

# Utility Accrual Scheduling under Arbitrary Time/Utility Functions and Multi-unit Resource Constraints

Haisang Wu\*, Binoy Ravindran\*, E. Douglas Jensen†, and Umut Balli\*

\*ECE Dept., Virginia Tech

Blacksburg, VA 24061, USA

{hswu02,binoy,uballi}@vt.edu

†The MITRE Corporation

Bedford, MA 01730, USA

jensen@mitre.org

## Abstract

*We present a uni-processor real-time scheduling algorithm called Resource-constrained Utility Accrual algorithm (or RUA). RUA considers an application model, where activities can be subject to arbitrarily-shaped time/utility function (TUF) time constraints and resource constraints including mutual exclusion under a multi-unit resource request model. For such a model, we consider the scheduling objective of maximizing the total utility accrued by all activities. This problem was previously open. Since the problem is  $\mathcal{NP}$ -hard, RUA heuristically computes schedules with a polynomial-time cost. We analytically establish several timeliness and non-timeliness properties of the algorithm, including upper bound on blocking time (under multi-unit request model) and deadlock-freedom. We also implement RUA on a POSIX RTOS and conduct experimental comparisons with other TUF scheduling algorithms that address a subset of RUA's model. Our implementation measurements show that RUA performs generally better than, or as good as, other TUF algorithms for the applicable cases.*

## 1 Introduction

Thread scheduling in real-time embedded systems is fundamentally concerned with satisfying application time constraints. Hard real-time systems are subject to absolute timing require-

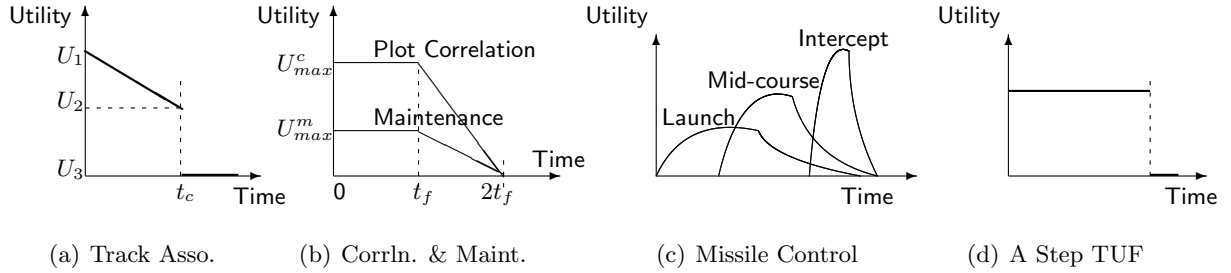


Figure 1: Example Time Constraints Specified Using Time/Utility Functions

ments, which are often expressed in terms of deadlines. With deadline time constraints, one can specify the hard timeliness optimality criterion of “always meet all deadlines” and use hard real-time scheduling algorithms [16] to achieve the criterion.

In this paper, we focus on dynamic, adaptive, embedded real-time control systems at any level(s) of an enterprise—e.g., devices in the defense domain such as multi-mode phased array radars [7] and battle management [6]. Such embedded systems include “soft” time constraints (besides hard) 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. Moreover, they often desire a soft timeliness optimality criterion such as completing all time-constrained activities as close as possible to their *optimal* completion times—so as to yield maximal collective utility—is the objective.

Jensen’s time/utility functions [11] (or TUFs) allow the semantics of soft time constraints to be precisely specified. A TUF, which is a generalization of the deadline constraint, specifies the utility to the system resulting from the completion of an activity as a function of its completion time. Figure 1 shows examples of time constraints specified using TUFs. Figures 1(a), 1(b), and 1(c) show time constraints of two large-scale, dynamic, embedded real-time applications specified using TUFs. The applications include: (1) the AWACS (Airborne Warning and Control System) surveillance mode tracker system [3] built by The MITRE Corporation and The Open Group (TOG); and (2) a coastal air defense system [18] built by General Dynamics (GD) and Carnegie Mellon University (CMU).

Figure 1(a) shows the TUF of the *track association* activity of the AWACS; Figures 1(b) and 1(c) show TUFs of three activities of the coastal air defense system called *plot correlation*, *track maintenance*, and *missile control*. Note that Figure 1(c) shows how the TUF of the missile control activity dynamically changes as the guided interceptor missile approaches its target.

The classical deadline constraint is a binary-valued downward “step” shaped TUF. This is shown in Figure 1(d).

When timing constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued utility from those activities—e.g., maximizing the sum, or the expected sum, of the activities’ attained utilities. Such criteria are called *Utility Accrual* (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that consider UA criteria are called UA sequencing algorithms. In general, other factors may also be included in the optimality criteria, such as resource dependencies and precedence constraints. Several UA scheduling algorithms are presented in the literature [2, 4, 12, 13, 17, 22].

To the best of our knowledge, GUS [13] and LBESA [17] are the only UA algorithms that consider arbitrarily-shaped TUFs. Further, only GUS [13] and DASA [4] consider mutual exclusion constraints, but under the single-unit resource request model. While GUS allows arbitrarily-shaped TUFs, DASA is restricted to step-shaped TUFs. However, UA scheduling under arbitrarily-shaped TUFs and mutual exclusion constraints under the multi-unit resource request model has not been studied. Multi-unit resource models are very important for many emerging real-time embedded systems, as many such systems use multi-unit (shared) resources and simultaneously need multiple units of a given resource for application progress [10].

In this paper, we precisely study this problem. We consider application activities that are subject to arbitrarily-shaped TUF time constraints and resource constraints including mutual exclusion under a multi-unit resource request model. For such an application model, we consider the scheduling objective of maximizing the total utility accrued by all activities. This problem is  $\mathcal{NP}$ -hard. We present a polynomial-time heuristic algorithm for the problem, called the Resource-contrainted Utility Accrual algorithm (or RUA). Further, we prove several timeliness and non-timeliness properties of the algorithm including upper bound on blocking time (under multi-unit resource request model), timeliness optimality during under-loads, deadlock-freedom, and correctness. We also implement RUA on a POSIX real-time operating system (RTOS) to verify its effectiveness. Our experimental measurements indicate that RUA performs very close to, if not better than, existing UA algorithms (that only consider a subset of RUA’s application model) for the cases that they apply.

Thus, the contribution of the paper is the RUA algorithm. To the best of our knowledge, we are not aware of any other efforts that solve the problem solved by RUA.

The rest of the paper is organized as follows: In Section 2, we outline our activity and utility models, and state the UA scheduling criterion. We present RUA in Section 3 and establish the algorithm’s timeliness and non-timeliness properties in Section 4. The experimental measurements are reported in Section 5. Finally, we conclude the paper and identify future work in Section 6.

## 2 Models and Objectives

### 2.1 Threads and Scheduling Segments

The basic scheduling entity that we consider is the thread abstraction. Thus, the application is assumed to consist of a set of threads, denoted as  $T_i, i \in \{1, 2, \dots, n\}$ . Threads can arrive arbitrarily and can be preempted arbitrarily.

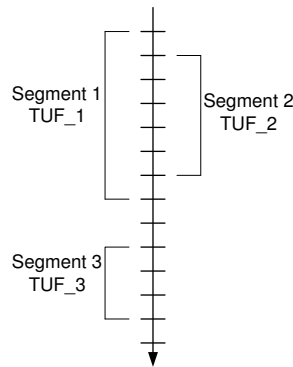


Figure 2: Example Scheduling Segments

A thread can be subject to time constraints. Following [9], a time constraint usually has a “scope”—a segment of the thread control flow that is associated with a time constraint. We call such a scope a “scheduling segment.” As in [9], we call a thread a “real-time thread” while it is executing inside a scheduling segment. Otherwise, it is called a “non-real-time thread.” TUF scheduling is general enough to schedule non-real-time and real-time threads in a consistent manner: time constraint of a non-real-time thread is modelled as a constant TUF whose utility represents its relative importance.

Scheduling segments can be nested or disjoint [9, 13] (see Figure 2 for an example). Thus, a thread can execute inside multiple scheduling segments. In such cases, it is governed by the tightest of the nested time constraints, which is often application-specific (e.g., earliest deadline

for downward step TUFs).

## 2.2 Resource Model

Threads can access non-CPU resources, which in general, are reusable. Examples include physical ones (e.g., disks) and logical ones (e.g., critical code sections guarded by mutexes). Resources can be shared, but are subject to mutual exclusion constraints. Names of the forms  $R$  and  $R_i$  always denote resources.

We consider the generic, multi-unit resource model described in [1]. For each multi-unit resource  $R$ , there is a fixed total number of units in the system,  $N_R$ . At any instant, a thread cannot request more than  $N_R$  units of a multi-unit resource. A nonnegative integer  $A_R$  denotes the number of available units of the resource  $R$ . The single-unit resource model in [4, 13, 20] is a special case of the multi-unit resource model with  $N_R = 1$ . A reader/writer lock is another example, for which  $N_R$  can exceed or be equal to the number of threads that may request  $R$ .

A thread may request multiple units of shared resources during its lifetime. The requested time intervals for holding resources may be nested or disjoint (similar to scheduling segments), but a resource request (from a thread) can only request one or more units of one resource at a time. Formally, a request operation is a triple  $(T, R, r)$ , which means the thread  $T$  is requesting  $r$  units of resource  $R$ .

## 2.3 Precedence Constraints

Threads can also have precedence constraints. For example, a thread  $T_i$  can become eligible for execution only after a thread  $T_j$  has completed, because  $T_i$  may require  $T_j$ 's results. As in [4, 13], such precedences can be programmed as resource dependencies.

## 2.4 Timeliness Model

A thread's time constraints are specified using TUFs. A TUF is always associated with a thread scheduling segment. The TUF associated with the scheduling segment of a thread  $T_i$  is denoted as  $U_i(\cdot)$ ; thus completion of  $T_i$ 's associated scheduling segment at a time  $t$  will yield a utility  $U_i(t)$ .

TUFs can be classified into unimodal and multimodal functions. Unimodal TUFs are those for which any decrease in utility cannot be followed by an increase. Examples are shown in

Figure 1. TUFs which are not unimodal are multimodal. We allow TUFs to take arbitrary shapes to facilitate maximum flexibility in specifying time constraints.

Each TUF  $U_i, i \in \{1, \dots, n\}$  has an initial time  $T_i.I$  and a termination time  $T_i.X$ . Initial and termination times are the earliest and the latest times for which the TUF is defined, respectively. If  $T_i.X$  is reached and execution of the corresponding scheduling segment has not been completed, an exception is raised. Normally, this exception will cause  $T_i$ 's abortion and execution of exception handlers, which may have time constraints themselves. We specify such time constraints using TUFs. Thus, handlers are scheduled similar to normal threads.

## 2.5 Utility Accrual Scheduling Criterion

Given the models previously described, several UA scheduling optimality criteria can be devised. In this paper, we consider the criterion of maximizing the sum of utility  $U$ . This problem is  $\mathcal{NP}$ -hard because it subsumes the problem of scheduling dependent threads with step-shaped TUFs, which has been shown to be  $\mathcal{NP}$ -hard in [4].

## 3 RUA Scheduling Algorithm

### 3.1 Algorithm Rationale

The potential utility that can be accrued by executing a thread defines a measure of its “return on investment.” Because of the unpredictability of future events, scheduling events that may happen later such as new thread arrivals and new resource requests cannot be considered at the time when the scheduler is invoked. Thus, a reasonable heuristic is the “greedy” strategy, which means selecting as many “high return” threads and their dependencies into the schedule as early as possible. This will increase the likelihood of maximizing the aggregate utility.

The metric used by RUA to determine the return on investment for a thread is called the *Potential Utility Density* (or PUD), which was originally developed in [4]. The PUD of a thread measures the amount of utility that can be accrued per unit time by executing the thread and the thread(s) that it depends upon.

To compute  $T_i$ 's PUD at time  $t$ , RUA considers  $T_i$ 's expected completion time (denoted as  $t_f$ ) and the expected utility by executing  $T_i$  and its dependent threads. For each thread  $T_j$  that is in  $T_i$ 's dependency chain and needs to be completed before executing  $T_i$ ,  $T_j$ 's expected completion

time is denoted as  $t_j$ . PUD of  $T_i$  is then computed as:  $\frac{U_i(t_f) + \sum_{T_j \in T_i.Dep} U_j(t_j)}{t_f - t}$ .

### 3.2 Algorithm Overview

The scheduling events of RUA include the arrival of a thread, the completion of a thread, a resource request, a resource release, and the expiration of a time constraint such as the arrival of the termination time of a TUF. To describe RUA, we define the following variables and auxiliary functions:

- $\mathcal{T}_r$  is the current set of unscheduled threads;  $\sigma$  is the ordered schedule.  $T_i \in \mathcal{T}_r$  is a thread. We use  $\sigma(i)$  to denote the thread occupying the  $i^{th}$  position in the schedule  $\sigma$ .
- $T_i.X$  is  $T_i$ 's termination time;  $T_i.ExecTime$  is  $T_i$ 's (known or expected) remaining execution time, and  $T_i.Dep$  is  $T_i$ 's dependency list.
- Function  $Owner(R)$  denotes the threads that are currently holding units of resource  $R$ ;  $OwnedUnit(T, R)$  denotes the number of units of  $R$  held by  $T$ .  $reqResource(T)$  returns the resource requested by  $T$ ;  $reqUnit(T, R)$  returns the number of units of  $R$  requested by  $T$ .
- $headOf(\sigma)$  returns the first thread in  $\sigma$ ;  $sizeOf(\sigma)$  returns the length of  $\sigma$ .
- $sortByPUD(\sigma)$  returns a schedule by non-increasing PUD. If two or more threads have the same PUDs, then the thread(s) with the largest  $ExecTime$  should appear before any others with the same PUDs.
- $Insert(T, \sigma, I)$  inserts  $T$  in the ordered list  $\sigma$  at the position indicated by index  $I$ ; if there are already entries in  $\sigma$  with the index  $I$ ,  $T$  is inserted before them. After insertion, the index of  $T$  in  $\sigma$  is  $I$ .
- $Remove(T, \sigma, I)$  removes  $T$  from ordered list  $\sigma$  at the position indicated by index  $I$ ; if  $T$  is not present at the position in  $\sigma$ , the function takes no action.
- $lookup(T, \sigma)$  returns the index value associated with the first occurrence of  $T$  in the ordered list  $\sigma$ .
- $feasible(\sigma)$  returns a boolean value indicating schedule  $\sigma$ 's feasibility. For a schedule  $\sigma$  to be feasible, the predicted completion time of each thread in  $\sigma$  must never exceed its termination time.

A description of RUA at a high level of abstraction is shown in Algorithm 1. When RUA is invoked at time  $t_{cur}$ , the algorithm first checks the feasibility of the threads. If the earliest predicted completion time of a thread is later than its termination time, it can be safely

```

1: input:  $\mathcal{T}_r$ ; output: selected thread  $T_{exe}$ ;
2: Initialization:  $t := t_{cur}$ ,  $\sigma := \emptyset$ ;
3: for  $\forall T_i \in \mathcal{T}_r$  do
4:   if  $feasible(T_i) = false$  then
5:     abort( $T_i$ );
   else
6:      $T_i.LUD = \frac{U_i(t+T_i.ExecTime)}{T_i.ExecTime}$ ;
7:      $T_i.Dep := buildDep(T_i)$ ;
8: for  $\forall T_i \in \mathcal{T}_r$  do
9:    $T_i.PUD := calculatePUD(T_i, t)$ ;
10:  $\sigma_{tmp} := sortByPUD(\mathcal{T}_r)$ ;
11: for  $\forall T_i \in \sigma_{tmp}$  from head to tail do
12:   if  $T_i.PUD > 0$  then
13:      $\sigma := insertByEDF(\sigma, T_i)$ ;
14:   else break;
15:  $T_{exe} := headOf(\sigma)$ ;
16: return  $T_{exe}$ ;

```

**Algorithm 1:** RUA: High Level Description

aborted (line 5). Otherwise, RUA calculates Local Utility Density (or LUD) of the thread by the definition given in line 6, and builds its dependency chain (line 7).

The PUD of each thread is computed by the procedure `calculatePUD()`, and the threads are then sorted by their PUDs (line 9 and 10). In each step of the *for*-loop from line 11 to 14, the thread with the largest PUD and its dependencies are inserted into  $\sigma$ , if it can produce a positive PUD. The output schedule  $\sigma$  is then sorted by the threads' termination times by the procedure `insertByEDF()`. Finally, RUA returns with the selected thread  $T_{exe}$ , which is at the head of  $\sigma$  (line 15–16).

### 3.3 Resource and Deadlock Handling

Before RUA computes thread partial schedules, the dependency chain of each thread must be determined. Algorithm 2 shows this procedure.

Algorithm 2 builds the General Resource Graph (GRG) for the operations on multi-unit resources. If the available units  $A_R$  of a resource  $R$  is less than the requested units of a thread  $T_i$ , then  $T_i$  is dependent on other threads that are holding units of resource  $R$  (line 4). RUA allocates such threads to be  $T_i$ 's predecessors, following the decreasing order of their LUDs, so as to increase the possible aggregate utility of the output schedule (line 8).

The procedure `buildDep()` traverses chains of resource request/ownership. In line 5, *for*-loop checks the current dependency list to insert the predecessors into the list  $T_i.Dep$ . The *for*-loop

```

1: input: Thread  $T_i$ ; output:  $T_i.Dep$  ;
2: Initialization:  $T_i.Dep(0) := T_i$ ;
3:  $reqResource(T_i) = R$ ;
4: if  $R \neq \emptyset \wedge reqUnit(T_i, R) > A_R$  then
5:   for  $n$  from 0 to  $sizeof(T_i.Dep) - 1$  do
6:      $j := n$ ;
7:     for  $\forall T \in owner(R)$  do
8:       Pick  $T_k$ :  $T_k \notin T_i.Dep$  and has the largest LUD;
9:       for  $m$  from 0 to  $sizeof(T_i.Dep) - 1$  do
10:        if  $T_i.Dep(m) = T_k$  then
11:          Remove( $T_i.Dep(m), T_i.Dep, m$ );
12:           $j := j - 1$ ;
13:           $n := j$ ;
14:          break;
15:         $j := j + 1$ ;
16:      Insert( $T_k, T_i.Dep, j$ );

```

**Algorithm 2:** buildDep( $T_i$ ): Build the Dependency List of Each Thread

starting from line 9 ensures that the predecessors are inserted in  $T_i.Dep$  at the positions right after their immediate successors. The procedure stops until each predecessor reaches one of the following two conditions: (1) it needs no resource, or (2) its requested units are available. Thus,  $T_i.Dep$  starts with  $T_i$  and ends with its farthest predecessor, and it should be executed from the tail to the head.

To handle deadlocks in the multi-unit resource model, we consider a deadlock detection and resolution strategy, instead of a deadlock prevention or avoidance strategy. Our rationale for this is that deadlock prevention or avoidance strategies normally pose extra requirements e.g., resources are always requested in ascending order of their identifiers.

Further, restricted resource access operations that can prevent or avoid deadlocks, as done in many resource access protocols, are not appropriate for the class of time-critical systems that we focus. For example, the Priority Ceiling Protocol [20] assumes that the highest priority of threads accessing a resource is known. Likewise, the Stack Resource Policy [1] assumes preemptive “levels” of threads *a priori*. Such assumptions are too restrictive for the class of systems that we focus (due to their dynamic nature).

With a multi-unit resource model, the presence of a cycle in the GRG is not always the sufficient condition for a deadlock, and under some request models, a knot in the GRG is sufficient [21]. But a cycle is always a necessary condition. Thus, we use a “strict” strategy to detect and resolve deadlocks: by a straightforward cycle-breaking algorithm.

The deadlock detection and resolution algorithm (Algorithm 3) is invoked by the scheduler

```

1: input: requesting thread  $T_i$ ;
2:  $Deadlock = \text{false}$ ;
3:  $\sigma_{thread}(0) = T_i, \sigma_{parent}(0) = T_i$ ;
4:  $\text{reqResource}(T_i) = R$ ;
5: if  $R \neq \emptyset \wedge \text{reqUnit}(T_i, R) > A_R$  then
6:   for  $n$  from 0 to  $\text{sizeOf}(\sigma_{thread}) - 1$  do
7:      $j := n$ ;
8:     for  $\forall T \in \text{owner}(R)$  do
9:       Pick  $T_k$  with the largest LUD;
10:      for  $m$  from 0 to  $\text{sizeOf}(\sigma_{thread}) - 1$  do
11:        if  $\sigma_{thread}(m) = T_k$  then
12:          for  $r$  from 0 to  $\text{sizeOf}(\sigma_{parent}) - 1$  do
13:            if  $\sigma_{parent}(r) = T_k$  then
14:               $Deadlock = \text{true}$ ;
15:              goto line 24;
16:             $\text{Remove}(\sigma_{thread}(m), \sigma_{thread}, m)$ ;
17:             $\text{Remove}(\sigma_{parent}(m), \sigma_{parent}, m)$ ;
18:             $j := j - 1$ ;
19:             $n := j$ ;
20:            break;
21:           $j := j + 1$ ;
22:           $\text{Insert}(T_k, \sigma_{thread}, j)$ ;
23:           $\text{Insert}(T_n, \sigma_{parent}, j)$ ;
24:   if  $Deadlock = \text{true}$  then
25:      $\text{abort}(T, \text{the thread with the lowest LUD in the cycle})$ ;

```

**Algorithm 3:** DLresolution( $T_i$ ): Deadlock Detection and Resolution

whenever a thread requests a resource. Initially, there is no deadlock in the system. By induction, it can be shown that a deadlock can occur if and only if the edge that arises in the GRG due to the new resource request lies on a cycle. Thus, it is sufficient to check if the new edge produces a cycle in the GRG.

In order to find a cycle in the GRG, the procedure DLresolution() traverses the GRG using two tentative schedules,  $\sigma_{thread}$  and  $\sigma_{parent}$ .  $\sigma_{thread}$  is used to build the dependency list with the new resource request;  $\sigma_{parent}$  is used to keep track of the immediate successors of the threads. If  $T_k$  is found both in  $\sigma_{thread}$  and  $\sigma_{parent}$  (line 11 and 13), a cycle in the GRG is detected.

To resolve the deadlock, some thread needs to be aborted. If a thread  $T$  were to be aborted, then its utility is lost. To minimize such loss of utility, we compute the utility that the thread can potentially accrue by itself if it were to continue its execution, which is measured by its LUD. RUA aborts the thread with the minimal LUD in the cycle to resolve a deadlock.

Timeliness of the system can be improved if we preempt  $T$  instead of abort it, given that  $T$  can complete before its termination time. We can roll-back and add the thread into the unordered

schedule at the next scheduling event. To roll-back  $T$ , at each resource request, a checkpoint should be saved for it, since resource requests are the events causing deadlocks. This can be a future improvement of RUA.

### 3.4 Manipulating Partial Schedules

The `calculatePUD()` algorithm (Algorithm 4) accepts a thread  $T_i$  (with its dependency list) and the current time  $t_{cur}$ . On completion, the algorithm determines PUD for  $T_i$ , by assuming that threads in  $T_i.Dep$  are executed from the current position (at time  $t_{cur}$ ) in the schedule, while following the dependencies.

```

1: input:  $T_i, t_{cur}$ ; output:  $T_i.PUD$ ;
2: Initialization :  $t_c := 0, U := 0$ ;
3: for  $\forall T_j \in T_i.Dep$ , from tail to head do
4:   |  $t_c := t_c + T_j.ExecTime$ ;
5:   |  $U := U + U_j(t_{cur} + t_c)$ ;
6:  $T_i.PUD := U/t_c$ ;
7: return  $T_i.PUD$ ;

```

**Algorithm 4:** `calculatePUD()`: Calculate PUD for Each Thread

To compute  $T_i$ 's PUD at time  $t_{cur}$ , RUA considers each thread  $T_j$  that is in  $T_i$ 's dependency chain, which needs to be completed before executing  $T_i$ . The total expected execution time upon completing  $T_j$  is counted using the variable  $t_c$  of line 4. With the known expected completion time of each thread, we can derive the expected utility for each thread, and thus get the total accrued utility  $U$  (line 5) to calculate  $T_i$ 's PUD.

The total execution time of the thread  $T_i$  and its dependent threads consists of two parts: (1) the time needed to execute the threads holding the resources that are needed to execute  $T_i$ ; and (2) the remaining execution time of  $T_i$  itself. According to the process of `buildDep()`, all the relative threads are included in  $T_i.Dep$ .

Note that we are calculating each thread's PUD assuming that the threads are executed at the current position in the schedule. This would not be true in the output schedule  $\sigma$ , and thus affects the accuracy of PUDs calculated. Actually, we are calculating the highest possible PUD of each thread by assuming that it is executed at the current position. Intuitively, this would benefit the final PUD, since `insertByEDF()` always takes the thread with the highest PUD at each insertion on  $\sigma$ . Also, the PUD calculated for the scheduled thread, which is at the head of the feasible schedule, is always accurate.

```

1: input   :  $T_i$  and an ordered thread list  $\sigma$ 
2: output  : the updated list  $\sigma$ 
3: if  $T_i \notin \sigma$  then
4:   copy  $\sigma$  into  $\sigma_{tmp}$ :  $\sigma_{tmp} := \sigma$ ;
5:   Insert( $T_i, \sigma_{tmp}, T_i.X$ );
6:    $CuTT = T_i.X$ ;
7:   for  $\forall T_j \in \{T_i.Dep - T_i\}$  from head to tail do
8:     if  $T_j \in \sigma_{tmp}$  then
9:        $TT = \text{lookup}(T_j, \sigma_{tmp})$ ;
10:      if  $TT < CuTT$  then continue;
11:      else Remove( $T_j, \sigma_{tmp}, TT$ );
12:       $CuTT := \min(CuTT, T_j.X)$ ;
13:      Insert( $T_j, \sigma_{tmp}, CuTT$ );
14:   if feasible( $\sigma_{tmp}$ ) then
15:      $\sigma := \sigma_{tmp}$ ;
16: return  $\sigma$ ;

```

**Algorithm 5:** insertByEDF(): Keep a Termination Time Ordered and Correct Queue

The details of `insertByEDF()` in line 13 of Algorithm 1 is shown in Algorithm 5. `insertByEDF()` updates the tentative schedule  $\sigma$  by attempting to insert each thread, along with all of its dependencies, to  $\sigma$ . The updated schedule  $\sigma$  is an ordered list of threads, where each thread is placed according to the termination time it should meet.

Note that the time constraint that a thread should meet is not necessarily the thread termination time. In fact, the index value of each thread in  $\sigma$  is the actual time constraint that the thread should meet.

A thread may need to meet an earlier termination time in order to enable another thread to meet its time constraint. Whenever a thread is considered for insertion in  $\sigma$ , it is scheduled to meet its own termination time. However, all of the threads in its dependency list must execute before it can execute, and therefore, must precede it in the schedule. The index values of the dependencies could be changed with `Insert()` in line 13 of Algorithm 5.

The variable  $CuTT$  is used to keep track of this information. Initially, it is set to be the termination time of thread  $T$ , which is tentatively added to the schedule (line 6, Algorithm 5). Thereafter, any thread in  $T.Dep$  with a later time constraint than  $CuTT$  is required to meet  $CuTT$ . If, however, a thread has a tighter termination time than  $CuTT$ , then it is scheduled to meet the tighter termination time, and  $CuTT$  is advanced to that time since all threads left in  $T.Dep$  must complete by then (lines 12–13, Algorithm 5). Finally, if this insertion produces a feasible schedule, then the threads are included in this schedule; otherwise, not (lines 14–15).

### 3.5 Computational Complexity

To analyze the complexity of RUA (Algorithm 1), we consider  $n$  threads, a maximum of  $m$  resources, and a maximum of  $N_R$  units for each resource. In the worst case, `buildDep()` may build a dependency list with a length  $n$ ; so the *for*-loop from line 3 to 7 requires  $O(n^2)$  time. Also, the *for*-loop containing `calculatePUD()` (line 8–9) can be repeated  $O(n^2)$  times in the worst case. The complexity of procedure `sortByPUD()` is  $O(n \log n)$ .

Complexity of the *for*-loop body starting from line 11 is dominated by `insertByEDF()` (Algorithm 5). Its complexity is dominated by the *for*-loop (line 7–13, Algorithm 5), which requires  $O(n \log n)$  time since the loop will be executed no more than  $n$  times and each execution will require  $O(\log n)$  time to perform `Insert()`, `Remove()` and `lookup()` operations on the tentative schedule. Therefore, the worst-case complexity of the RUA algorithm is  $2 \times O(n^2) + O(n \log n) + n \times O(n \log n) = O(n^2 \log n)$ .

## 4 Properties of RUA

### 4.1 Non-Timeliness Properties

We now present RUA’s non-timeliness properties including deadlock-freedom, correctness, and mutual exclusion.

RUA respects resource dependencies by ensuring that the thread selected for execution can execute immediately. Thus, no thread is ever selected for normal execution if it is dependent on some others.

**Theorem 1** *RUA ensures deadlock-freedom.*

**Proof** A cycle in the GRG is the necessary condition for a deadlock in the multi-unit resource model. RUA does not allow such a cycle by deadlock detection and resolution; so it is deadlock free. □

**Lemma 2** *In `insertByEDF()`’s output, all the dependents of a thread must execute before it can execute, and therefore, must precede it in the schedule.*

**Proof** `insertByEDF()` seeks to maintain an output queue ordered by threads’ termination times, while respecting resource dependencies. Consider thread  $T_i$  and its dependent  $T_j$ . If  $T_j.X$  is earlier than  $T_i.X$ , then  $T_j$  will be inserted before  $T_i$  in the schedule. If  $T_j.X$  is later than

$T_i.X, T_j.X$  is advanced to be  $T_i.X$  by the operation with  $CuTT$ . According to the definition of  $\text{insert}()$ , after advancing the termination time,  $T_j$  will be inserted before  $T_i$ .  $\square$

**Theorem 3** *When a thread  $T_i$  that requests some or all units of a resource  $R$  is selected for execution by RUA,  $T_i$ 's requested number of units of  $R$  will be free. We call this RUA's correctness property.*

**Proof** From Lemma 2, the output schedule  $\sigma$  is correct. Thus, RUA is correct.  $\square$

Thus, if insufficient units of a resource are available for a thread  $T_i$ 's request, threads holding units of the resource will become  $T_i$ 's predecessors. We present RUA's mutual exclusion property by a corollary.

**Corollary 4** *RUA satisfies mutual exclusion constraints in resource operations.*

## 4.2 Timeliness Properties

With Corollary 4, for a single-unit resource model, when a thread needs to hold a resource, it must wait until no other thread is holding the resource. A thread waiting for an exclusive resource is said to be *blocked* on that resource. Otherwise, it can hold the resource and enter the piece of code executed under mutual exclusion constraints, which is called a *critical section*. We first derive the maximum blocking time that each thread may experience under RUA.

**Theorem 5** *Under RUA with the single-unit resource model, a thread  $T_i$  can be blocked for at most the duration of  $\min(n, m)$  critical sections, where  $n$  is the number of threads that could block  $T_i$  and have longer termination times than  $T_i$  has, and  $m$  is the number of resources that can be used to block  $T_i$ .*

**Proof** The operation of the procedure  $\text{insertByEDF}()$  conforms to the Priority Inheritance Protocol (or PIP) [20]. In Algorithm 5, any thread in  $T_i.Dep$  with a later time constraint than  $CuTT$  could block  $T_i$ , and it is required to meet  $CuTT$ , which is initially set to be  $T_i.X$  (line 6). If, however, a dependent thread has a tighter termination time than  $CuTT$ , then it is scheduled to meet the tighter termination time, and  $CuTT$  is advanced to that time since all threads left in  $T_i.Dep$  must complete by then. Note that in line 13, after insertion, the index of  $T_j$  is changed to  $CuTT$ . This is exactly the priority inheritance operation. Thus, the theorem immediately follows from properties of the PIP [20]  $\square$

With multi-unit resources, one resource may block a thread for more than one critical section, because the resource contains multiple units, and each unit may result in a blocking critical section. We present the maximum blocking time for the multi-unit resource model in Theorem 6.

**Theorem 6** *Under RUA with the multi-unit resource model, a thread  $T_i$  can be blocked for at most the duration of  $\frac{n(n-1)}{2}$  critical sections, where  $n$  is the number of threads that could block  $T_i$ , and have longer termination times than  $T_i$  has.*

**Proof** According to the output of `buildDep()` (Algorithm 2),  $T_i.Dep$  contains the  $n$  predecessors with later termination times than that of  $T_i$ . In the worst case,  $T_i$  requests all units of a resource, which are held by each of the  $n$  dependents. Thus, each of the  $n$  dependents can block  $T_i$  for one critical section. Furthermore, the same worst case can exist among the  $n$  dependents. However, a thread can only be blocked by its predecessors, and each predecessor can block it for only one critical section. Therefore,  $T_i$  can be blocked for at most  $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$  critical sections.  $\square$

We also consider other timeliness properties under no resource dependencies, where RUA can be compared with a number of well-known algorithms. Specifically, we consider: (1) a set of independent periodic activities, where each activity has a single computational thread with a step TUF (such as the one in Figure 1(d)); and (2) there are sufficient processor cycles for meeting all thread termination-times i.e., there is no overload.

**Theorem 7** *Under conditions (1) and (2), a schedule produced by EDF [8] is also produced by RUA, yielding equal utilities. Not coincidentally, this is simply a termination-time ordered schedule.*

**Proof** We prove this by examining Algorithms 1 and 5. For a thread  $T$  without dependencies,  $T.Dep$  only contains  $T$  itself. During non-overload situations,  $\sigma$  from line 13 of Algorithm 1 is termination time ordered.

The TUF termination time that we consider is analogous to a deadline in [8]. As proved in [8, 16], a deadline-ordered schedule is optimal (with respect to meeting all deadlines) when there are no overloads. Thus,  $\sigma$  yields the same total utility as EDF.  $\square$

Some important corollaries about RUA’s timeliness behavior during non-overload situations can be deduced from EDF’s optimality [5].

**Corollary 8** *Under conditions (1) and (2), RUA always meets all thread termination times.*

**Corollary 9** *Under conditions (1) and (2), RUA minimizes the possible maximum lateness.*

## 5 Experimental Evaluation

We implemented RUA, five other UA algorithms including GUS [13], DASA [4], LBESA [17], UPA [22], and  $D^{over}$  [12], and two non-UA algorithms including EDF [8] and fixed priority (or

FP). For FP, we assign thread priorities based on maximum utility of TUFs i.e., high utility implies high priority. The algorithms were implemented in our previously developed scheduling framework called *meta-scheduler* [14, 15]. The meta-scheduler is an application-level framework for implementing UA schedulers on POSIX RTOSes, without RTOS modifications. We use QNX Neutrino 6.2 RTOS [19] in our study.

We denote  $C_{avg}$  as the average thread execution time, and  $\rho$  as the average load. During each experiment, 100 threads are generated with randomly distributed parameters such as thread execution times and termination times. For instance, we chose  $C_{avg}$  to be exponentially distributed with a mean of 0.5 *sec*. Given a workload  $\rho$ , we calculate the mean inter-arrival time as  $C_{avg}/\rho$ . In addition, the laxity of a thread is uniformly distributed between 50 *msec* and 1 *sec*. Thus, the sum of the thread execution time and laxity is the relative thread termination time, after which we assume zero utility can be accrued by the thread.

Moreover, TUFs of the threads may be step, linear, parabolic, or combinations of these basic shapes. In our experiments, the maximal utility of threads are uniformly distributed between 10 and 500. We use uniform distributions to define the experimental resource request parameters, which typically include the number of resources requested by a thread and resource hold times.

## 5.1 Algorithm Effectiveness

Since all algorithms except RUA cannot handle the multi-unit resource model, we conducted experiments with the single-unit model to compare their performance. The experiments were conducted using two categories of TUFs, namely step TUFs and arbitrarily-shaped TUFs.

Our first set of experiments consider resource-independent threads with downward step TUFs. Note that all algorithms allow this model, thus allowing a comparative study with RUA. Figure 3 shows the accrued utility ratio (or AUR) and termination time meet rate (or XMR) of the algorithms as the average load increases. AUR is defined as the ratio of accrued aggregate utility to the maximum possible utility, and XMR is the ratio of the number of threads meeting their termination times to the total number of thread releases.

We observed that DASA, GUS, and RUA have very close performance for the entire load range; so we exclude DASA and GUS from Figure 3. As the figure shows, UPA has the best performance during light/medium load situations. However, during heavy loads, we observe that: (1) DASA, GUS, and RUA outperform UPA; (2)  $D^{over}$  exhibits the worst performance

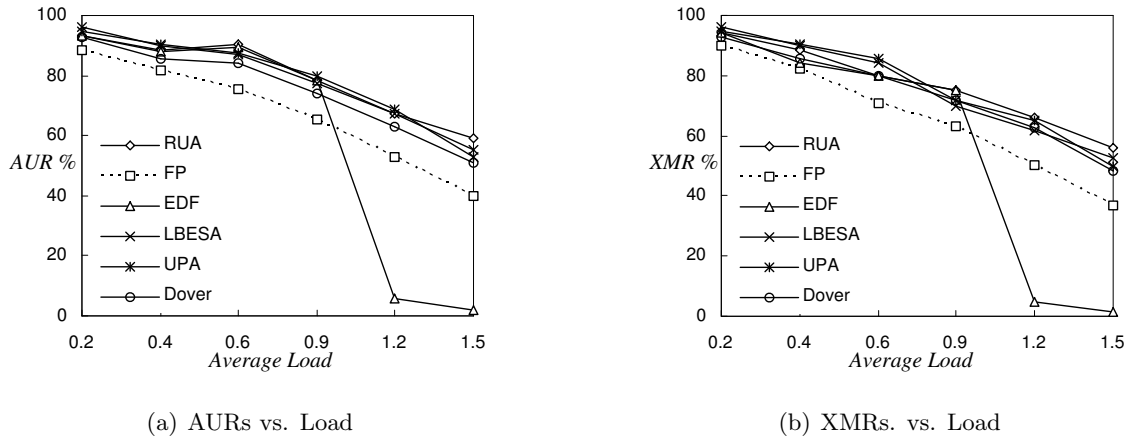


Figure 3: Algorithm Performance under Step TUFs and No Dependencies

among all UA algorithms; (3) FP performs worse than the UA algorithms; and (4) EDF suffers domino effects during overloads [17]. Thus, the results clearly illustrate that RUA offers superior adaptivity than traditional real-time scheduling algorithms.

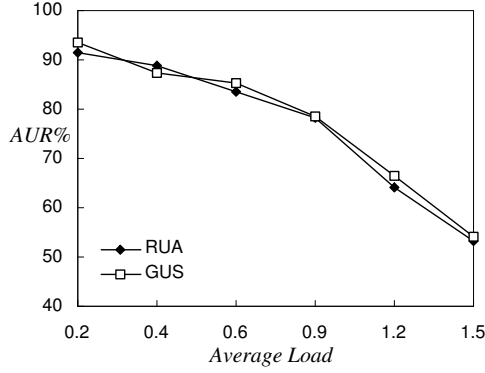
Figure 4 shows the algorithm performance for threads with arbitrarily-shaped TUFs and (single-unit) resource dependencies. The TUFs are represented by  $2^{rd}$ -order polynomials in the experiments, and coefficients of the polynomials are changed to generate basic shapes of TUFs that we described before. Observe that RUA performs very close to GUS during both underloads and overloads. The performance gap between the two algorithms is no more than 5% in terms of AUR and XMR during different loads.

GUS neither conducts a feasibility test, nor considers the termination time order for scheduling the feasible thread subset. RUA performs such operations, but doesn't consider execution mode, which is considered by GUS and DASA. With such pros and cons, RUA shows similar performance to GUS.

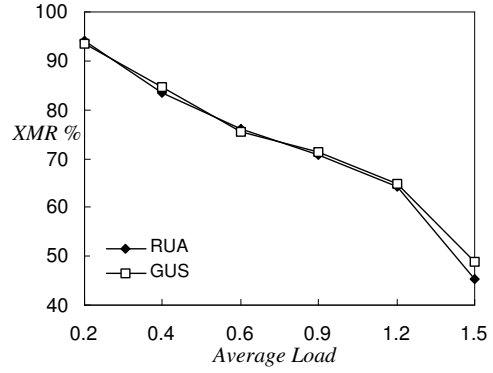
## 5.2 Algorithm Overhead

We also measured the algorithm overhead, which is the execution time of the scheduler function. Since overhead is specific to scheduling algorithms and the implementation, we are particularly interested in observing how the number of threads to be scheduled affects RUA's overhead for our un-optimized, prototype implementation.

To measure the overhead, we used Neutrino 6.2's `ClockCycles()` system call to directly access the hardware clock cycle counter. The hardware clock cycle counter increments its value by one



(a) AURs vs. Load of RUA and GUS



(b) XMRs vs. Load of RUA and GUS

Figure 4: Algorithm Performance under Arbitrary TUFs and Dependencies

at each CPU cycle. For the hardware used in our experiments, the CPU frequency is measured as 448.78 MHz. Thus, the accuracy of overhead measurement can be close to  $2 \text{ nsec}$ , which is measured by  $1/(448.78 \times 10^6)$ .

We measured RUA's overhead by conducting experiments on a total number of 200 aperiodic and randomly generated threads. Figure 5 shows RUA's overhead; we also include DASA's overhead for comparison. The figure shows the measured algorithms' overhead (mean values in  $\text{usec}$ ) under increasing number of threads in the ready queue, and its standard deviation (as vertical lines at the mean value points).

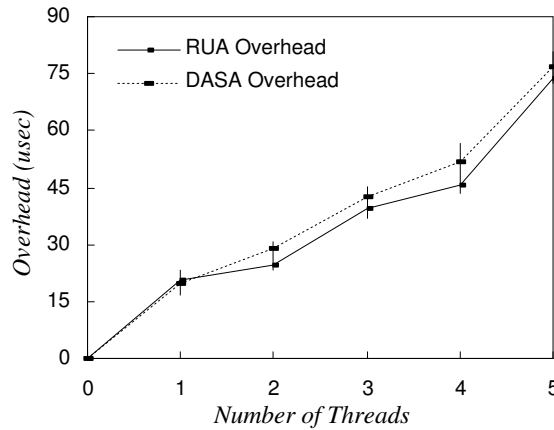


Figure 5: The Overhead of RUA and DASA

We observe that RUA's overhead is comparable to DASA's, even on our un-optimized implementation. For example, with 3 threads in the ready queue, the overhead costs of RUA and DASA are  $39.4 \text{ usec}$  and  $42.1 \text{ usec}$ , respectively. Further, the algorithm overhead only consti-

tutes a small fraction of the thread execution times. In our experiments, the average thread execution time is 0.5 sec; therefore, algorithm overhead of RUA is less than 0.01% of thread execution time for a ready queue with 3 threads.

Since RUA aborts infeasible threads at the beginning of each scheduling event (line 5, Algorithm 1), the longest ready queue that we could obtain has a length of only five. We plan to “fabricate” longer queues to measure the scheduler overhead in the future.

## 6 Conclusions and Future Work

The RUA scheduling algorithm presented in this paper solves a previously open problem: UA scheduling under arbitrarily-shaped TUF time constraints and multi-unit resource dependencies. We establish several timeliness and non-timeliness properties of RUA including maximum blocking time, timeliness optimality during under-loads, deadlock-freedom, and correctness. Our analytical and experimental results establish that RUA (1) yields the timeliness-optimality of traditional real-time scheduling algorithms; (2) outperforms traditional algorithms during overloads and thus has superior adaptability; and (3) has overhead comparable to current UA algorithms.

There are several interesting directions for future work. One direction is to augment the UA model presented here with energy constraints. Another direction is to consider stochastic UA criteria.

## Acknowledgements

This work was supported by the US Office of Naval Research under Grant N00014-00-1-0549, The MITRE Corporation under Grant 52917, and QNX Software Systems Ltd. through a software grant.

## References

- [1] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–99, March 1991.
- [2] K. Chen and P. Muhlethaler. A scheduling algorithm for tasks described by time value function. *Real-Time Systems*, 10(3):293–312, May 1996.
- [3] R. Clark, E. D. Jensen, and et al. An adaptive, distributed airborne tracking system. In *Proc. 7th WPDRTS*, volume 1586 of *LNCS*, pages 353–362. Springer-Verlag, April 1999.
- [4] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990. CMU-CS-90-155.

- [5] M. Dertouzos. Control robotics: the procedural control of physical processes. *Information Processing*, 74, 1974.
- [6] GlobalSecurity.org. Bmc3i battle management, command, control, communications and intelligence. <http://www.globalsecurity.org/space/systems/bmc3i.htm/>.
- [7] GlobalSecurity.org. Multi-platform radar technology insertion program. <http://www.globalsecurity.org/intell/systems/mp-rtip.htm/>.
- [8] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.
- [9] E. D. Jensen. Asynchronous decentralized real-time computer systems. In *Real-Time Computing*, NATO Advanced Study Institute. Springer Verlag, October 1992.
- [10] E. D. Jensen. A timeliness paradigm for mesosynchronous real-time systems. *Invited Talk*, IEEE RTAS, 2003. <http://www.real-time.org>.
- [11] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *IEEE RTSS*, pages 112–122, December 1985.
- [12] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for overloaded real-time systems. In *IEEE RTSS*, pages 290–299, December 1992.
- [13] P. Li. *A Utility Accrual Scheduling Algorithm for Resource-Constrained Real-Time Activities*. Phd dissertation proposal, Virginia Tech, 2003. <http://www.ee.vt.edu/~realtime/li-proposal03.pdf>.
- [14] P. Li, B. Ravindran, et al. Choir: A real-time middleware architecture supporting benefit-based proactive resource allocation. In *IEEE ISORC*, pages 292–299, May 2003.
- [15] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi. Utility accrual scheduling on off-the-shelf posix real-time operating systems: A formally verified scheduling framework. *IEEE Transactions on Software Engineering*. Submitted September 2003 (under review), Available at: <http://nile.ece.vt.edu/submissions/Meta-TSE03.zip>.
- [16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.
- [17] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986. CMU-CS-86-134.
- [18] D. P. Maynard, S. E. Shipman, et al. An example real-time command, control, and battle management application for alpha. Archons Project TR-88121, Carnegie Mellon University, Dec. 1988.
- [19] QNX. QNX Neutrino RTOS. [http://www.qnx.com/products/ps\\_neutrino/](http://www.qnx.com/products/ps_neutrino/).
- [20] 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.
- [21] M. Singhal and N. G. Shivaratri. *Advanced concepts in operating systems*. McGraw-Hill Inc., 1994.
- [22] J. Wang and B. Ravindran. Time-utility function-driven switched ethernet: Packet scheduling algorithm, implementation, and feasibility analysis. *IEEE TPDS*, 15(2):119–133, February 2004.