreasing PUD-order, inserting the job and its dependents into a tentative schedule in the critical-time order, respecting dependencies, and testing for schedule feasibility. For a single job J, this entire process costs $O(nlogn)$, because each job in J's dependency chain (which is $O(n)$-long in the worst case) must be inserted into the tentative schedule in the critical-time order, respecting the job dependencies. The process of inserting a job into a tentative schedule in the ECF order and respecting dependencies costs $O(logn)$, as this requires operations such as lookup, remove, and insert on an ordered list (i.e., the schedule), each of which costs $O(logn)$. For n jobs in the dependency chain, the cost is therefore $O(nlogn)$. Thus, for n jobs in the ready queue, the total cost becomes $O(n^2logn)$.

Summing up the costs, the total algorithm cost is therefore $O(n^2logn)$.

Note that the cost of the algorithm is dominated by the cost of Step 5, which itself costs $O(n^2logn)$. All other steps including the deadlock detection step (i.e., Step 3) cost less. Thus, even if nested critical sections are not allowed (which will allow the algorithm to skip Step 3), the algorithm will still cost $O(n^2logn)$.

RUA's cost is higher than that of many traditional real-time scheduling algorithms. However, this high cost is justified for applications with longer execution time magnitudes such as those that we focus on 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.

The major speed bottleneck of the algorithm that cross-cuts across all algorithm steps is the algorithm's concept of an "aggregate" computation — i.e., a job and its dependents, the size of which is $O(n)$-long in the worst case. At its core, the algorithm views this aggregate computation as a logically single entity to respect the dependencies, measures the "worth" of an aggregate by computing its PUD, examines an aggregate when it becomes the highest-PUD aggregate in the unexamined aggregate pile, and then determines whether to include it in the schedule.

Piecemeal speed optimizations of the algorithm are possible—e.g,., the step of testing for schedule feasibility can be optimized through randomization as in [17] (with concomitant trade-

offs). However, a cross-cutting optimization, which is needed to improve the total algorithm speed, is difficult without reducing the size of the aggregate computation. This is precisely what we accomplish through lock-free synchronization, where dependencies are avoided, thereby effectively reducing the size of the aggregate computation to a single job. Of course, this comes with concomitant tradeoffs. Thus, one of our goals is to identify those tradeoffs and understand when such a lock-free RUA would be better and more effective than lock-based RUA.

4 Bounding Retries Under the UAM

4.1 Preemptions Under UA Schedulers

In an effort to catalogue various preemptive scheduling algorithms, Carpenter *et al.* categorized scheduling disciplines (for periodic or sporadic tasks) according to whether task priorities are: (1) static, (2) dynamic but fixed within a job (job-level dynamic), or (3) fully-dynamic [5]. In [1], Anderson *et al.* showed examples of each type including: (1) rate-monotonic (RM), (2) earliest-deadline-first (EDF), (3) least-laxity-first (LLF) scheduling.

Under static and job-level dynamic priority schedulers, a job can be preempted at most once by another 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. On the other hand, fully-dynamic priority schedulers allow two jobs to preempt each other more than once as scheduling events happen. In the sense that execution eligibility of a job dynamically changes, the behavior of UA schedulers such as RUA is similar to those of fully-dynamic priority schedulers. The fully-dynamic priority schedulers allow the mutual preemption illustrated in Figure 6, i.e., two jobs continue preempting each other.

Hence, the maximum number of preemptions under RM or EDF that a job may suffer can be bounded by the number of releases of other jobs during a given time interval (see [3]), whereas the maximum number of preemptions that a job may experience under RUA is bounded by the number

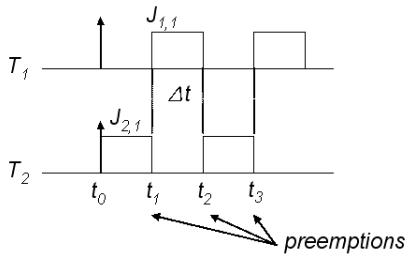


Figure 6: Mutual Preemption Under RUA

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 [22], because it is impossible for more preemptions to occur than scheduling events. Especially, when lock-free synchronization and RUA are considered, the scheduling events only include job arrivals and departures, but not lock and unlock requests.

4.2 Bounding Retries

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 retries of RUA under this model.

Theorem 2 (Lock-Free Retry Bound Under UAM)**.** *Let jobs arrive under the UAM* $< 1, a_i, W_i >$ *and be 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 7, J_i is released at time t_0 and has the 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:

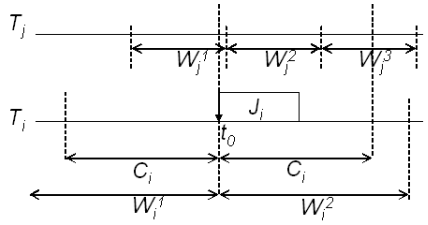


Figure 7: Interferences Under UAM

Case 1 (Scheduling events from other tasks): To account for the worst-case event occurrence where the maximum number of scheduling events of T_j occurs within W_i, 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 $(t_0 + C_i)$. (In the proof, W_j^k denotes the k^{th} window of T_j.) 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 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 where the maximal scheduling events by T_i occurs within W_i, 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_i 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_i 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[7] by other jobs, the lock-free object accessing time, and f_i retry time.

5 Lock-Based versus Lock-Free

We now formally compare lock-based and lock-free sharing. As indicated in Section 4, we do not consider nested critical sections. For an apples-to-apples comparison, we disallow nested sections for lock-based RUA and exclude its deadlock detection mechanism. As described in Section 3.6, doing so does not reduce the asymptotic cost of lock-based RUA, since that cost hinges on algorithm steps other than deadlock detection (i.e., it depends on Step 5 in Section 3.6).

We formally compare lock-based and lock-free sharing in terms of 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 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.

[7] Interference time of a task is the time that the task has to wait to get the processor for execution (since its arrival) as the processor is busy executing other tasks (since those tasks were deemed by the scheduler as more eligible to execute). Thus, for a task T_i, its interference time is the time spent on executing other tasks when task T_i is runnable.

The worst-case sojourn time of a job with lock-based is $u_i + I_i + r \cdot m_i + B_i$, where B_i is the worst-case blocking time and I_i is the worst-case interference time. In [27], it is shown that a job J_i under RUA can be blocked for at most $min(m_i, n_i)$ times, where n_i is the number of jobs that could block J_i and m_i is the number of objects that can be used to block J_i. Thus, B_i can be computed as $r \cdot min(m_i, n_i)$. On the other hand, the worst-case sojourn time of a job with lock-free is $u_i + I_i + s \cdot m_i + R_i$, where R_i is the worst-case retry time. R_i can be computed as $s \cdot f_i$ by Theorem 2. Thus, the difference between $r \cdot m_i + B_i$ and $s \cdot m_i + R_i$ is the sojourn time difference between lock-based and lock-free.

Based on the notation, we formally compare lock-based and lock-free sharing under our model.

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

$$\begin{cases} \frac{s}{r} < \frac{2}{3}, & \textit{when } m_i \leq n_i \\ \frac{s}{r} < \frac{m_i+n_i}{m_i+3a_i+2x_i}, & \textit{when } m_i > n_i, \end{cases}$$

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

Theorem 3 states that no matter whether m_i is greater than n_i, $\frac{s}{r} < 1$ is necessary for jobs to obtain a shorter sojourn time under lock-free (since $\frac{m_i+n_i}{m_i+3a_i+2x_i}$ is less than one when $m_i > n_i$). Especially when $m_i \leq n_i$, $\frac{s}{r} < 2/3$ is sufficient for jobs to obtain a shorter sojourn time under lock-free. Shorter sojourn times always yield higher utility with non-increasing TUFs, but not always with increasing TUFs. However, it is likely to improve performance at the system-level even with increasing TUFs, because each job can save more CPU time for other jobs.

We now describe the proof of the theorem.

Proof. Let X denote $r \cdot m_i + r \cdot min(m_i, n_i)$ and Y denote $s \cdot m_i + s \cdot f_i$ to compare sojourn times. n_i is bounded by the total number of jobs that may happen during executing J_i so that n_i is at most $2a_i + \sum_{j=1,j\neq i}^{N} a_j \left(\lceil C_i/W_j \rceil + 1 \right) = 2a_i + x_i$. 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 $m_i \leq n_i$, X becomes $2r \cdot m_i$. Y is $s(m_i + 3a_i + 2x_i)$. Therefore:

$$Y = \frac{s}{2r}X + s\left(3a_i + 2x_i\right),$$

where $X = 2rm_i \leq 2rn_i \leq 2r\left(2a_i + x_i\right)$. 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 $X = \frac{2rs(3a_i+2x_i)}{2r-s}$. Hence, when $\frac{2rs(3a_i+2x_i)}{2r-s} < X$, X is greater than Y, where $X \leq 2r\left(2a_i + x_i\right)$. This leads to:

$$\frac{r}{s} > \frac{1}{2} + \frac{3a_i + 2x_i}{2m_i}.$$

Since $m_i \leq 2a_i + x_i$,

$$\frac{r}{s} > \frac{1}{2} + \frac{3a_i + 2x_i}{2m_i} \geq \frac{1}{2} + \frac{3a_i + 2x_i}{4a_i + 2x_i} \geq \frac{3}{2} - \frac{1}{4 + 2\frac{x_i}{a_i}}.$$

Therefore, the sufficient condition is $\frac{r}{s} > \frac{3}{2}$.

Case 2: When $m_i > n_i$, X becomes $r\left(m_i + n_i\right)$ and we can obtain $Y = \frac{s}{r}X + s\left(3a_i + 2x_i - n_i\right)$. 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 $X = \frac{rs(3a_i+2x_i-n)}{r-s}$. Therefore,

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

This inequality leads to:

$$\frac{s}{r} < \frac{m_i + n_i}{m_i + 3a_i + 2x_i}.$$

This implies that $\frac{s}{r} < 1$, since $\frac{m_i+n_i}{m_i+3a_i+2x_i} < 1$ considering $n_i \leq 2a_i + x_i$. □

In [3], 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. It also is known that lock-free approaches work particularly well for such simple objects. On the other side, lock-free approaches tend to become more expensive for complicated objects such as transactional memory.

Much smaller access time of lock-free approaches is in line with the results of our experiments in Section 6, where we consider UAM arrival and RUA scheduling. Thus, lock-free sharing is very attractive for UA scheduling as most UA schedulers' object sharing mechanisms have higher time complexity.

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. The AURs of lock-based and lock-free sharing under RUA are bounded as follows:

Lemma 4 (Lock-Free AUR)**.** *Let the i^{th} task's jobs arrive under the UAM, $< l_i, a_i, W_i >$, and be scheduled by RUA. If all jobs are feasible and their TUFs are non-increasing, then the AUR of lock-free sharing, AUR, over time converges into:*

$$\frac{\sum_{i=1}^{N} \frac{l_i}{W_i} U_i(u_i + s \cdot m_i + I_i + R_i)}{\sum_{i=1}^{N} \frac{l_i}{W_i} U_i(0)} < AUR < \frac{\sum_{i=1}^{N} \frac{a_i}{W_i} U_i(u_i + s \cdot m_i)}{\sum_{i=1}^{N} \frac{a_i}{W_i} U_i(0)},$$

where $U_i(t)$ is task i's utility at time t, and N is the number of tasks.

Proof. The AUR is computed as $\sum_{i=1}^{N} n_i \cdot U_i(s_i^{lf}) / \sum_{i=1}^{N} n_i \cdot U_i(0)$, where s_i^{lf} is the sojourn time of T_i and n_i is the number of jobs of T_i completed within a time interval. Under UAM, n_i is within the range of $[l_i \lfloor \frac{\Delta t}{W_i} \rfloor, a_i(\lceil \frac{\Delta t}{W_i} \rceil + 1)]$ in a time interval Δt, i.e., $l_i \lfloor \frac{\Delta t}{W_i} \rfloor$ is the minimum possible jobs, n_i^{min}, and $a_i(\lceil \frac{\Delta t}{W_i} \rceil + 1)$ is the maximum possible jobs, n_i^{max}, within Δt under UAM. The AUR can be rewritten as:

$$AUR = \frac{n_1 U_1(s_1^{lf})}{\sum_{i=1}^{N} n_i U_i(0)} + ... + \frac{n_N U_N(s_N^{lf})}{\sum_{i=1}^{N} n_i U_i(0)}.$$

Each $n_i U_i(s_i^{lf}) / \sum_{i=1}^{N} n_i U_i(0)$ is a increasing function of n_i when $n_i > 0$, and therefore, the maximum value of each n_i yields the maximum AUR and the minimum value of each n_i vice versa. Thus,

$$\frac{n_i^{min} U_i(s_i^{lf})}{\sum_{i=1}^{N} n_i^{min} U_i(0)} \leq \frac{n_i U_i(s_i^{lf})}{\sum_{i=1}^{N} n_i U_i(0)} \leq \frac{n_i^{max} U_i(s_i^{lf})}{\sum_{i=1}^{N} n_i^{max} U_i(0)}.$$

When a long time interval Δt is considered, since $n_i^{max} = a_i(\lceil \frac{\Delta t}{W_i} \rceil + 1) < a_i(\frac{\Delta t}{W_i} + 2)$,

$$AUR < \frac{\sum_{i=1}^{N} \frac{a_i}{W_i} U_i(s_i^{lf})}{\sum_{i=1}^{N} \frac{a_i}{W_i} U_i(0)}.$$

On the other hand, since $n_i^{min} = l_i \lfloor \frac{\Delta t}{W_i} \rfloor > l_i(\frac{\Delta t}{W_i} - 1)$,

$$\frac{\sum_{i=1}^{N} \frac{l_i}{W_i} U_i(s_i^{lf})}{\sum_{i=1}^{N} \frac{l_i}{W_i} U_i(0)} < AUR.$$

The sojourn time s_i^{lf} is within $[u_i + sm_i, u_i + sm_i + I_i + R_i]$. With the non-increasing TUFs, the shortest possible sojourn time $u_i + sm_i$ yields the maximum AUR and the longest possible sojourn time $u_i + sm_i + I_i + R_i$ yields the minimum AUR. □

Lemma 5 (Lock-Based AUR)**.** *Let the i^{th} task's jobs arrive under the UAM, $< l_i, a_i, W_i >$, and be scheduled by RUA. If all jobs are feasible and their TUFs are non-increasing, then the AUR of lock-based sharing, AUR, over time converges into:*

$$\frac{\sum_{i=1}^{N} \frac{l_i}{W_i} U_i(u_i + r \cdot m_i + I_i + B_i)}{\sum_{i=1}^{N} \frac{l_i}{W_i} U_i(0)} < AUR < \frac{\sum_{i=1}^{N} \frac{a_i}{W_i} U_i(u_i + r \cdot m_i)}{\sum_{i=1}^{N} \frac{a_i}{W_i} U_i(0)},$$

where $U_i(t)$ is task i's utility at time t, and N is the number of tasks.

Proof. See the proof of Lemma 4 while replacing s, s_i^{lf}, and R_i with r, s_i^{lb}, and B_i respectively, where s_i^{lb} denotes the sojourn time of T_i under lock-based sharing. □

In the lock-free RUA, the steps (1) and (3) of RUA (Section 3.6) are entirely avoided, as job dependencies do not exist. Further, the cost of step (2) is reduced to $O(n)$, since there is no dependency chain for a job. Finally, the cost of step (5) is also reduced to $O(n^2)$ since the dependency chain is not present. Thus, the lock-free RUA costs $O(n^2)$ in total.

Note that our comparison of lock-free RUA against lock-based RUA is under no nested critical sections. As discussed in Section 3.6, the cost of lock-based RUA without nested sections is still $O(n^2 logn)$.

6 Implementation Experience

We implemented lock-based and lock-free objects with RUA in the *meta-scheduler* scheduling framework [18], which allows middleware-level real-time scheduling atop POSIX

RTOSs. Our motivation for implementation is to verify our analytical results. We used

the 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 [21] in our implementation.

6.1 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.

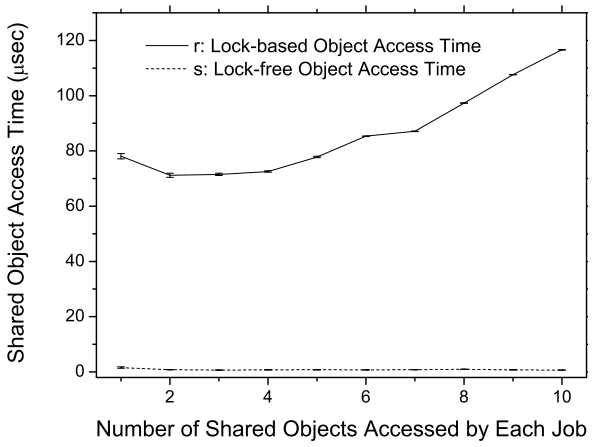


Figure 8: Lock-Based and Lock-Free Shared Object Access Time

Figure 8 shows r and s under increasing number of shared objects accessed by jobs, not al-

lowing any nested critical section.[8] From the figure, we observe that r is significantly larger than s. This is because: (1) RUA's lock-based mechanism is expensive, as shown in Section 3.6; and (2) the meta-scheduler overhead is also playing a big role since

our meta-scheduler implementation is at the application level. (In [18], we experimentally quantify the overhead of the meta-scheduler through a number of metrics.)

As the number of shared objects accessed by each job is increasing, r is increasing. Note that r includes the time for lock-based RUA's resource sharing mechanism.

When $s << r$, Theorem 3 implies that it increases the likelihood of satisfying the sufficient condition under which lock-free is more likely to perform better than 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* (or CML), which was originally defined in [18] for the purpose of evaluating scheduler overhead. 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* (or AL) 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.

Thus, the CML of an ideal EDF scheduler (without any overhead) is 1.0, as EDF is theoretically guaranteed to meet all deadlines (or critical times) during underloads and no dependencies. Similarly, since RUA is equivalent to EDF during underloads and no dependencies, its CML is also equal to 1.0 under those conditions. However, an actual implementation using the meta scheduler framework [18] incurs certain overhead. Furthermore, the scheduler overhead tends to manifest itself for jobs with short execution times. That is, shorter the job execution time, the lower is the CML. Thus, the relationship between CML and average job execution time provides a way of evaluating the impact of r and s on lock-based RUA's and lock-free RUA's performance (and thus the viability of particular implementations of those schedulers).

We exclude object access time in CML, because an ideal implementation of objects for syn-

[8] The error bar around each data point represents 95% confidence interval of that data point.

chronization 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."

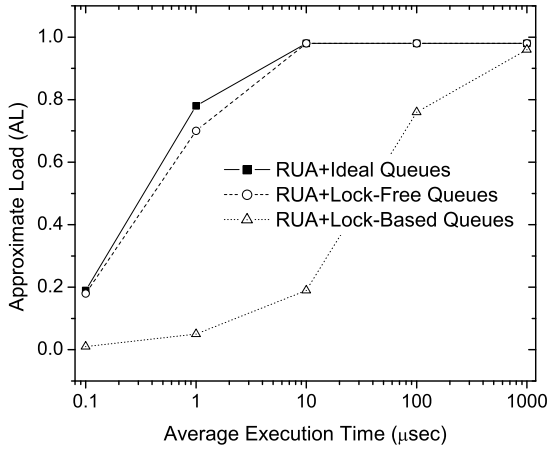


Figure 9: Critical Time Miss Load

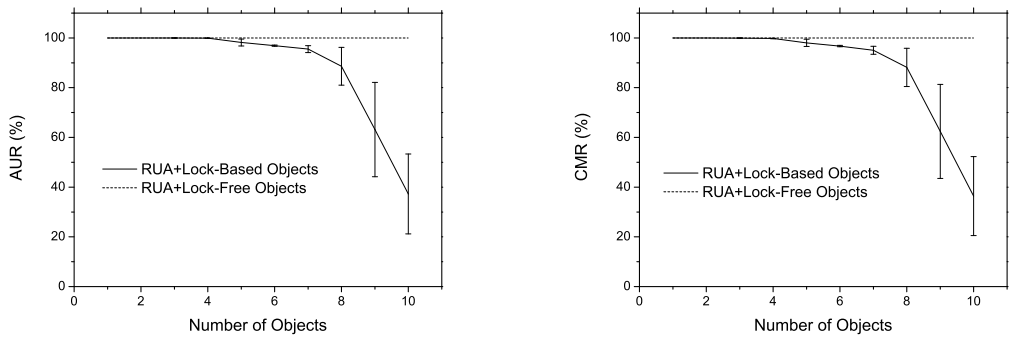
We consider a task set of 10 tasks, accessing 10 shared queues, and measure the CML of lock-free, lock-based, and ideal RUA under increasing average job execution times. Figure 9 shows the results. We observe that lock-free RUA yields almost the same CML as that of ideal RUA, as it exploits the eliminated 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 8.

6.2 Accrued Utility, Critical Time-Meets

We now measure the AUR and the *critical-time-meet ratio* (or 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, arbitrarily. 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: (1) a homogeneous class including just step shapes and (2) a heterogenous class including step, parabolic, and linearly-decreasing shapes.

Figures 10 and 11 show the AUR and CMR of lock-based and lock-free RUA for step TUFs and heterogenous TUFs, respectively, during underloads (AL = ≈0.4), under increasing number of shared objects. Figures 12 and 13 show similar results during overloads (AL = ≈1.1). [9]

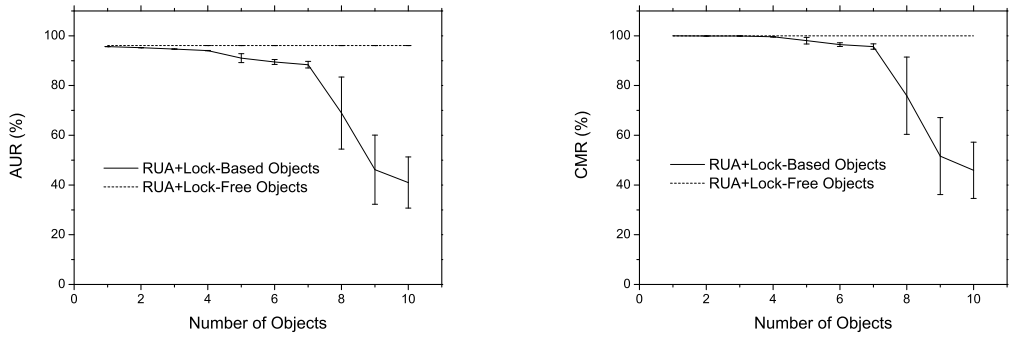


(a) AUR (b) CMR

Figure 10: AUR/CMR During Underload, Step TUFs

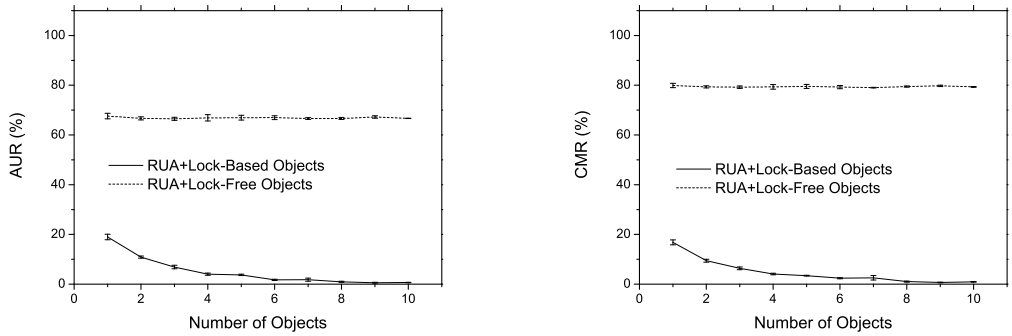
As expected, the figures show that lock-based RUA's AUR and CMR sharply decreases, eventually reaching 0% during overloads, as the number of objects increases. This is because, as the number of objects increases, greater number of task blockings' occurs, due to the large r, resulting in increased sojourn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

[9] In all figures in this section, the error bar around each data point represents 95% confidence interval of that data point.



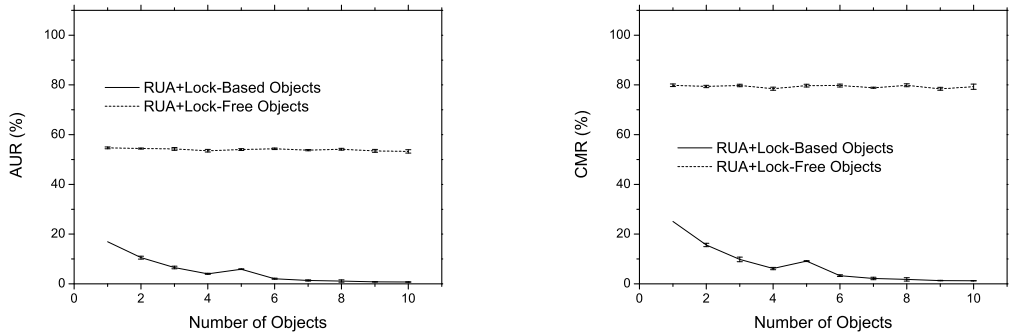
(a) AUR (b) CMR

Figure 11: AUR/CMR During Underload, Heterogeneous TUFs



(a) AUR (b) CMR

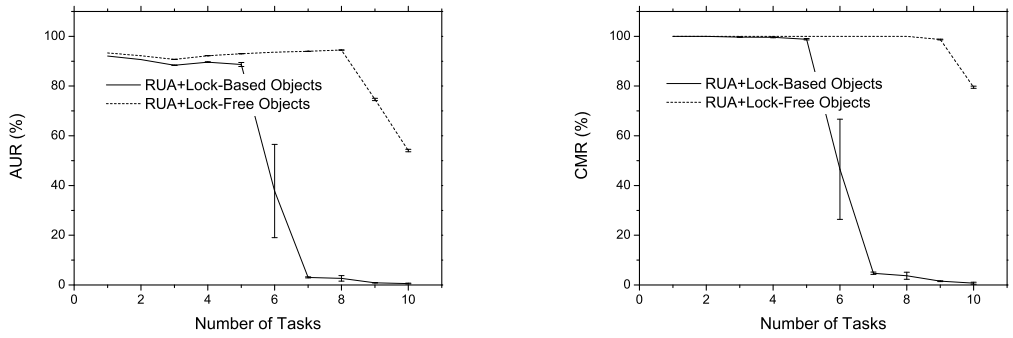
Figure 12: AUR/CMR During Overload, Step TUFs



(a) AUR (b) CMR

Figure 13: 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 underloads, lock-free RUA achieves almost 100% AUR and CMR, whereas during overloads, the algorithm achieves higher AUR by as much as ≈65% and higher CMR, by as much as ≈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.



(a) AUR (b) CMR

Figure 14: AUR/CMR During Increasing Readers, Heterogeneous TUFs

We repeated similar experiments as in Figures 10–13 for increasing number of reader tasks (instead of increasing shared objects) and observed exact similar trends and consistent results. Figure 14 shows a snapshot of these results (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.

7 Conclusions

In this paper, we consider non-blocking synchronization for dynamic 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 lower and upper bound on their total utilities — 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. In addition, the nested sharing that allows tasks to hold multiple objects is a subject for future research.

Acknowledgement

This work was supported by the Office of Naval Research under Grant N00014-05-1-0179 and The MITRE Corporation under Grant 52917. A preliminary version of this paper appeared as "Lock-Free Synchronization for Dynamic Embedded Real-Time Systems," H. Cho, B. Ravindran, and E. D. Jensen, *ACM Design, Automation, and Test in Europe (DATE), Real-Time Systems Track*, pages 438-443, March 2006.

References

[1] J. Anderson, V. Bud, and U. C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, pages 199–208, July 2005.

[2] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *IEEE Real-Time Systems Symposium (IEEE RTSS)*, pages 346–355, December 1998.

[3] J. H. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *IEEE Real-Time Systems Symposium (IEEE RTSS)*, pages 111 – 122, December 1997.

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

[5] J. Carpentar, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In J. Y. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, pages 30–1 — 30–19. Chapman and Hall/CRC, Boca Raton, Florida, USA, 2004.

[6] J. Chen and A. Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *10th Euromicro Workshop on Real-Time Systems*, pages 2–9, June 1998.

[7] H. Cho, B. Ravindran, and E. D. Jensen. A space-optimal, wait-free real-time synchronization protocol. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, pages 79 – 88, July 2005.

[8] R. Clark, E. D. Jensen, A. Kanevsky, and J. Maurer. An adaptive, distributed airborne tracking system. In *IEEE Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, volume 1586 of *LNCS*, pages 353–362. Springer-Verlag, April 1999.

[9] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990.

[10] R. K. Clark, E. D. Jensen, and N. F. Rouquette. Software organization to facilitate dynamic processor scheduling. In *IEEE Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, page p. 122b, April 2004.

[11] U. C. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global edf scheduling on multiprocessors. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, pages 75–84, July 2006.

[12] J.-F. Hermant and G. L. Lann. A protocol and correctness proofs for real-time high-performance broadcast networks. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 360–369, 1998.

[13] P. Holman and J. H. Anderson. Object sharing in pfair-scheduled multiprocessor systems. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, pages 111–120, June 2002.

[14] 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.

[15] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *IEEE Real-Time Systems Symposium (IEEE RTSS)*, pages 112–122, December 1985.

[16] H. Kopetz and J. Reisinger. The non-blocking write protocol nbw: A solution to a real-time synchronisation problem. In *IEEE Real-Time Systems Symposium (IEEE RTSS)*, pages 131–137, December 1993.

[17] P. Li and B. Ravindran. Fast, best effort real-time scheduling algorithms. *IEEE Transactions on Computers*, 53(9):1159 – 1175, 2004.

[18] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Transactions on Software Engineering*, 30(9):613 – 629, September 2004.

[19] D. P. Maynard, S. E. Shipman, et al. An example real-time command, control, and battle management application for alpha. Technical report, CMU CS Dept., Dec. 1988. Archons Project TR 88121.

[20] M.Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[21] 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 (JPDC)*, 51(1):1–26, May 1998.

[22] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 55 – 60, May 2005.

[23] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[24] N. R. Soparkar, H. F. Korth, and A. Silberschatz. *Time-Constrained Transaction Management*. Kluwer Academic Publishers, 1996.

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

[26] J. D. Valois. Lock-free linked list using compare-and-swap. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 214–222, 1995.

[27] H. Wu, B. Ravindran, E. D. Jensen, and U. Balli. Utility accrual scheduling under arbitrary time/utility functions and multiunit resource constraints. In *IEEE Real-Time and Embedded Computing Systems and Applications (RTCSA)*, pages 80–98, August 2004.