

On the Joint Utility Accrual Model

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 Jensen’s 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 activity utility as a function of activity 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.

1 Introduction

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

Further, 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 an image understanding service depends upon how accurately the objects in the image can be recognized.

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 course updates to an in-flight interceptor (to compensate for changes in the direction of the target and navigational inaccuracies) depends upon when tracking services that produce location estimates of the interceptor and target complete. Furthermore, it depends on the accuracy of the estimates. If the tracking services produce estimates that are too late, even though the estimates are highly accurate, the estimates will probably be too stale to be useful for course update calculations, as the target and the interceptor are continuously moving. Moreover, if the estimates are produced too early, but inaccurate, the inaccuracy (of the estimates) can cause course update calculations to be ineffective.

Service requests in many emerging real-time systems often cause resource contention, as there are many resources that are shared. Example shared resources include processors, storage devices, communication devices and channels, power, and application-devices such as sensors and actuators. Sharing of resources leads to resource contention. 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.

1.1 Time/Utility Functions

Jensen’s time/utility functions [5] (TUFs) allow specification of application QoS. A TUF specifies the utility to the system (derived from application QoS) of completing an activity as an application- or situation-specific function of *when* that activity completes. Figure 1 shows example time constraints specified using TUFs.

When time 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. We call such criteria, Utility Accrual (UA) criteria, and scheduling algorithms that consider UA criteria, UA scheduling 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, 7, 9, 11, 13].

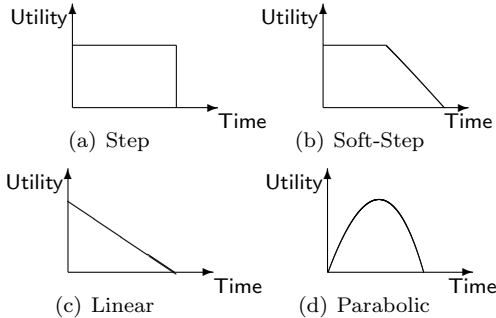


Figure 1: Time Constraints Specified Using TUFs

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 [8].

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. We implement CUA on a POSIX RTOS and verify the effectiveness of JUFs.

The rest of the paper is organized as follows: In Section 2, we introduce JUFs, discussing their motivation and rationale. In Section 3, we outline our activity and utility models and state the UA scheduling criterion. We present the CUA algorithm in Section 4 and report our experimental measurements 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 application contexts, it is possible for an activity to accrue utility that depends on when

other activities complete and the accuracy of their results. We call this notion of utility as *joint utility*. To illustrate the joint utility concept, we describe a notional, combat system application scenario that is derived from [4].

The scenario includes a moving 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. (The semantics of the tracker activities are similar to that of the AWACS application described in [2].) Further, the platform contains *guidance* activities that monitor the progress of in-flight interceptors and provides course updates to the interceptors, in order to compensate for changes in the direction of targets and for navigational inaccuracies.

The weapons platform contains *weapons control* activities that determine launch parameters of interceptor weapons and issue launch commands and initial guidance information.

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, like the AWACS association activity [2], are computationally demanding, as they employ highly sophisticated, computationally-intensive algorithms for object recognition. Thus, they produce location estimates of targets and interceptors, the accuracy of which is a function of their execution times. Generally, longer the execution time for the trackers, higher is the accuracy of the location estimates produced by them.

However, a guidance activity computes course updates from the location estimates, the utility of which is a function of *when* the course updates are provided to the interceptor, since the interceptor is moving with respect to the target. 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 little time for the guidance activity to provide timely course updates that will result in successful interception.

Similarly, low accurate, 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 thus define the joint utility of an activity as a function of the *completion-time utility* and *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, greater the activity’s progress, greater is the activity’s quality/accuracy, and thus its progressive utility. Later, we define the scope of progressive utility functions in Section 3.5.

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 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 3.6.

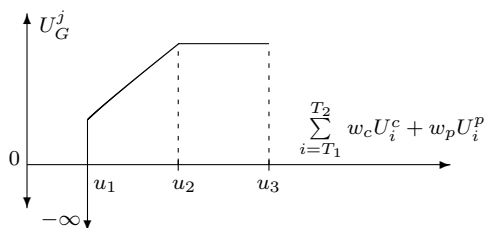


Figure 2: Example JUF of Guidance

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 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 activity 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 the guidance activity increasing time and location estimate accuracy and thus progressively high likelihood for successfully engaging the target.

Further, the guidance’s joint utility remains the same 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 infinitely 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 tracker activities and the guidance activity i.e., the guidance activity can execute only after the tracker activities complete. This is simply because guidance needs location estimates of target and interceptor, which are produced by the trackers, for its 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 the trackers are too late, then one can appropriately (and dynamically) scale-up the TUF of the guidance, 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, as the process of scaling the guidance TUF can be done only when the trackers’ completion-times are known. Thus, until the trackers complete, the scheduler is completely unaware of the guidance TUF and thus the *time dependency* between the trackers and guidance.

Note that 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 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 interference to the trackers.

Alternately, one can use JUFs. Here, the TUF and the JUF of the guidance will be available

well before the trackers complete, 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 arbitrarily arrive and can be arbitrarily preempted.

A thread can be subject to time constraints. Following [5], 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 [5], we call a thread a “real-time thread” while it is 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 [5, 7]. Thus, a thread can execute inside multiple scheduling segments. When a thread does so, its time constraint is often application-specific (e.g., earliest deadline for “downward step” TUFs).

The execution time of a thread’s scheduling segment, which is used for scheduling, is defined using a PUF. We discuss PUFs in Section 3.5.

3.2 Resource Model

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. Further, resources can be shared and can be subject to mutual exclusion constraints. Moreover, a thread may request multiple

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

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. This time, called *HoldTime* (as in [3, 7]), is used for scheduling.

3.3 Precedence Constraints

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 precedence constraints to be programmed as resource dependencies.

3.4 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 refer to the notion of 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)–1(d). In this paper, we focus on non-increasing unimodal TUFs.

Each TUF $U_{T_i}^c, i \in \{1, \dots, n\}$ 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. If a thread 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.

3.5 Progressive Utility

It is possible for application activities to accrue utility as a function of their *progress* [8]. The class of “anytime” algorithms (e.g., Monte Carlo) allow for activities whose accuracy can vary with respect to their execution time. In general, larger the execution time, 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 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 the 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.

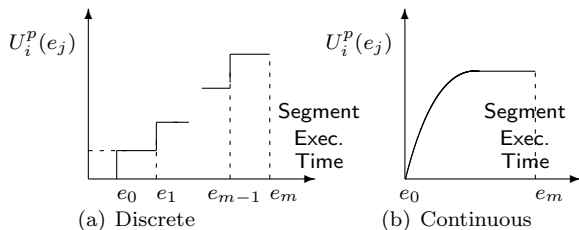


Figure 3: Progressive Utility Functions

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.

With two-level PUFs ($m = 1$ in Figure 3(a)), a thread’s scheduling segment can be split 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 the basis for flexible scheduling [8] and multi-version applications [10]. Note that if m is large, a discrete PUF can be approximated by its continuous counterpart.

In this paper, we consider non-decreasing, unimodal PUFs (without loss of generality) in both discrete and continuous domains.

3.6 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.

3.7 Utility Accrual Scheduling Criteria

Given the models previously described, several UA scheduling optimality criteria can be devised. In this paper, we consider the 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 is *open*. We present the first algorithm called CUA, for this problem.

4 The CUA Algorithm

4.1 Algorithm Rationale and Overview

The potential utility that can be accrued by executing a thread defines a measure of its “return of investment”. Because of the unpredictability of future events, scheduling events that may happen later such as new thread arrivals cannot be considered at the time when the scheduler is invoked. Thus, a reasonable heuristic is the “greedy” strategy, which means selecting as much “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 of 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 e of a thread T_i to be the minimum value that yields the highest $U_{T_i}^p(e)$. To compute T 's CUD at time t , CUA considers T 's expected completion time (denoted as t_c), the expected $U^c(t_c)$ by executing T and its dependent threads, and T_i 's $U_{T_i}^j$. CUD of T_i is then computed as: $\frac{\sum[U^c(t_c) + U_{T_i}^j + U_{T_i}^p(e)]}{t_c}$.

4.2 Algorithm Internals

The scheduling events of CUA include the arrival of a thread, the completion of a thread, a resource request, and a resource release. A description of CUA at a high level of abstraction is shown in Algorithm 1. 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.
- $T_i.Dep$ is the dependency list of T_i , and $T_i.PartialSched$ is a partial schedule which consists of T_i and its dependents. Note that the partial schedule may contain only portions of T_i 's dependents. $T_i.e$ is T_i 's execution time.
- $T_i.t_c$ and $T_i.U^{total}$ are the total execution time and utility of $T_i.PartialSched$, respectively. $T_i.U^c(T_i.t_c)$ is the completion-time utility accrued by T_i when executing $T_i.PartialSched$.
- Function `Owner(R)` denotes the thread that is currently holding the resource R , and `reqResource(T)` denotes the resource that is currently being requested by the thread T . The `holdTime(T,R)` returns the holding time that is desired for a resource R by a thread T .

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

Algorithm 1: High Level Description of the CUA Algorithm

```

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

```

is later than its termination time, it can be safely aborted. CUA then chooses the execution time and builds the chain of dependencies for each thread (lines 6-7). The CUD of each thread are computed (lines 8-9) by considering the thread and all threads in its dependency chain. The thread with the largest CUD and its dependencies are added into the current schedule (lines 10-12).

Note that a partial schedule is appended at the tail of the existing schedule (Algorithm 1, line 11), 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 13). 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. Finally, the sub procedure `scheduleOptimize()` is invoked to further increase the accrued utility, since re-ranking of schedule σ can yield results as good as, if not better than, the schedule before re-ranking.

4.2.1 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 simply 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

Algorithm 2: buildDep()

```

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

```

stops either because a predecessor thread does not need any resource or the requested resource is free. Note that we use the operator “ \cdot ” to denote 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
   sched.; 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: /* consider threads in  $T_i$ ’s dep chain */;
4: for  $\forall T_j \in T_i.Dep \wedge T_j \neq T_i$ , from head to tail
   do
5:    $R :=$  reqResource( $T_j \rightarrow Next$ );
6:    $T_i.PartialSched := T_i.PartialSched \cdot T_j$ ;
7:    $T_i.t_c := T_i.t_c + holdTime(T_j, R)$ ;
8:    $T_i.U^{total} :=$ 
    $T_i.U^{total} + U_{T_j}^c(t + T_i.t_c) + U_{T_j}^p + U_{T_j}^j$ ;
9: /* consider  $T_i$  itself */;
10:  $T_i.PartialSched :=$ 
    $T_i.PartialSched \cdot \langle T_i, T_i.ExecTime \rangle$ ;
11:  $T_i.t_c := T_i.t_c + T_i.ExecTime$ ;
12:  $T_i.U^{total} := T_i.U^{total} + U_{T_i}^c(t_c) + U_{T_i}^p + U_{T_i}^j$ ;
13: return  $\langle T_i.PartialSched, T_i.t_c, T_i.U^{total} \rangle$ ;

```

The createPartialSchedule() algorithm (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 assuming that threads in $T_i.Dep$ are executed 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 release the resources that are needed to execute T_i (lines 5-8 of Algorithm 3); and (2) the remaining execution time of T_i itself (lines 10-11 of Algorithm 3).

If the selected partial schedule contains a thread T_i , it needs to be removed from the unordered thread list \mathcal{UT} . CUA uses another sub procedure called delPartialSchedule() to delete a partial schedule from an unordered thread list, as shown in Algorithm 4. This function examines the partial schedule due to thread T_i , from the head to the tail. If a thread T in the partial schedule has been determined to execute, it may release

Algorithm 4: delPartialSchedule(): Removing a Partial Schedule from a Thread List

```

1: input:  $T_i.PartialSched$  and an unordered
   thread list  $\mathcal{UT}$ ; output: a reduced list  $\mathcal{UT}'$ ;
2: copy  $\mathcal{UT}$  into  $\mathcal{UT}'$ ;  $\mathcal{UT}' := \mathcal{UT}$ ;
3: for  $\forall \langle T, Time \rangle \in T_i.PartialSched \wedge T \neq T_i$ 
   from head to tail do
4:   for  $\forall \langle R, HoldTime \rangle \in T.HeldResources$  do
5:     Update HoldTime:
      $HoldTime := HoldTime - Time$ ;
6:     if  $HoldTime := 0$  then
7:        $T.HeldResource :=$ 
        $T.HeldResource - \langle R, HoldTime \rangle$ ;
8:        $R.Owner := \emptyset$ ;
9:   Remove  $T$  from  $\mathcal{UT}'$ :  $\mathcal{UT}' := \mathcal{UT}' - \{T\}$ ;
10: Remove  $T_i$  from  $\mathcal{UT}'$ :  $\mathcal{UT}' := \mathcal{UT}' - \{T_i\}$ ;
11: return  $\mathcal{UT}'$ ;

```

one or more resources during the allocated time. Therefore, the HoldTimes of the resources that are currently held by T are also updated (lines 5-8). Then, T is removed from \mathcal{UT}' (line 9). Finally, thread T_i (recall that the partial schedule is due to thread T_i) is removed from \mathcal{UT}' (line 10).

4.2.2 Schedule Optimization

The schedule obtained after the while-loop of Algorithm 1 can be further optimized by the scheduleOptimize procedure at line 14.

Algorithm 5: scheduleOptimize(): Fine-tuning the ordered schedule

```

1: input:  $\sigma$ ; output: an optimized schedule  $\sigma'$ ;
2: Initialization :  $\sigma' := \emptyset$ ;
3: for  $\forall T_i \in \sigma$  from head to tail do
4:   Intermediate queue  $\sigma_{opt} := \emptyset$ ;
5:   Intermediate variable  $U_{opt} = 0$ ;
6:   for  $\forall T_j \in \sigma'$  from head to tail do
7:     Insert  $T_i$  before  $T_j$  to get  $\sigma_{tmp}$ ;
8:      $U := U^c(\sigma_{tmp}) + U^j(\sigma_{tmp}) + U^p(\sigma_{tmp})$ ;
9:     Reduce  $T_i.e$  by  $\delta$  and repeat line 7
     and 8 until  $U$  doesn’t increase;
10:    if  $U > U_{opt}$  then
11:       $\sigma_{opt} := \sigma_{tmp}$ ;
12:       $U_{opt} := U$ ;
13:    $\sigma' := \sigma_{opt}$ ;
14: return  $\sigma'$ ;

```

Execution of two ordered schedules that contain the same thread set, but with different orders will probably accrue different utilities. Threads sorted in different orders have different completion times, which can affect the accrued U^c and U^j . Changing the execution times of threads can affect the accrued U^p . Consider two such schedules, σ_a and σ_b . Let $\Delta_{a,b} = U^c(\sigma_a) + U^j(\sigma_a) + U^p(\sigma_a) - (U^c(\sigma_b) + U^j(\sigma_b) + U^p(\sigma_b))$. If $\Delta_{a,b} > 0$, then schedule σ_a will yield a higher aggregate utility than σ_b .

Thus, in scheduleOptimize() (Algorithm 5), we examine threads from σ , from head to tail,

which is the output schedule of the *while*-loop in Algorithm 1. Each thread T is inserted into a tentative schedule σ' , which is initially empty. Upon each insertion, we compare the Δ of possible schedules obtained by inserting T into all possible locations in σ' and find the best, denoted as σ_{opt} . Note that in line 9 of Algorithm 5, we seek to change the execution time of each thread, so as to find a better value for higher accrued utility.

5 Experimental Evaluation

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. Other threads are randomly generated to interfere with these threads.

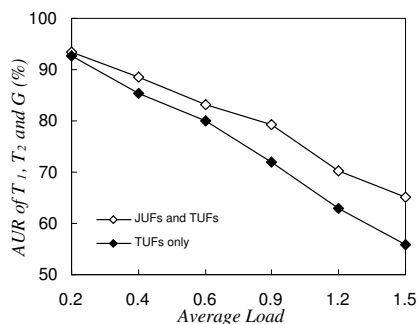


Figure 4: TUFs-Only vs. TUFs with JUFs

Figure 4 shows the accrued utility ratio (AUR) of T_1 , T_2 , and G scheduled by CUA on a Pentium PC, running the QNX real-time OS [12] and our previously developed *meta-scheduler* UA scheduling framework [6]. AUR is the ratio of accrued aggregate U^c to the maximum possible U^c .

We observe that with TUFs and JUFs, the accrued aggregate U^c of T_1 , T_2 , and G is 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 .

6 Conclusions

Our experimental results thus 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. *Real-Time Systems*, 10(3):293–312, May 1996.
- [2] R. Clark et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, volume 1586, pages 353–362, April 1999.
- [3] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990. CMU-CS-90-155.
- [4] GlobalSecurity.org. Precision direct attack munition (pdam), on-target weapon, long-range (owl), affordable moving surface target engagement (amste). <http://www.globalsecurity.org/military/systems/munitions/amste.htm/>.
- [5] E. D. Jensen. Asynchronous decentralized real-time computer systems. In *Real-Time Computing*, NATO Advanced Study Institute. Springer Verlag, October 1992.
- [6] P. Li et al. Choir: A real-time middleware architecture supporting benefit-based proactive resource allocation. In *IEEE ISORC*, pages 292–299, May 2003.
- [7] P. Li et al. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. <http://nile.ece.vt.edu/submissions/GUS-TOC03.zip>, August 2003. Under review at IEEE TC.
- [8] J. W. S. Liu et al. Imprecise computations. *IEEE*, 82(1):83–94, January 1994.
- [9] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986. CMU-CS-86-134.
- [10] C. Lu et al. Feedback control real-time scheduling: Framework, modeling and algorithms. *Real-Time Systems*, 23(1/2):85–126, 2002.
- [11] D. Mosse et al. Value-density algorithm to handle transient overloads in scheduling. In *IEEE ECRTS*, pages 278–286, June 1999.
- [12] QNX. Qnx neutrino rtos. http://www.qnx.com/products/ps_neutrino/.
- [13] 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.