

Utility Accrual Scheduling Under Joint Utility and Resource Constraints*

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

*ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{hswu02, binoy}@vt.edu

†The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

Abstract

We extend time/utility functions and utility accrual model with the concept of joint utility functions (or JUFs) that allow an activity's utility to be described as a function of the completion times of other activities and their progress. We also specify the concept of progressive utility that generalizes the previously studied imprecise computational model, by describing an activity's utility as a function of its progress. Given such an extended utility accrual model, we consider the scheduling criterion of maximizing the weighted sum of completion time, progressive, and joint utilities. We present an algorithm called the Combined Utility Accrual algorithm (or CUA) for this criterion. Experimental measurements with an implementation of CUA on a POSIX RTOS illustrate the effectiveness of JUFs in a class of applications of interest to us.

1. Introduction

Next-generation real-time, embedded systems that are emerging in many domains such as military, space, and telecommunications often have multiple, dynamic, concurrent requests for services. The quality of service (QoS) to the mission, application, or system is often *multi-dimensional*.

The quality of the provided service often depends on *when* the service is performed. In many situations, the service quality *varies* with the time at which the service is provided. For example, the quality of a missile launch service that launches an interceptor missile to engage a hostile target depends upon when the missile is actually launched—if launched too early or too late, the interceptor will probably miss the target.

In some situations, the quality of the provided service depends on the *accuracy* of the computational results that are embodied in the service. For example, the quality of a radar-based tracking service depends on how accurately the objects in the sky can be tracked.

Furthermore, in some situations, the quality of a provided service depends upon when *other* services are completed and the accuracy of the computational results that are embodied in those services. For example, the quality of a missile control service that provides guidance updates to an in-flight interceptor (to compensate for changes in the course and speed of the target and navigational inaccuracies) depends upon the completion times of tracking services that produce course and speed estimates of the interceptor and the target. It also depends on the accuracy of the estimates. If the tracking services produce estimates too late, even though the estimates are highly accurate, the estimates will probably be too stale to be useful for interceptor guidance update calculations, because both the target and the interceptor are continuously moving—and the former's movement is not necessarily highly predictable. However, if the estimates are produced too early, but inaccurate, the inaccuracy (of the estimates) can cause guidance update calculations to be ineffective.

Service requests in many emerging dynamic real-time systems often cause resource contention, as there are many resources that are shared. Example shared resources include: physical ones such as processors, storage devices, communication devices and channels, power; logical ones such as locks; and application-level devices such as sensors and actuators. Resolution of the resource contention directly affects the quality of provided services to the users, mission, and application, and hence the system's behavior and degree of mission success. Thus, contention resolution policies must be mission-oriented, application/situation-specific, and dynamic and adaptive.

Jensen's time/utility functions [5] (or TUFs) allow the semantics of soft time constraints to be precisely specified. A TUF specifies the utility to the system that results from the completion of an activity as a function of the activity's completion time.

Figures 1(a), 1(b), 1(c), and 1(d) show time constraints of two significant, embedded real-time systems specified using TUFs. The systems include: (1) AWACS surveillance tracker [2] built by The MITRE Corporation and The Open Group (TOG); and (2) coastal air defense system [11] built by General Dy-

* 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.

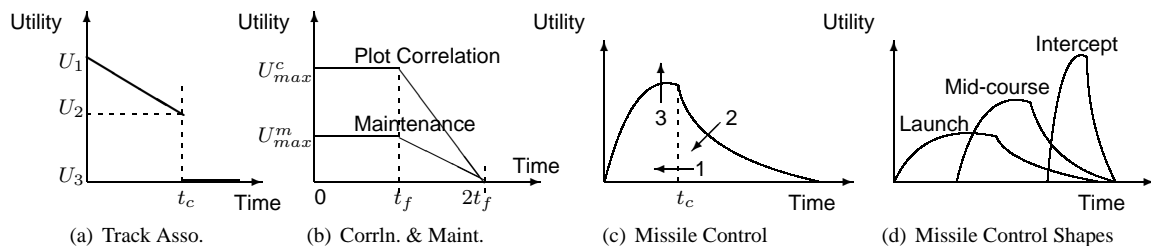


Figure 1. TUFs of (a) MITRE/TOG AWACS tracker and (b,c,d) GD/CMU coastal air defense

namics (GD) and Carnegie Mellon University (CMU). Figure 1(a) shows the TUF of the AWACS *track association* activity. While Figure 1(b) shows TUFs of the air-defense *plot correlation* and *track maintenance* activities, Figure 1(c) shows the TUF of the air-defense *missile control* activity. Figure 1(d) shows how the TUF of the missile control activity dynamically changes. Note that the classical deadline constraint is a “step” shaped TUF.

When timing constraints are expressed with TUFs, the scheduling optimality criteria are based on factors that are in terms of maximizing accrued utility from those activities—for example, 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 [1, 3, 6, 7, 10, 12].

In this paper, we extend the TUF/UA model with the concept of *joint utility functions* (or JUFs) that allow an activity’s utility to be specified in terms of the completion times of other activities and their progress. We also describe the concept of *progressive utility functions* (or PUFs) that allow an activity’s utility to be described as a function of its progress—e.g., the computational accuracy of the activity’s results. PUFs are a generalization of the previously studied *imprecise computational* model [9].

For such an extended utility accrual model, we consider the UA scheduling criterion of maximizing the weighted sum of completion time, progressive, and joint utilities. We present an algorithm called the Combined Utility Accrual algorithm (or CUA) for this criterion, and implement it on a POSIX RTOS to verify the effectiveness of JUFs in an application domain—defense—of particular interest to us.

The rest of the paper is organized as follows. We discuss the motivation and rationale of JUFs in Section 2. In Section 3, we outline the activity and utility models and state the UA scheduling criterion. We present the CUA algorithm in Section 4 and report experimental evaluation in Section 5. Finally, we conclude the paper and identify future work in Section 6.

2. The Joint Utility Function

2.1. Motivation

In some situations, an activity can accrue utility that depends on when *other* activities complete and their progress such as computational accuracy. We call this notion of utility *joint utility*. To illustrate the concept, we describe a notional combat system application scenario derived from [4].

The scenario includes a air target engagement system that consists of a sensor platform (e.g., a sensor aircraft) and a weapons platform (e.g., a fighter aircraft). The sensor platform is responsible for tracking airborne objects, including hostile targets (that need to be engaged) and interceptor weapons that are launched to engage them. The weapons platform is responsible for launching interceptor weapons to engage hostile targets that are detected by the sensor platform.

The sensor platform contains *tracker* activities that track hostile targets and launched interceptor weapons by associating sensor reports to target and interceptor tracks. In this scenario, the sensor/tracker pair for targets is separate from that for interceptors, because initially the targets (e.g., cruise missiles, ballistic missiles) are too distant from the interceptors for a single air-to-air sector sensor/tracker pair to track both the targets and the interceptors with sufficient accuracy.

Further, the sensor platform contains *guidance* activities that monitor the progress of in-flight interceptors and provide course updates to the interceptors, in order to compensate for changes in the direction of targets and for navigational inaccuracies.

The sensor platform’s trackers produce location estimates of targets and interceptors. The location estimates are then used by the platform’s guidance activities to compute course updates for the interceptors. Typically, tracker activities are computationally demanding, as they employ highly sophisticated algorithms. Thus, they produce location estimates of targets and interceptors, the accuracy of which is a function of their execution times. Generally, the longer the execution time for the trackers, the higher is the accuracy of the location estimates they produce.

A guidance activity computes course updates from the location estimates, the utility of which is a function of *when* course updates are provided to the interceptor,

since the interceptor and the target are moving with respect to each other. The target’s course and speed may vary. The interceptor does not necessarily follow the shortest direct route to the target; it may “zig-zag” its way to the target because of guidance update time variations resulting from resource (e.g., processing, communication) contention. Thus, highly accurate, but significantly late location estimates produced by tracker activities can result in low utility to a guidance activity because the delayed estimates allow less time for the guidance activity to provide timely course updates that will result in successful interception.

Similarly, low accuracy, but significantly early location estimates produced by tracker activities also result in low utility to a guidance activity, because the inaccurate estimates can cause the guidance activity to compute inaccurate course updates, resulting in non-successful interception.

2.2. Definition

We define the “joint utility” of an activity as a function of the *completion time utility* and the *progressive utility* of a set of activities, excluding that of the activity itself. Completion time utility of an activity is expressed by a TUF—i.e., the utility accrued by the activity as function of its completion time. Progressive utility of an activity represents the utility accrued by the activity as a function of its *progress*. We regard progressive utility as an application-specific function of the quality/accuracy of the activity’s output. Generally, the greater the activity’s progress, the greater is the activity’s quality/accuracy, and thus its progressive utility. Later, we define the scope of progressive utility functions in Section 3.4.

Several functions can be used to combine completion time and progressive utilities. A reasonable function is the *weighted sum*, as that allows: (1) the combined utility value to be easily reasoned about by application designers and scheduling algorithms; and (2) the relative importance of completion time and progressive utilities to be reflected in the combined utility value in an application-specific way. Of course, other functions are possible. Later, we define the scope of proposed functions in Section 3.5.

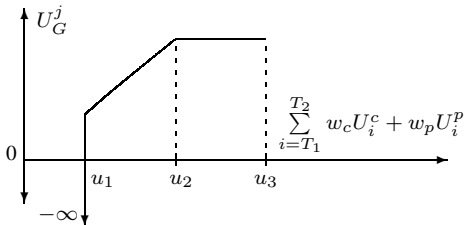


Figure 2. Example JUF of guidance

In Figure 2, we show the joint utility of a guidance activity (U_G^j) as the weighted sum of the completion time utility (U^c) and progressive utility (U^p) of two

(interceptor and target) tracker activities T_1 and T_2 , where w_c and w_p denote the weights of U^c and U^p , respectively. The joint utility function (or JUF) shown in the figure indicates that the guidance progressively accrues increasing joint utility as the weighted sum of the two tracker utilities increases from u_1 to u_2 , because this will give guidance increasing time and location estimate accuracy and thus progressively higher likelihood for successfully engaging the target.

Further, the guidance activities’ joint utility remains constant as the weighted sum of tracker utilities increases from u_2 to u_3 , because the time left and location estimate accuracy are sufficient enough (during the utility range $[u_2, u_3]$) for the activity to cause successful interception.

Finally, guidance accrues infinitively negative joint utility when the weighted sum is less than u_1 , because the time left and/or the location estimate accuracy is not sufficient for acceptably completing the course update calculations of the activity. In such a situation, the activity must be aborted.

2.3. TUFs-Only versus TUFs with JUFs

It is important to observe that there exists the classical precedence dependency between the two tracker activities and the guidance activity—i.e., guidance can perform another repetition only after each of the trackers complete their preceding repetition of tracking. This is because guidance needs location estimates of the target and the interceptor, which are produced by the trackers, for their course update calculations.

It is also important to observe that the joint dependency between the trackers and guidance characterized by the JUF is *not* equivalent to precedence or data dependency. Classical precedence and data dependencies do not include the (joint) dependency’s *time dimension* embodied in the JUF. For example, a precedence dependency only indicates that a “consumer” activity starts execution after the “producer” activity completes, and does not indicate *when* the producer/consumer must start or complete.

However, it is possible to partially characterize the joint dependency using TUFs and thus *not* use JUFs. For example, if either or both of the trackers execute too late in any repetition, then the TUF of the guidance activity can be appropriately (and dynamically) scaled up, so that the scheduler will spend more cycles on the guidance to compensate for the trackers’ lateness. Note that in such a situation, the TUF of the guidance will not be available to the scheduler *until* the trackers complete their preceding repetition, as the process of scaling the guidance TUF can be done only when the trackers’ repetition completion times are known. Thus, until the trackers complete, the scheduler is unaware of the guidance TUF and thus the *time dependency* between the trackers and guidance.

Although the scheduler knows about the precedence dependency, that may not help it from preventing the trackers’ lateness. This is because there may be other activities that can arrive before or after the trackers execute their repetitions and possibly accrue higher utility than the trackers due to properties of their scheduling parameters such as arrival times, TUF shapes, resource needs, and remaining execution times. Such activities can thus cause interferences to the trackers.

Alternately, one can use JUFs. Here, the TUF and the JUF of guidance will be available well before the trackers complete each repetition, as the joint dependency is characterized using the JUF. Now, the scheduler will be aware of the guidance JUF as soon as guidance arrives.¹ This will enable the scheduler to schedule the trackers acceptably early, with sufficient remaining time for making acceptable progress. We hypothesize that such scheduling situations and resulting schedules will increase the likelihood for guidance to complete at acceptable times, increasing the chances for successful interception. In Section 5, we verify this hypothesis and thus verify the usefulness of JUFs.

3. Models and Objectives

3.1. 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.

A thread can be subject to time constraints. Following [3, 7], 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 [3, 7], 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.” Note that TUF-driven scheduling is general enough to schedule non-real-time and real-time threads in a consistent manner: the time constraint of a non-real-time thread is modelled as a constant TUF whose utility represents its relative importance.

It is possible for scheduling segments to be nested or disjoint [3,7]. Thus, a thread can execute inside multiple scheduling segments. When it does so, it is governed by the tightest of the nested time constraints, which is often application-specific (e.g., earliest deadline for step TUFs).

The (known or expected) execution time of a thread’s scheduling segment, which is used for scheduling, is defined using a PUF. We discuss PUFs in Section 3.4.

¹ Guidance can arrive before or after trackers. If it arrives before the trackers, it will be blocked because of the precedence.

3.2. Resource Models

Threads can access non-CPU resources, which in general, are reusable. Examples include physical resources (e.g., disks) and logical resources (e.g., critical code sections guarded by mutexes).

Similar to [3, 7], we consider a single-unit resource model. Resources can be shared and can be subject to mutual exclusion constraints. A thread may request multiple shared resources during its lifetime. The requested time intervals for holding resources may be nested or disjoint (similar to scheduling segments).

We assume that a thread explicitly releases all resources before the end of its execution. Thus, a thread that is requesting a resource must specify the worst-case time that it intends to hold the requested resource.

Threads can have precedence constraints. For example, it is possible that 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 [3, 7], we allow such precedences to be programmed as resource dependencies.

3.3. Completion Time Utility

We specify a thread’s time constraints 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 $U_{T_i}^c(\cdot)$; thus completion of T_i ’s associated scheduling segment at a time t will yield a utility $U_{T_i}^c(t)$. We use U^c to denote the completion time utility, without being thread/scheduling segment specific.

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 in utility. TUFs which are not unimodal are multimodal. Example unimodal TUFs are shown in Figures 1(a) to 1(c). In this paper, we focus on non-increasing unimodal TUFs.

Each thread $U_{T_i}^c, i \in \{1, \dots, n\}$ has an initial time I_i and a termination time X_i . Initial and termination times are the earliest and the latest times for which the TUF is defined, respectively. We assume that I_i is equal to the arrival time of the thread T_i . If T_i ’s X_i 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 allow such time constraints to be specified using TUFs. Thus, handlers are scheduled similar to normal threads.

3.4. Progressive Utility

It is possible for application activities to accrue utility as a function of their *progress* [9]. The class of

“anytime” algorithms (e.g., Monte Carlo) allow for activities whose accuracy can vary with respect to their execution time. In general, the larger the execution time, the greater is the accuracy.

With anytime algorithms, we can thus devise *progressive utility functions* (or PUFs) for application threads, where x -axis defines the thread’s cumulative execution time, and y -axis defines “progressive” utility. We define progressive utility as an application-specific function of the accuracy of thread output. In general, progressive utility may depend on many system/application resources. Here, we focus on execution time demand (for CPU).

A PUF is always associated with a thread scheduling segment. The PUF associated with the scheduling segment of a thread T_i is denoted as $U_{T_i}^p(\cdot)$; thus, T_i will gain a progressive utility of $U_{T_i}^p(e)$, if the associated scheduling segment has executed for e time units. We use U^p to refer to the progressive utility concept, without being thread/scheduling segment specific.

A PUF can be either discrete or continuous on the y -axis. A discrete PUF implies that the thread’s U^p changes only after expending some minimum execution time (e.g., one iteration of an iterative algorithm) for the associated scheduling segment.

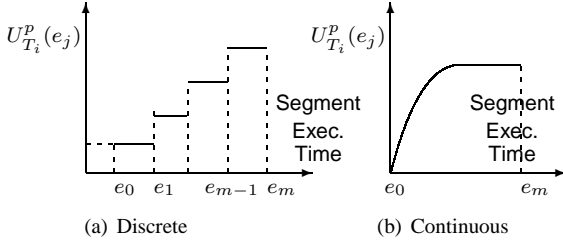


Figure 3. Progressive utility functions

Figure 3(a) shows an example PUF, where the $U_{T_i}^p(\cdot)$ of a thread T_i ’s scheduling segment increases to a new level once the segment has executed for $e_j, j \in \{0, 1, \dots, m\}$ time units. We call e_j , the PUF’s changing points.

In the special case of two-level PUF’s ($m = 1$ in Figure 3(a))—referred to in the literature as *imprecise computations*—a thread’s scheduling segment is divided into a *mandatory* part (by execution for e_0 time) and an *optional* part (by further execution for $(e_1 - e_0)$ time). Execution of the mandatory part yields the minimal acceptable result and execution of the optional part produces a higher quality result. Such two-level PUFs are a basis for flexible scheduling [9]. If m is large, a discrete PUF can be approximated by its continuous counterpart.

In this paper, we consider non-decreasing, unimodal PUFs only in the discrete domain.

3.5. Joint Utility

A JUF describes the utility of the scheduling segment of a thread T_i as a function of a set of thread-scheduling segments’ completion time utility and progressive utility, excluding that of T_i . We refer to such threads as *joint threads* of T_i .

A JUF is always associated with a thread scheduling segment. We denote the JUF of the scheduling segment of a thread T_i as $U_{T_i}^j(T_i.jt, r)$, where $T_i.jt$ is the joint thread set of the scheduling segment, $r \in \mathcal{R}$ is the joint utility accrual rule, and \mathcal{R} is the joint utility accrual rule set. The rule r defines the function on the x -axis of T_i ’s scheduling segment’s JUF. We use U^j to refer to the joint utility notion, without being thread/scheduling-segment specific.

The joint-utility accrual rule set \mathcal{R} for a thread T_i ’s scheduling segment includes the AND rule, where U^j is the weighted sum of U^c and U^p of *all* joint threads in $T_i.jt$ (as motivated in Section 2). Further, \mathcal{R} can also include:

- **OR rule**, where U^j is the weighted sum of U^c and U^p of *any one* joint thread in $T_i.jt$. In the context of the moving target engagement system, the OR rule would be useful when there are multiple tracker activities that use different sensors for tracking targets and interceptors. Thus, guidance will be able to compute an effective course update if *any one* tracker completes at an acceptable time with acceptably accurate estimates of a moving target and interceptor;

- **AND-OR rule**, where some threads in $T_i.jt$ follow the AND model, while others follow the OR model. Again, in the target engagement system context, such an accrual rule would be useful when there are multiple trackers and acceptable completion of some trackers with acceptable accuracy is a must for the guidance activity, perhaps, due to some unique properties of the tracker’s sensors. Further, acceptable completion of the other trackers is optional, due to their similar, redundant sensor characteristics.

- **P-out-of-Q rule**, where U^j is the weighted sum of U^c and U^p of *at least P out of Q threads* in $T_i.jt$. This rule’s motivation is a straightforward extension of that given for the AND-OR rule.

Not all shapes make sense for JUFs. For example, a linearly decreasing JUF is counter-intuitive. Without loss of generality, in this paper, we consider non-decreasing, unimodal JUFs.

3.6. Utility Accrual Scheduling Criteria

Given the models previously described, we consider the scheduling criterion of maximizing the weighted sum of U^c , U^p , and U^j . 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 [3]. Further, it has not been previously studied.

4. The CUA Algorithm

4.1. 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 a “greedy” strategy. A good greedy strategy is to select as many “high return” threads and their dependents into the schedule as early as possible. This will increase the likelihood of maximizing the aggregate utility.

The metric used by CUA to determine the return on investment for a thread is called the *Combined Utility Density* (or CUD), which measures the amount of utility that can be accrued per unit time by executing the thread and the thread(s) that it depends upon.

CUA first decides the execution time $ExecTime$ of a thread T to be the minimum value that yields the highest $U_T^p(ExecTime)$. To compute T 's CUD at time t , CUA considers T 's expected completion time (denoted as t_f), the expected $U^c(t_f)$ by executing T and its dependent threads, and T 's U_T^j . CUD of T is then computed as: $\frac{\sum [U^c(t_f) + U_T^j + U_T^p(ExecTime)]}{t_f - t}$.

4.2. Overview

The scheduling events of CUA include the arrival and 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 CUA, we define the following variables and auxiliary functions:

- \mathcal{T}_r is the current set of unscheduled threads; σ is the ordered schedule. $T_i \in \mathcal{T}_r$ is a thread.
- $T_i.ExecTime$ is T_i 's (known or expected) remaining execution time; $T_i.CE$ is its predicated cumulative execution time.
- $T_i.Dep$ is T_i 's dependency list; $T_i.PartialSched$ is a partial schedule consisting of T_i , and all of, or portions of its dependencies, if it has dependencies.
- $T_i.t_c$ and $T_i.U^{total}$ are the total execution time and utility of $T_i.PartialSched$, respectively.
- Function $Owner(R)$ denotes the thread that is currently holding the resource R , and $reqRes(T)$ denotes the resource that is currently being requested by the thread T .

Algorithm 1 shows a description of CUA at a high level abstraction. When CUA 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 aborted. CUA then builds each thread's dependency chain (line 5), and chooses its execution time (line 8).

Algorithm 1: CUA: high level description

```

1: input:  $\mathcal{T}_r$ ; output:  $\sigma$ ;
2: Initialization:  $t := t_{cur}$ ,  $\sigma := \emptyset$ ;
3: for  $\forall T_i \in \mathcal{T}_r$  do
4:   abort ( $T_i$ ) if it is not feasible;
5:   Build dependency list of  $T_i$ :
    $T_i.Dep := buildDep(T_i)$ ;
6: while  $\exists T_i : T_i \in \mathcal{T}_r \wedge T_i \notin \sigma$  do
7:   for  $\forall T_i : T_i \in \mathcal{T}_r \wedge T_i \notin \sigma$  do
8:      $T_i.ExecTime =$ 
        $\min\{ExecTime | U_{T_i}^p(CE) = \max(U_{T_i}^p)\}$ ;
9:      $\langle T_i.PartialSched, T_i.t_c, T_i.U^{total} \rangle :=$ 
        $createPartialSchedule(T_i, T_i.Dep)$ ;
10:     $T_i.CUD := \frac{T_i.U^{total}}{T_i.t_c}$ ;
11:   Select  $T_k \in \mathcal{T}_r, T_k \notin \sigma$  with the largest CUD;
12:   if  $T_k.CUD > 0$  then
13:      $\sigma := \text{append}(\sigma, T_k.PartialSched, t)$ ;
14:      $t := t + T_k.t_c$ ;
15:   else break;
16: return  $\sigma$ ;

```

The CUD of each thread is computed by considering the thread and all threads in its current dependency chain (line 7–10). The thread with the largest CUD and its dependencies are selected to construct the schedule (line 11), if they can produce a positive utility.

Note that in line 13 the partial schedule is further optimized by the procedure $\text{append}()$, in which the possibly optimized partial schedule is appended at the tail of the existing schedule σ instead of being inserted. This is because of the way we compute a thread's CUD, where we assume that the threads are executed at the current position in the schedule. If the selected partial schedule is inserted into the existing schedule, the previously computed CUDs become void.

Once a partial schedule is added to the tentative schedule, CUA updates the time t , which is the starting time of the next partial schedule, if there exists one (line 14). We call this time variable t , *virtual time*, because it denotes a future time. The algorithm repeats the procedure until it exhausts the unordered list.

4.3. Manipulating Partial Schedules

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

Algorithm 2: buildDep()

```

1: input: Thread  $T_i$ ; output:  $T_i.Dep$ ;
2: Initialization:  $T_i.Dep := T_i$ ;  $PrevT := T_i$ ;
3: while  $reqRes(PrevT) \neq \emptyset \wedge$ 
    $Owner(reqRes(PrevT)) \neq \emptyset$  do
4:    $T_i.Dep := Owner(reqRes(PrevT)) . T_i.Dep$ ;
5:    $PrevT := Owner(reqRes(PrevT))$ ;

```

Algorithm 2 follows the chain of resource request/ownership. For convenience, the input thread T_i is also included in its own dependency list. Each thread T_j other than T_i in the dependency list has a successor thread that needs a resource which is currently held by T_j . Algorithm 2 stops either because a pre-

decessor thread does not need any resource or the requested resource is free. Note that “.” denotes an append operation. Thus, the dependency list starts with T_i ’s farthest predecessor and ends with T_i .

Algorithm 3: createPartialSchedule ()

```

1: input:  $T_i, T_i.Dep, t$ : start time for partial schedule; output:  $T_i.PartialSched, T_i.t_c, T_i.U^{total}$ ;
2: Initialization :  $T_i.PartialSched := \emptyset; T_i.t_c := 0; T_i.U^{total} := 0$ ;
3: for  $\forall T_j \in T_i.Dep \wedge T_j \notin \sigma$ , from head to tail do
4:    $T_i.PartialSched := T_i.PartialSched \cdot T_j$ ;
5:    $T_i.t_c := T_i.t_c + T_j.ExecTime$ ;
6:    $T_i.U^{total} := T_i.U^{total} + U_{T_j}^c(t + T_i.t_c) + U_{T_j}^p + U_{T_j}^j$ ;
7: return  $\langle T_i.PartialSched, T_i.t_c, T_i.U^{total} \rangle$ ;

```

Algorithm createPartialSchedule () (Algorithm 3) accepts $T_i, T_i.Dep$, and the virtual time t . On completion, the algorithm produces a partial schedule for T_i , the total execution time (t_c) and aggregate utility (U^{total}) of the partial schedule by executing it at time t . The algorithm computes the partial schedule by executing threads in $T_i.Dep$ from the current position (at time t) in the schedule, while following the dependencies.

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. This is considered in the for-loop of Algorithm 3 (line 3-6).

Algorithm 4: append () : optimizing the partial schedule

```

1: input:  $\sigma, T_k.PartialSched, t$ : start time for partial schedule; output: updated  $\sigma$ ;
2: Initialization :  $T_k.PS = T_k.PartialSched, U_{tmp} := T_k.U^{total}$ ;
3: for  $\forall T_i \in T_k.PartialSched$  from head to tail do
4:   while  $T_i.ExecTime > 0$  do
5:     Reduce  $T_i.ExecTime$  by 1 time unit, and update  $T_k.PS$ ;
6:      $t_c := 0, U := 0$ ;
7:     for  $\forall T_j \in T_k.PS$  from head to tail do
8:        $t_c := t_c + T_j.ExecTime$ ;
9:        $U := U + U_{T_j}^c(t + t_c) + U_{T_j}^p + U_{T_j}^j$ ;
10:    if  $U > U_{tmp}$  then
11:       $U_{tmp} := U$ ;
12:       $T_k.PartialSched := T_k.PS$ ;
13:       $T_k.t_c := t_c$ ;
14:    else break;
15:  $\sigma := \sigma \cdot T_k.PartialSched$ ;
16: return  $\sigma$ ;

```

In line 13 of Algorithm 1, we further optimize the highest CUD thread T_k ’s partial schedule with the procedure append (), which is shown in Algorithm 4.

Since we consider discrete PUFs, for each thread from head to tail in $T_k.PartialSched$ (line 3 of Algorithm 4), we reduce its execution time to the next changing point (line 5). Such reduction sacrifices its

U^p , but may increase its U^c with the non-increasing TUF we consider, and is prone to affect its U^j . Thus, we compare the changed partial schedule, $T_k.PS$ with $T_k.PartialSched$ and select the one with higher total utility. We continue such execution time reduction until no more aggregate utility is yielded (line 7-14). Finally, the selected $T_k.PartialSched$ is appended at the tail of current schedule.

4.4. Computational Complexity

To analyze CUA’s cost (Algorithm 1), we consider the worst-case with n resource-dependent threads and a maximum of m changing points among the threads’ PUFs. In the worst-case, buildDep () may build a dependency list with a length n , so the for-loop from line 3 to 5 of Algorithm 1 requires $O(n^2)$ time. The while-loop starting at line 6 in Algorithm 1 can be repeated $O(n)$ times in the worst-case, since each thread can only be in one partial schedule.

The worst-case cost of the while-loop body is dominated by createPartialSchedule () (Algorithm 3), which has a cost of $O(n)$, and append () (Algorithm 4), whose cost is $O(mn^2)$. Therefore, CUA’s worst-case complexity is $O(n^2) + n \times [O(n) + O(mn^2)] = O(mn^3)$.

5. Experimental Evaluation

We implemented CUA and five other UA algorithms including GUS [7], DASA [3], LBESA [10], UPA [12], and D^{over} [6]. Two non-UA algorithms, EDF and fixed priority (or FP) were also implemented to compare with the UA algorithms. All the algorithms were implemented in our previously developed scheduling framework called meta-scheduler [8]. The meta-scheduler is an application-level framework for implementing scheduling algorithms on POSIX RTOSes, without RTOS changes. We use the QNX Neutrino 6.2 RTOS in our study.

In our experiments, we select the average thread execution time, which is denoted as C_{avg} , to be exponentially distributed with a mean of 0.5 sec. During each experiment, more than 500 threads were generated with randomly distributed parameters such as thread execution times and termination times.

5.1. Effectiveness of JUFs

We evaluate the effectiveness of JUFs by the previously discussed TUFs-only vs. TUFs with JUFs scenario. We define a JUF between three threads called T_1, T_2 , and G , which are analogous to the trackers and guidance in Section 2.2. The JUF of thread G is similar to the one described in Figure 2. Other threads are randomly generated to interfere with these threads.

Figure 4 shows the accrued utility ratio (or AUR) and termination time meet rate (or XMR) of T_1, T_2 ,

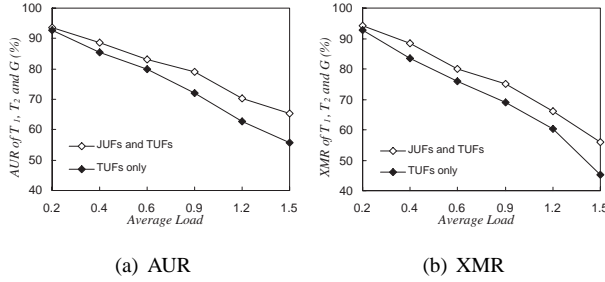


Figure 4. CUA's effectiveness with JUFs

and G under CUA, under increasing average loads. AUR is the ratio of accrued aggregate (completion-time) utility to the maximum possible (completion-time) utility, and XMR is the ratio of the threads meeting their termination times to the total thread releases.

We observe that with TUFs and JUFs, both the accrued aggregate U^c and satisfied termination times of T_1 , T_2 , and G are improved compared with the TUFs-only case, especially under increasing load and thread interference. Since all threads have non-increasing TUFs, this means that CUA can schedule T_1 , T_2 , and G with TUFs and JUFs acceptably early to accrue higher U^c .

5.2. Effectiveness of Utility Accrual

Figures 5(a) and 5(b) show the AUR and XMR of the algorithms for resource-independent threads with step TUFs under increasing loads, respectively. Note that all the algorithms allow this model. With FP, we decide each thread's priority based on the utility of its TUF—the higher utility, the higher priority.

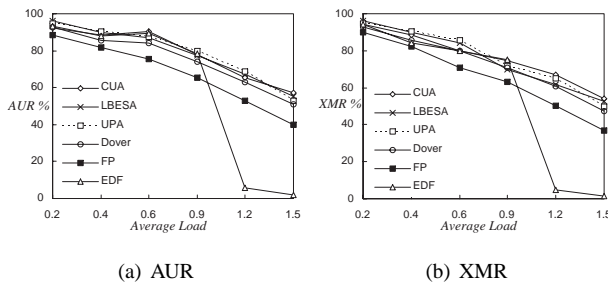


Figure 5. Performance under step TUFs

DASA, GUS, and CUA have very close performance for the entire load range, so we only show the performance of CUA in Figure 5. As the figure shows, UPA has the best performance during light and medium load situations. However, during heavily loaded situations, DASA, GUS, and CUA outperform UPA, D^{over} , EDF and FP. Note that D^{over} exhibits the worst performance among all UA algorithms; FP performs worse than the UA algorithms, and EDF suffers domino effects during overloads [10].

6. Conclusions

In this paper, we extend the utility accrual model with the notion of joint utility functions that allow an activity's utility to be described as a function of the completion times of other activities and their progress. We present the CUA algorithm for such a model. Our experimental results verify our hypothesis: JUFs allow scheduling of the joint-dependent threads acceptably early, with sufficient remaining time for making acceptable progress. Moreover, the resulting schedules increase the aggregate completion time utility of the joint-dependent threads.

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

References

- [1] K. Chen and P. Muhlethaler. A scheduling algorithm for tasks described by time value function. *Journal of Real-Time Systems*, 10(3):293–312, May 1996.
- [2] R. Clark, E. D. Jensen, and et al. An adaptive, distributed airborne tracking system. In *WPDRTS*, volume 1586 of *LNCS*, pages 353–362. Springer-Verlag, April 1999.
- [3] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, CMU, 1990. CMU-CS-90-155.
- [4] Globalsecurity.org. Precision direct attack munition, on-target weapon, long-range, affordable moving surface target engagement. <http://www.globalsecurity.org/military/systems/munitions/amste.htm/>.
- [5] 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.
- [6] 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.
- [7] 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>.
- [8] P. Li, B. Ravindran, et al. Choir: A real-time middleware architecture supporting benefit-based proactive resource allocation. In *IEEE ISORC*, May 2003.
- [9] J. W. S. Liu et al. Imprecise computations. *IEEE*, 82(1):83–94, January 1994.
- [10] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986. CMU-CS-86-134.
- [11] D. P. Maynard, S. E. Shipman, et al. An example real-time command, control, and battle management application for alpha. Technical Report Archons Project TR-88121, CMU CS Dept., December 1988.
- [12] 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.