A Utility Accrual Scheduling Algorithm for Real-Time Activities With Mutual Exclusion Resource Constraints

Peng Li, Haisang Wu, Binoy Ravindran, and E. Douglas Jensen

Abstract

This paper presents a uni-processor real-time scheduling algorithm called the Generic Utility Scheduling algorithm (which we will refer to simply as GUS). GUS solves a previously open real-time scheduling problem—scheduling application activities that have time constraints specified using arbitrarily shaped time/utility functions, and have mutual exclusion resource constraints. A time/utility function is a time constraint specification that describes an activity's utility to the system as a function of that activity's completion time. Given such time and resource constraints, we consider the scheduling objective of maximizing the total utility that is accrued by the completion of all activities. Since this problem is $\mathcal{NP}$-hard, GUS heuristically computes schedules with a polynomial-time cost of $O(n^3)$ at each scheduling event, where n is the number of activities in the ready queue. We evaluate the performance of GUS through simulation and by an actual implementation on a real-time POSIX operating system. Our simulation studies and implementation measurements reveal that GUS performs close to, if not better than, the existing algorithms for the cases that they apply. Furthermore, we analytically establish several properties of GUS.

Index Terms

Real-time scheduling, time/utility functions, utility accrual scheduling, resource dependency, mutual exclusion, overload management, resource management

Peng Li is with Microsoft Corporation, Redmond, WA 98052. E-mail: pengli@microsoft.com. This work was performed while he was at Virginia Tech.

Haisang Wu and Binoy Ravindran are with the Real-Time Systems Laboratory, Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061. E-mail: {hswu02,binoy}@vt.edu.

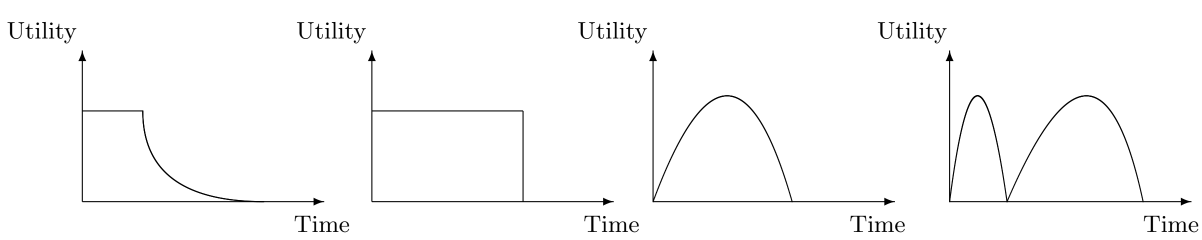
E. Douglas Jensen is with the MITRE Corporation, Bedford, MA 01730-1420, E-mail: jensen@mitre.org.

I. Introduction

Real-time computing is fundamentally concerned with satisfying application time constraints. The most widely studied time constraint is the deadline. A deadline time constraint for an application activity essentially implies that completing the activity before the deadline implies the accrual of some "utility" to the system and that utility remains the same if the activity were to complete *anytime* before the deadline. With deadline time constraints, one can specify the hard timeliness optimality criterion of "always meet all hard deadlines" and use hard real-time scheduling algorithms [1] 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 [2] and battle management [3]. 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 [4] (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 example soft time constraints specified using TUFs.



(a) (b) (c) (d)

Fig. 1: Soft Timing Constraints Specified Using Jensen's Time-Utility Functions

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—

e.g., maximizing the sum [5], or the expected sum [6], 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 criteria, such as resource dependencies and precedence constraints.

Scheduling tasks with non-step TUFs has been studied in the past, most notably in [6] and [7]. However, to the best of our knowledge, Locke's Best Effort Scheduling Algorithm [6], called LBESA, is the only algorithm that considers almost arbitrarily shaped TUFs.

Besides arbitrarily shaped TUFs, dependencies often arise between tasks due to the exclusive use of shared non-CPU resources. Sharing of resources that have mutual exclusion constraints between deadline-constrained tasks has received significant attention in the past [8], [9]. However, little work has been done for sharing resources (that have mutual exclusion constraints) between tasks that have time constraints expressed using TUFs. In [5], Clark considers mutual exclusion resource dependencies, but for tasks with only step TUFs. Furthermore, none of the prior research on step TUFs [10], [11] and non-step TUFs [7], [6], [12] consider resource dependencies.

In this paper, we encompass these two task models. That is, we consider the problem of scheduling tasks that have their time constraints specified using arbitrarily shaped TUFs, and have mutual exclusion resource dependencies. This scheduling problem can be shown to be $\mathcal{NP}$-hard. We present a heuristic algorithm for this problem, called the Generic Utility Scheduling algorithm, which we will refer to simply as GUS. GUS has a polynomial-time complexity of $O(n^3)$ at every scheduling event, given n tasks in the ready queue.

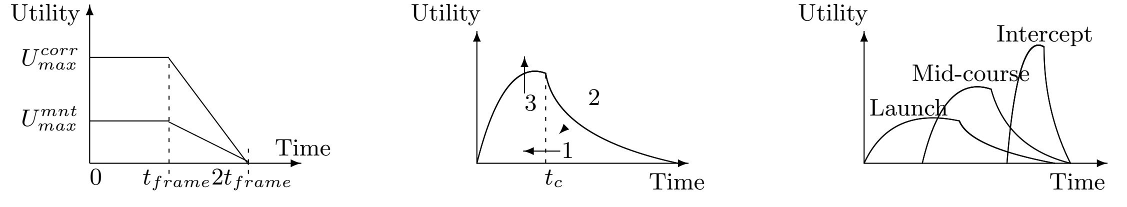
We study the performance of GUS through simulation and implementation. The simulation studies reveal that GUS performs very close to, if not better than, the best existing algorithms for the cases to which they apply (subsets of the cases considered by GUS). Furthermore, we implement GUS and several other existing algorithms on top of the QNX Neutrino real-time operating system using POSIX API's. Our implementation measurements reveal the strong effectiveness of the algorithm.

The rest of the paper is organized as follows. We first overview a real-time application to provide the motivating context for soft time constraints in Section II. In Section III, we introduce our task and resource models, and the scheduling objective. Section IV discusses the heuristics employed by GUS and their rationale. Before describing the GUS algorithm, we introduce notations used in the descriptions and GUS' deadlock handling mechanism in Section V. We describe the GUS algorithm in Section VI. Section VII analyzes the computational complexity of GUS. We establish several non-timeliness properties of the algorithm in Section VIII. Performance evaluations through simulation and implementation are presented in Sections IX and X, respectively. We compare and contrast past and related efforts with GUS in Section XI. Finally, the paper concludes by describing its contributions and future work in Section XII.

II. Motivating Application Examples

As an example of real-time control systems requiring the expressiveness and adaptability of soft yet mission-critical time constraints, we summarize an application from the defense domain: a coastal air defense system [13] that was built by General Dynamics (GD) and Carnegie Mellon University (CMU).

Time constraints of two application activities in the GD/CMU coastal air defense system—called *radar plot correlation* and *track database maintenance*—have similar semantics. The correlation activity is responsible for correlating plot reports that arrive from sensor systems against a tracking database. The maintenance activity periodically scans the tracking database, purging old and uncorrelated reports so that stale information does not cause errors in tracking.



(a) Correln. & DB Maint.　(b) Missile Control　(c) Missile Control Shapes

Fig. 2: TUFs of Three Activities in GD/CMU Coastal Air Defense

Both activities have "critical times" that correspond to the radar frame arrival rate: It is best if both are completed before the arrival of next data frame. However, it is acceptable for them to be late by one additional time frame under overloads. Furthermore, the correlation activity has a greater utility to the system during overloads. TUFs in Figure 2(a) reflect these semantics.

The *missile control activity* of the air defense system provides timely course updates to guide the intercepter such that the hostile targets can be destroyed. However, the frequency and importance of course updates at the desired time depend upon several factors.

As the distance between the target and the interceptor decreases, more frequent course corrections are needed (see arrow 1 in Figure 2(b)). In the meanwhile, it is best to abort a late update and restart course correction calculations with fresh information. Arrow 2 in Figure 2(b) illustrates how this requirement is reflected by a decrease in the utility obtained for completing the course correction activity after the critical time.

The utility of successfully intercepting a target depends upon the "threat potential" of the target. The threat potential depends upon changing parameters such as the distance of the target from the coastline. For example, intercepting a target that has deeply penetrated inside the coastline yields higher utility than a target that is farther away from the coastline. This is reflected by arrow 3 in Figure 2(b) that shows how the function is scaled upward as the threat increases. Figure 2(c) shows how the shape of the TUF dynamically changes.

A more recent application, called AWACS (Airborne WArning and Control System) surveillance mode tracker system [14] was built by The MITRE Corporation and The Open Group, and used similar TUFs for describing time constraints and scheduling (see Figure 3).

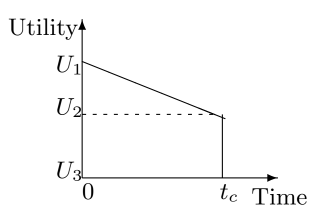


Fig. 3: Track Association TUF in MITRE/TOG AWACS

III. Models and Objectives

This section describes the task and resource models, and the optimization objectives of GUS. In describing the models, we outline the scope of the research.

A. Task and Resource Models

1) Threads and Scheduling Segments: We consider the "thread" abstraction—a single flow of execution—as the basic scheduling entity in the system. Thus, the application is assumed to consist of a set of threads, denoted as $T_i, i \in \{1, 2, ..., n\}$, where the number of threads n is unrestricted. In this paper, a "thread" is equivalent to a "task" or a "job" in the literature. Threads can arrive arbitrarily and can be preempted arbitrarily.

A thread can be subject to time constraints. Following [15], 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 [15], 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: the time constraint of a non-real-time thread is modeled as a constant TUF whose utility represents its relative importance.

GUS allows disjointed and nested scheduling segments (see Figure 4 for an example). Thus, it is possible that a thread executes inside multiple scheduling segments. If that is the case, GUS uses the "tightest" time constraint for scheduling, which is application-specific (e.g., the earliest deadline for step TUFs). Therefore, for a real-time thread, the scheduler only uses one time constraint for scheduling purpose at any given time. From the perspective of the scheduler, a scheduling segment corresponds to a real-time thread. Thus, the terms "scheduling segment," "thread," and "task" are used interchangeably in the rest of the paper, unless otherwise specified.

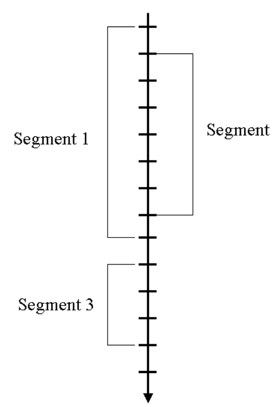


Fig. 4: Example Scheduling Segments

2) Basic Assumptions on Resource Model: To model non-CPU resources and resource requests, we make the following assumptions: (A.1) Resources are reusable and can be shared, but have mutual exclusion constraints. Thus, only one thread can be using a resource at any given time; (A.2) Only a single instance of a resource is present in the system; and (A.3) A resource request (from a thread) can only request a single instance of a resource.

Assumption A.1 applies to physical resources, such as disks and network segments, as well as logical resources such as critical code sections that are guarded by mutexes. Assumption A.2 implies that if multiple identical resources or multiple instances of the same resource are available, each identical instance of a resource should be considered as a distinct resource.

Furthermore, Assumption A.2 requires that a thread explicitly specifies which resource it wants to access. This is exactly the same resource model as assumed in protocols such as Priority Inheritance Protocol [8] and Priority Ceiling Protocol [8].

Without loss of generality, we make Assumption A.3 mainly for practical reasons. If multiple resources are needed for a thread to make progress, the thread must acquire all the resources through a set of consecutive resource requests.

3) Resource Request and Release Model: During the life time of a thread, it may request one or more shared resources. In general, the requested time intervals of holding resources may be overlapped.

We assume that a thread can explicitly release resources before the end of its execution. Thus, it is necessary for a thread that is requesting a resource to specify the time to hold the requested resource. We refer to this time as $HoldTime$. The scheduler uses the $HoldTime$ information at run time to make scheduling decisions.

4) Abortion of Threads: There are several reasons to abort a thread. First a thread may have to be aborted to resolve a deadlock. Secondly and more commonly, in case of resource requests, the scheduler may decide to abort the current owner thread and grant the resource to the requesting thread. The motivation for doing so is that executing the latter thread (that became eligible to execute with the granting of the resource) may lead to greater total timeliness utility than executing the former owner thread, in spite of the overhead associated with doing so.

Aborting a thread usually involves necessary cleanup operations by both the system software (e.g., operating system or middleware) and one or more exception handlers in the application (in that order of execution). We refer to the time consumed by this cleanup as *AbortTime* [1], and say that the thread is executing in `ABORT` mode during that time. Otherwise, the thread is executing in `NORMAL` mode.

Furthermore, some threads cannot necessarily be aborted at arbitrary times, or even cannot be aborted at all. Often, application-specific properties of the controlled physical environment require that the environment's state be transitioned to a physically safe and stable state before a thread can be aborted. We refer to this aspect of a thread as its "abortability." For those threads that can be aborted, an application can specify the allowable abortion points.

As an example, the POSIX specification allows two types of abortions (or "cancellation" in POSIX terminology): (1) a thread abortion can take effect any time during the execution of the thread, called "asynchronous cancellation"; and (2) a thread abortion can only happen at some well defined cancellation points, called "deferrable cancellation."

If a thread can be asynchronously cancelled (or aborted), execution time of its cleanup handler(s) is measured as *AbortTime* in our model. In case that abortions can only happen at well-defined cancellation points, *AbortTime* consists of the execution time of the thread cleanup handler(s) and the execution time from acquiring the resource until the nearest cancellation point. [2] The exception is for the case where the nearest cancellation point happens *after* the resource is released. For this case, *AbortTime* should be set to infinity, to indicate that the thread cannot be aborted while it is holding the resource. Likewise, the infinite *AbortTime* can be used for other cases where it simply means that a thread is not abortable, holding a resource or not.

B. 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

[1] The POSIX specification [16] uses `pthread_cancel()` to force a thread terminate. Thread cancellation handlers (similar to exception handlers) should be invoked before the designated thread terminates. Thus, execution times of the cleanup handlers are measured as *AbortTime* in our model.

[2] This time interval only measures the *upper bound* on the time needed to reach the nearest cancellation point. At run-time, a thread may need less time to reach the nearest cancellation point, because the thread may have held the resource for some amount of time.

results.

Precedence constraints between tasks can also be modeled as resource dependencies. The precedence constraint that T_j precedes T_i is equivalent to the situation where T_i requires a logical resource (before it can start its execution) that is available only after T_j has completed its execution. Thus, if T_j has completed its execution before T_i arrives, then this logical resource is immediately available for T_i and T_i becomes eligible to execute upon arrival. This respects the precedence relation semantics. Furthermore, if T_j has not completed its execution when T_i arrives, then the logical resource is not available, and therefore T_i is conceptually blocked upon arrival. Later, when T_j competes its execution, the logical resource becomes available and T_i is unblocked. This again respects the semantics of the precedence relation. This technique requires both T_i and T_j share a binary semaphore S with an initial value zero. The first operation of T_i is to execute $P(S)$ and the last operation of T_j is to execute $V(S)$.

Thus, by allowing resource dependencies in the task model, we also allow, albeit indirectly, precedence constraints between tasks.

C. Time-Utility Functions and A Soft Timeliness Optimality Criterion

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 $U_i(t)$.

A TUF $U_i, i \in [1, n]$ has an initial time I_i and a termination time TM_i. Initial time is the earliest time for which the function is defined and termination time is the latest time for which the function is defined. That is, $U_i(\cdot)$ is defined in the time interval of $[I_i, TM_i]$. Beyond that, $U_i(\cdot)$ is undefined. If the termination time of U_i is reached and the thread has not finished execution (of the scheduling segment) or has not begun to execute, an exception is raised. Usually, the exception causes abortion of the thread. We discuss details of how GUS handles this exception in Section VI-C.

Furthermore, a TUF is allowed to take arbitrary shapes, as shown in Figure 5. For $t \in [I_i, TM_i]$, $U_i(t)$ could be positive, zero, or negative. However, U_i does not need to have zero or negative values—i.e., it may never "touch" the time axis. This kind of TUFs

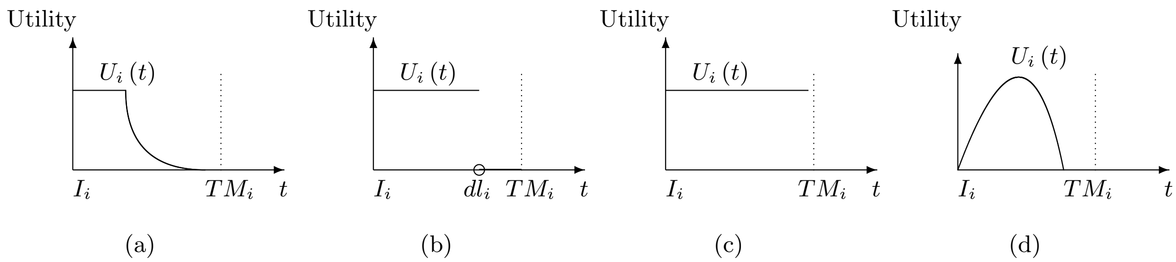


Fig. 5: Example Time-Utility Functions

implies that completion of an activity can always yield some utility to the system no matter when the activity finishes, which is particularly useful for describing non real-time activities. For example, a constant TUF (see Figure 5(c)) can be used for representing a non-time constrained activity: the height of the constant TUF could be used as a way for expressing the activity's relative importance.

Note that our model does not use the "deadline" notation as in hard real-time computing. However, a deadline time constraint can be specified as a step time/utility function (see Figure 5(b)). That is, completing the activity before the deadline accrues some uniform utility and accrues zero utility otherwise.

D. Utility Accrual Scheduling Criterion

Given time/utility functions to describe the time constraints of dependent threads, we consider the soft timeliness optimality criterion of maximizing the total timeliness utility that is accrued by the completion of all threads, i.e., *maximize* $\sum_{i=1}^{n} U_i(f_i)$, where f_i is the finishing time of thread T_i.

This scheduling problem is $\mathcal{NP}$-hard, as it subsumes the problems of: (1) scheduling dependent tasks with step-shaped time/utility functions; and (2) scheduling independent tasks with non step-shaped, but non-increasing time/utility functions. Both these scheduling problems have been shown to be $\mathcal{NP}$-hard in [5], and in [7], respectively. The GUS algorithm presented here is therefore a heuristic algorithm that seeks to maximize the total accrued utility while respecting all thread dependencies.

IV. Algorithm Rationale

The key concept of GUS is the metric called <u>P</u>*otential* <u>U</u>*tility* <u>D</u>*ensity* (or PUD), which was originally developed in [5]. [3] The PUD of a thread simply measures the amount of value (or utility) that can be accrued per unit time by executing the thread and the thread(s) that it depends upon; it essentially measures the "return on investment" for the thread. Furthermore, by considering the dependent threads in computing the PUD, we explicitly account for the dependency relationships among the threads.

Since we cannot predict the future, the scheduling events that may happen later cannot be considered at the time when the scheduler is invoked. The scheduler is invoked at the following scheduling events: task arrival, task departure, resource request, resource release, and the expiration of a time constraint. Thus, a reasonable heuristic is to use a "greedy" strategy, which means selecting a thread and the threads that it is dependent on (i.e., its predecessors), whose execution will yield the maximum PUD over others.

To deal with an arbitrarily shaped TUF, our philosophy is to regard it as a user-specified "black box" in the following sense: The black box (or the function) simply accepts a thread completion time and returns a numerical utility value. Thus, we ignore the information regarding the specific shape of TUFs in constructing schedules.

Therefore, to compute the PUD of a task T_i at time t, the algorithm considers the expected completion time of T_i (denoted as t_f), and the expected finishing times of T_i's predecessors as well. For each task T_j in T_i's dependency chain that needs to be completed before executing T_i, its expected finishing time is denoted as t_j. PUD of task T_i is calculated as $U_{total}/(t_f - t)$, where the expected utility $U_{total} = U_i(t_f) + \sum_{T_j \in T_i.Dep} U_j(t_j)$.

GUS does not mimic a deadline-based scheduling algorithm such as EDF, unlike many overload scheduling algorithms such as <u>D</u>ependent <u>A</u>ctivity <u>S</u>cheduling <u>A</u>lgorithm (referred to as DASA here) [5], who mimics EDF to reap its optimality during under-loads. This is because, for a task model with arbitrarily shaped TUFs, the deadline of a thread (with an associated TUF) may neither specify its timing urgency[4] nor its relative importance with respect to other threads. Thus, an "optimal" schedule—one that accrues the maximal possible utility—may not be directly related to the thread

[3] In [5], this metric was called <u>P</u>*otential* <u>V</u>*alue* <u>D</u>*ensity* (or PVD) then.

[4] "Urgency" generally means how much time left for a task to be completed in a timely manner.

deadlines. Furthermore, for non-step TUFs, the notion of an under-load situation in terms of timeliness feasibility does not make sense, as threads can yield different timeliness utility depending upon their completion times.

GUS can be used as a dispatching and a scheduling algorithm. The purpose of a dispatching algorithm is to determine the next task to run whereas a scheduling algorithm concerns about a complete schedule. If used as a dispatching algorithm, GUS outputs the next task to run without producing a complete schedule.

V. Preliminaries: State Components and Deadlocks

This section first introduces the state components of the algorithm and a set of auxiliary functions used in the description of GUS. We then discuss GUS' deadlock handling mechanism in the subsection that follows.

A. State Components and Auxiliary Functions

The following state components are used to facilitate the algorithm description.

1) Resource requests and assignments

   Each resource in the system is associated with an integer number, denoted as $ResourceId$. It serves as the identifier of the resource and is used by the scheduler and application threads. For each resource R, $R.Owner$ denotes the thread that is currently holding it. If R is not held by any thread (i.e., is free), $R.Owner = \emptyset$. Function `Owner(`R`)` returns the task that is currently holding the resource R.

   A request for a resource is a triple, called $ResourceElement$ that is defined as $\langle ResourceId, HoldTime, AbortTime \rangle$, where $ResourceId$ refers to the identifier of the requested resource; $HoldTime$ is the time for holding the resource; and $AbortTime$ is the time for releasing the resource by abortion. The $ResourceElement$ triple can also apply to the resource that is currently held by a thread. In that case, $ResourceId$ is the identifier of the resource that is being held and $HoldTime$ is the remaining holding time for the resource.

   Let function `holdTime(`T, R`)` return the holding time that is desired for a resource R by a thread T. Similarly, function `abortTime(`T, R`)` returns the time that is needed to release the resource R by aborting the thread T holding R.

2) State components of threads

The current execution mode of a thread is denoted by $Mode \in \{\texttt{NORMAL}, \texttt{ABORT}\}$ (see Section III). $ExecTime$ denotes the *currently remaining* execution time of a thread. Recall that we assume that a thread will release all resources it acquires before it ends. Thus, it follows that for any resource R held by a thread T, `holdTime(`T, R`)` $\leq T.ExecTime$.

$AbortTime$ denotes the *currently remaining* time to abort a thread. As discussed previously, $AbortTime$ is always associated with shared resources. Thus, whenever a thread acquires a shared resource, which is requested as $\langle R, HoldTime, AbortTime\rangle$, the thread's $AbortTime$ is increased. Furthermore, we assume that resources are released in the reverse order that they are acquired if the owner thread is aborted. [5]

$ReqResource$ is a $ResourceElement$ triple that describes the resource requested by a thread. Note that our resource request model does not allow multiple resources to be requested as part of a single resource request. Thus, for any thread, there is only one $ReqResource$ component. A thread not requesting any resource is described as $ReqResource = \langle\emptyset, \emptyset, \emptyset\rangle$. We use the function `reqResource(`T`)` to denote the identifier of the resource that is currently requested by a thread T.

$HeldResource = \{\langle R_i, HoldTime_i, AbortTime_i\rangle\}$ denotes the set of resources that is currently held by a thread, meaning zero or more resources are held by the thread.

3) The schedule

The output of the scheduling algorithm is an ordered sequence of triples, called a "schedule." A schedule consists of zero or more triples of $SchedElement = \langle ThreadId, Mode, Time\rangle$, where $ThreadId$ is the identifier of a thread; $Mode$ is the execution mode of the thread (either `NORMAL` or `ABORT`); and $Time$ is the CPU time allocated to the thread for the current execution.

It is possible that one thread appears at several positions within a schedule. This is because the scheduler may decide to execute a thread just long enough (either in `NORMAL` mode or in `ABORT` mode) to release the resource requested by other threads.

[5] POSIX specification requires maintaining a stack of cleanup handlers for each thread. These cleanup handlers are pushed into the stack by invoking `pthread_cleanup_push()` and can be popped out by using `pthread_cleanup_pop()`.

The remaining portion of that thread may be scheduled to execute later.

B. Deadlock Handling

The deadlock handling mechanism is invoked upon a scheduling event and before the GUS algorithm is executed. 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. Furthermore, some resource access protocols make assumptions on the resource requirements. For example, the Priority Ceiling Protocol [8] assumes the highest priority of the threads that will access a resource, called "ceiling" of the resource, is known. Likewise, the Stack Resource Policy [9] assumes "preemptive levels" of threads *a priori*. Such requirements or assumptions, in general, are not practical, due to the dynamic nature of the real-time applications that we are focusing in this paper.

There can be different strategies for deadlock detection and resolution. We present one such mechanism in Algorithm V.1, which considers the loss of utility.

For a single-unit resource request model, the presence of a cycle in the resource graph is the necessary *and* sufficient condition for a deadlock. Thus, the complexity of detecting a deadlock can be mitigated by a straightforward cycle-detection algorithm. For a system with n tasks, the worst case complexity of detecting a deadlock is $O(n)$.

The deadlock handling mechanism is therefore invoked by the scheduler 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 resource graph 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 resource graph.

However, the main difficulty here is to determine a thread to abort such that the loss of utility resulting from the abortion is minimized. Our strategy for this follows: For any thread T_j that lies on a cycle in the resource graph, we compute the utility that the thread can potentially accrue by itself if it were to continue its execution. If the thread T_j were to be aborted, then that amount of utility is lost. Thus, the loss of

```
input     : requesting task T_i; the current time t;
/* deadlock detection */
Deadlock := false;
T_j := Owner(reqResource(T_i));
while T_j ≠ ∅ do
    if T_j.Mode = NORMAL then
        T_j.LossPUD := U_j (t + T_j.ExecTime) /T_j.ExecTime;
    else
        Deadlock := false;
        break;
    if T_j = T_i then
        Deadlock := true;
        break;
    else
        T_j := Owner(reqResource(T_j));
/* deadlock resolution if any */
if Deadlock = true then
    abort(The minimal LossPUD task T_k in the cycle);
```

Algorithm V.1: Deadlock Detection and Resolution in GUS

utility per unit time by aborting a thread T_j called $LossPUD$, can be determined as $U_j\left(t + T_j.ExecTime\right)/T_j.ExecTime$. Once the $LossPUD$'s of all threads that lie on the cycle are computed, the algorithm then aborts the thread whose abortion will result in the smallest loss of utility. Note that here $LossPUD$ cannot be calculated by taking into account T_j's predecessors in the graph, since T_j lies on a cycle.

VI. The GUS Algorithm

The scheduling events of GUS 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.

A description of GUS at a high level of abstraction is shown in Algorithm VI.1. The algorithm accepts an unordered task list and produces a schedule. The format of the GUS schedule differs a simple ordered list of thread (or task) identifiers in the following two ways: (1) any given thread must execute in a certain mode, either `NORMAL` or `ABORT`; and (2) a thread may be split into several segments within the same schedule, where each segment executes for some designated time $Time$.

When the GUS scheduler is invoked at time t_{cur}, it first builds the chain of dependencies for each task (line 5-6). Then, each task's PUD is computed (line 7-8) by considering the task and all tasks in its dependency chain, called a $PartialSchedule$. Note that

$T_i.TotalTime$ and $T_i.TotalUtility$ (line 7) are the total execution time and the utility of T_i's partial schedule, respectively. $createPartialSched()$ returns a triple. The first element of the triple is the partial schedule, the second element is total execution of the partial schedule, and the third element is the utility accrued by executing the partial schedule. Finally, the maximum PUD task and its dependencies are added into the current schedule (line 9-12), if they can produce a positive utility.

1: **input** : An unordered task list UT;
2: **output** : An ordered schedule $Sched$;
3: *Initialization:* $t := t_{cur}$, $Sched := \emptyset$;
4: **while** $UT \neq \emptyset$ **do**
5:     **for** $\forall T_i \in UT$ **do**
6:         Build dependency list of T_i: $T_i.Dep :=$ `buildDep(`T_i`)`;
7:         $\langle T_i.PartialSched, T_i.TotalTime, T_i.TotalUtility \rangle :=$ `createPartialSched(`$T_i, T_i.Dep$`)`;
8:         $T_i.PUD := \dfrac{T_i.TotalUtility}{T_i.TotalTime}$;
9:     Pick the largest PUD task T_k among all tasks left in UT;
10:     **if** $T_k.PUD > 0$ **then**
11:         $Sched := Sched \cdot T_k.PartialSched$;
12:         $UT :=$ `delPartialSched(`$UT, T_k.PartialSched$`)`;
13:         $t := t + T_k.TotalTime$;
14:     **else**
15:         **break**;
16: **return** $Sched$;

Algorithm VI.1: A High-level Description of the GUS Algorithm

Note that a partial schedule is appended to the existing schedule (Algorithm VI.1, line 11), instead of being inserted. This is because of the way we compute the PUD for each task, where we assume that the tasks are executed at the current position in the schedule. If the selected partial schedule is inserted into the existing schedule, the previously computed PUDs become void. Furthermore, the algorithm does not consider the deadline order, due to the reasons we discussed in Section IV.

Once a partial schedule is appended to the schedule, GUS updates the time t, which is the starting time of the next partial schedule if there exists one. We call this time variable t as *virtual time* in the rest of the paper, because it denotes time in the future. GUS repeats the procedure until either it exhausts the unordered list, or no tasks can produce any positive utility.

Note that in line 9, GUS picks the task with the maximum PUD. Since this dispatching decision affects the completion time (and thus, utility) of all the other tasks in the system, we also consider the approach of checking the k largest PUD tasks to make the dispatching decision. We permute the k largest PUD tasks to find *their* best order, and append the k tasks at the end of the output schedule. Our goal is to minimize the utility loss for the other (delayed) tasks. We call this additional heuristic as k-step PUDs, and picking the largest PUD task becomes a special case of 1-step PUD. The heuristic of k-step PUDs ($k > 1$) yield no better performance than 1-step PUD. This is elaborated and evaluated in Section X.

We discuss details of creating and deleting partial schedules in Section VI-A. A sub-problem of creating a partial schedule is to determine the execution mode of tasks in the dependency chain, and it is addressed in Section VI-B.

A. Manipulating Partial Schedules

A partial schedule is part of the complete schedule, containing a sequence of *SchedElement*'s. We use $T_i.PartialSched$ to denote the partial schedule that is computed for task T_i. $T_i.PartialSched$ consists of task T_i, and all of, or portions of T_i's predecessors. We show how GUS computes the partial schedule for T_i in Algorithm VI.3.

Before GUS computes task partial schedules, the dependency chain of each task must be determined. This procedure is shown in Algorithm VI.2. The algorithm simply follows the chain of resource request/ownership. Each task T_j in the dependency list has a successor task that needs a resource that is currently held by T_j. A successor cannot be scheduled to execute until its predecessor completes. For convenience, the input task T_i is also included in its own dependency list, so that all other tasks in the list have a successor task. The `buildDep` algorithm stops either because a predecessor task 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 the farthest predecessor of T_i (which can be retrieved by the function `Head(`$T_i.Dep$`)` and ends with T_i itself.

The `createPartialSched()` algorithm accepts a task T_i, its dependency list $T_i.Dep$, and a virtual time t. The virtual time t is the time to execute the partial schedule to be created. On completion, the `createPartialSched()` algorithm produces a partial schedule for T_i, the total execution time of the partial schedule called $TotalTime$, and

input : task T_i;
output : dependency list of T_i: $T_i.Dep$;
Initialization : $T_i.Dep := T_i$;
$PrevT := T_i$;
while `reqResource(PrevT)` $\neq \emptyset \bigwedge$ `Owner(reqResource(PrevT))` $\neq \emptyset$ **do**
    $T_i.Dep :=$`Owner(reqResource(PrevT))` $\cdot T_i.Dep$;
    $PrevT :=$ `Owner(reqResource(PrevT))`;

Algorithm VI.2: `buildDep()`: Build Dependency List

the aggregate utility that can be obtained by executing the partial schedule at time t, called $TotalUtility$. The algorithm computes the partial schedule by assuming that tasks 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 task T_i and its predecessors consists of two parts: (1) the time needed to release the resources that are needed to execute T_i; and (2) the remaining execution time of T_i itself. The order of executing the corresponding tasks or portions of tasks, in their particular modes, together becomes the partial schedule.

Lines 4-14 of Algorithm VI.3 compute the time for T_i to acquire its requested resources. Lines 15-21 account for the remaining execution time of T_i itself. Note that, to release a resource R, a task T_j can either execute in `NORMAL` mode for `holdTime(`T_j, R`)` or in `ABORT` mode for `abortTime(`T_j, R`)`. These two alternatives are accounted for in lines 6-14 of the algorithm by calling the algorithm `determineMode()`.

Since our application model requires explicit release of resources before the end of a thread, it is possible that a task is selected to execute for only a portion of its remaining execution time, after which one or more of the resources that it holds are released. The remaining portion of that task may be scheduled to execute later.

If a task T_j is scheduled to complete it's execution after it releases resources that are needed by its successor, then T_j may accrue some utility. This is accounted for in lines 10-11. Finally, task T_i itself may be executed in either `NORMAL` or `ABORT` mode, which has been determined before the current scheduling event. If T_i is executing in `NORMAL` mode, naturally, it may accrue some positive utility (line 18). Otherwise, no utility can be accrued from the execution of T_i.

If the selected partial schedule contains the remaining portion of a task T, either in `NORMAL` mode or in `ABORT` mode, task T needs to be removed from the unordered task

input : task T_i and its dependency list $T_i.Dep$; t: the time to start executing the partial schedule;
output : a partial schedule $PartialSched$; the total execution time of $PartialSched$, called $TotalTime$; the total utility accrued by executing $PartialSched$, called $TotalUtility$;
Initialization: $PartialSched := \emptyset$; $TotalTime := 0$; $TotalUtility := 0$;
`/* consider tasks in` T_i`'s dependency chain */`
for $\forall T_j \in T_i.Dep \bigwedge T_j \neq T_i$, *starting from the immediate dependency task* **do**
    $R :=$ `reqResource(`$T_j \rightarrow Next$`)`;
    $Mode :=$`determineMode(`$T_j, TotalUtility, TotalTime, t$`)`;
    if $Mode =$ *NORMAL* **then**
        $PartialSched := PartialSched \cdot \langle T_j,$ `NORMAL`, `holdTime(`T_j, R`)`$\rangle$;
        $TotalTime := TotalTime +$ `holdTime(`T_j, R`)`;
        if *holdTime*$(T_j, R) = T_j.ExecTime$ **then**
            $TotalUtility := TotalUtility + U_j\left(t + TotalTime\right)$;
    else
        $PartialSched := PartialSched \cdot \langle T_j,$ `ABORT`, `abortTime(`T_j, R`)`$\rangle$;
        $TotalTime := TotalTime +$ `abortTime(`T_j, R`)`;
`/* consider` T_i `itself */`
if $T_i.Mode =$ *NORMAL* **then**
    $PartialSched := PartialSched \cdot \langle T_i,$ `NORMAL`$, T_i.ExecTime\rangle$;
    $TotalTime := TotalTime + T_i.ExecTime$;
    $TotalUtility := TotalUtility + U_i\left(t + TotalTime\right)$;
else
    $PartialSched := PartialSched \cdot \langle T_i,$ `ABORT`$, T_i.AbortTime\rangle$;
    $TotalTime := TotalTime + T_i.AbortTime$;
return $\langle PartialSched, TotalTime, TotalUtility\rangle$

Algorithm VI.3: The `createPartialSched()` Algorithm

list UT. Consequently, if the selected partial schedule only contains a portion of task T's remaining part, state components of T need to be updated to reflect this.

The GUS algorithm uses another algorithm called `delPartialSched` to delete a partial schedule from an unordered task list, as shown in Algorithm VI.4. The `delPartialSched` algorithm examines the partial schedule, from the head to the tail. If a task T has been determined to execute in `NORMAL` mode, then its remaining execution time is updated (line 11). Moreover, T may release one or more resources during the allocated time. Therefore, the $HoldTime$'s of the resources that are currently held by T are also updated (lines 6-10). In the event that T is selected to complete its execution such that it can release the resources, then T is completely removed from RUT (lines 12-13). Furthermore, we consider the *abortion* of a task as the execution of a different piece of code segment for the task. Thus, the same procedure applies to those tasks that have been determined to execute in the `ABORT` mode (lines 15-22).

If a task T is not the tail of a partial schedule, it must release at least one resource. This is because, the only reason for executing the task T, either in `NORMAL` mode or in ABORT mode, is to release the requested resource, so that the successor of task T is able to execute. However, task T_i (recall that the partial schedule is due to task T_i)must complete in the partial schedule. Therefore, it is completely removed from RUT (line 23).

```
1:  input   : a partial schedule Ti.PartialSched and an unordered task list UT;
2:  output  : a reduced task list RUT;
3:  Copy UT into RUT: RUT := UT;
4:  for ∀ ⟨T, Mode, Time⟩ ∈ PartialSched ⋀ T ≠ Ti from head to tail do
5:      if Mode = NORMAL then
6:          for ∀ ⟨R, HoldTime, AbortTime⟩ ∈ T.HeldResources do
7:              Update HoldTime: HoldTime := HoldTime − Time;
8:              if HoldTime := 0 then
9:                  T.HeldResource := T.HeldResource − ⟨R, HoldTime, AbortTime⟩;
10:                 R.Owner := ∅;
11:         Update T.ExecTime, T ∈ RUT: T.ExecTime := T.ExecTime − Time;
12:         if T.ExecTime = 0 then
13:             Remove T from RUT : RUT := RUT − T;
14:     else
15:         for ∀ ⟨R, HoldTime, AbortTime⟩ ∈ T.HeldResources do
16:             Update AbortTime : AbortTime := AbortTime − Time;
17:             if AbortTime = 0 then
18:                 T.HeldResource := T.HeldResource − ⟨R, HoldTime, AbortTime⟩;
19:                 R.Owner := ∅;
20:         Update T.AbortTime, T ∈ RUT: T.AbortTime := T.AbortTime − Time;
21:         if T.AbortTime := 0 then
22:             Remove T from RUT : RUT = RUT − T;
23: Remove Ti from RUT;
24: return RUT;
```

Algorithm VI.4: Removing a Partial Schedule from a Task List delPartialSched()

B. Determining Task Execution Mode

Besides resolving deadlocks, abortions can also be used to improve the aggregate utility. The intuition for doing so is that the time to abort a task may be different from the normal resource hold time, which in turn may affect the timeliness of the tasks that depend upon it. Thus, determining the execution mode of the tasks in a dependency list is necessary to achieve better performance.

Algorithm VI.5 determines the execution mode of a task T_j in the dependency list of task T_i. In case that task T_j is running in ABORT mode, the determineMode() algorithm immediately returns ABORT mode (lines 3). If a thread cannot be aborted (i.e., $AbortTime$ is infinity), the algorithm immediately returns NORMAL mode (line 4).

In other cases, to determine which execution mode is better, the algorithm compares the PUD's of T_i, when T_j is executed in the two modes—`NORMAL` mode (lines 5-6) and `ABORT` mode (lines 7-8). The required time and accrued utility are computed under the two execution modes (called $NormalTime$ and $NormalUtility$ for the first scenario; $AbortTime$ and $AbortUtility$ for the second scenario). The algorithm then follows the dependency chain to examine T_j's successors in $T_i.Dep$ except T_i, assuming that all of them execute normally (lines 5-20) if they are not currently in `ABORT` mode. This assumption is reasonable even though the execution modes of those tasks have not yet been determined when T_j is examined, because a task's initial mode is `NORMAL`, unless changed by the scheduler.

Finally, the algorithm considers task T_i itself (lines 21-28), whose mode has been determined before the current scheduling event. If T_i is in `NORMAL` mode, it requires $T_i.ExecTime$ to finish the execution of the task. This may or may not produce some positive utility. On the other hand, if T_i is being aborted, it needs $T_i.AbortTime$ to complete the abortion. However, this will not produce any utility.

Once total execution times and total utilities under the two scenarios are computed, the algorithm computes the two different PUD's. If executing T_j in `NORMAL` mode will yield a higher PUD for T_i, then the algorithm decides to execute T_j normally (lines 29-30). Otherwise, T_j is aborted (lines 31-32).

Note that our approach to determine a task execution mode is different from that of the DASA algorithm [5]. DASA seeks to acquire the requested resource as soon as possible. Thus, if the abort time of a task is shorter than its execution time, the task is aborted. Otherwise, the task executes normally. This simple criterion works well for step time/utility functions, because the timeliness of a task will not be negatively affected if the task finishes earlier than its deadline. However, this is not true for arbitrarily shaped time/utility functions. In fact, for non-increasing time/utility functions, the GUS `determineMode()` algorithm can simply return the `NORMAL` mode if a task's abort time is longer than its execution time; otherwise it can return the `ABORT` mode.

```
input   : task T_j ∈ T_i.Dep, the current accumulative utility of T_i, TotalUtility; the current
          accumulative execution time of T_i, TotalTime, the current virtual time t;
output  : the execution mode of T_j;
/* T_j is currently running in ABORT mode */
if T_j.Mode = ABORT then        return ABORT;
 /* T_j is not abortable */
if T_j.AbortTime = ∞ then       return NORMAL;
 /* scenario I: assuming T_j executes normally */
NormalTime := TotalTime+ holdTime(T_j, reqResource(T_j → Next));
NormalUtility := TotalUtility + U_j (t + NormalTime);
 /* scenario II: assuming T_j aborts execution */
AbortTime := TotalTime+ abortTime(T_j, reqResource(T_j → Next));
AbortUtility := TotalUtility;
NextT := T_j → Next;
while NextT ≠ ∅ do
    /* consider tasks in T_i's dependencies */
    if NextT.Mode = NORMAL then
        NormalTime := NormalTime+ holdTime(NextT, reqResource(NextT → Next));
        NormalUtility := NormalUtility + U_NextT (t + NormalTime);
        AbortTime := AbortTime+ holdTime(NextT, reqResource(NextT → Next));
        AbortUtility := AbortUtility + U_NextT (t + AbortTime);
        NextT := NextT → Next;
    else
        NormalTime := NormalTime+ abortTime(NextT, reqResource(NextT → Next));
        AbortTime := AbortTime+ abortTime(NextT, reqResource(NextT → Next));
        NextT := NextT → Next;

/* consider T_i itself */
if T_i.Mode = NORMAL then
    NormalTime := NormalTime + T_i.ExecTime;
    NormalUtility := NormalUtility + U_i (t + NormalTime);
    AbortTime := AbortTime + T_i.ExecTime;
    AbortUtility := AbortTime + U_i (t + AbortTime);
else
    NormalTime := NormalTime + T_i.AbortTime;
    AbortTime := AbortTime + T_i.AbortTime;
/* determine the execution mode of T_j */
if NormalUtility/NormalTime ≥ AbortUtility/AbortTime then
    Mode := NORMAL;
else
    Mode := ABORT;
return Mode;
```

Algorithm VI.5: `determineMode()`: Determining Task Execution Mode

C. Handling Termination Time Exceptions

Each TUF U_i has a termination time TM_i (see Section III). If the termination time is reached and thread T_i has not finished execution or even not yet started execution, an exception should be raised. Normally, this exception causes abortion of T_i, which implies execution of the thread's cleanup handlers, if any.

In this paper, we assume that an handler for a termination time exception is associated with an application-specific time constraint—i.e., a time/utility function. Thus, the termination handler is scheduled in the same way as other threads. In fact, execution of the exception handler is simply part of the thread itself.

VII. Complexity of GUS

To analyze the complexity of the GUS algorithm (Algorithm VI.1), we consider n tasks and a maximum of r resources in the system. Observe that, in the worst case, each task may hold up to r resources and may be split into $(2r+1)$ partial schedules. Thus, the *while*-loop starting at line 4 in Algorithm VI.1 may be repeated $O(nr)$ times. Each execution of the loop body examines up to n tasks (or portions of the tasks) that remain in the unordered task list. Clearly, complexity of the *while*-loop body is dominated by the complexity of creating a partial schedule (Algorithm VI.1, line 7), which in turn is dominated by the cost of determining execution modes of up to n tasks in the dependency chain. Since Algorithm VI.5 costs $O(n)$ for each task, the worst-case complexity of Algorithm VI.3 is $n \times O(n) = O(n^2)$. Therefore, the worst-case complexity of the GUS algorithm is $nr \times (n \times O(n^2)) = O(n^4 r)$.

Note that dispatching using the GUS algorithm is sufficient, unlike most other scheduling algorithms. This is because, a new partial schedule is appended by GUS only at the tail of the existing schedule, and cannot affect the partial schedule at the head (of the existing schedule) by any means. Thus, for GUS to work as a *dispatching* algorithm instead of a *scheduling* algorithm, the *while*-loop starting at line 4 in Algorithm VI.1 can be eliminated. In this case, the complexity of GUS as a dispatching algorithm can be reduced to $O(n^3)$.

VIII. Non-Timeliness Properties of GUS

This section presents a class of non-timeliness properties of GUS, including resource safety and task execution mode assurance.

Property VIII.1. *A task within a partial schedule is either ready to execute, or becomes ready after its predecessor task within the same partial schedule executes. We call a partial schedule "self-contained."*

Proof. This property can be shown to be true by examining the `createPartialSched` algorithm (Algorithm VI.3). Consider a task T_i within a partial schedule PS. If T_i lies at the head of PS, then it must also lie at the head of the dependency chain. Thus, it is ready to execute and the resource that it needs is available.

If T_i does not lie at the head of PS, then let T_j be the immediate predecessor task of T_i in PS. Let R be the resource requested by T_i. If T_j is added into the partial schedule in `NORMAL` mode, then it must execute for `holdTime(`T_j, R`)` time units (line 8). On the other hand, if T_j is selected to execute in `ABORT` mode, then T_j is aborted after `abortTime(`T_j, R`)` (line 13) time units, releasing resource R. Thus, resource R must be free when T_i is scheduled to execute. □

Property VIII.2. *A complete schedule is self-contained if every partial schedule within it is self-contained.*

Proof. We consider a task T within a complete schedule. Since a complete schedule is a concatenation of a set of partial schedules, T must also belong to a partial schedule PS. By Property VIII.1, this property is therefore true. □

Property VIII.3. *When a task T_i that requests a resource R is selected for execution by GUS, the resource R will be free at the time of execution of T_i.*

Proof. The property can be proved using Property VIII.1 and Property VIII.2. □

Property VIII.4. *If a task T_i is executing in `ABORT` mode, GUS will not later schedule T_i to execute in `NORMAL` mode.*

Proof. This property is a natural requirement, because our task model assumes that an aborted task cannot execute in `NORMAL` mode in the future, although the abort operation may be split into several stages. The correctness of this property can be seen from lines 3-3 of the `determineMode()` algorithm (Algorithm VI.5). This mode information is further used in the `createPartialSched()` algorithm. If a task is executing in `ABORT` mode, the algorithm will append the task in `ABORT` mode at the tail of the partial schedule (lines 12-14 and lines 19-21). □

Recall that a non-abortable thread is associated with infinite abortion time. Thus, by similar argument (see line 4-4 of Algorithm VI.5), we can prove the following property

Property VIII.5. *If a task T_i is not abortable, GUS will not schedule it to execute in*

`ABORT` *mode.*

IX. Simulation Results

We performed two sets of experiments. The first set of experiments were "static" simulations in the sense that each experiment examines a task ready queue at a particular scheduling event and produces a schedule. The major advantage of conducting such static simulations is that we can compare the schedule produced by GUS with that obtained by exhaustive search. The second set of experiments were "dynamic" simulations, as we randomly generated streams of tasks at run time and measured the aggregate utility produced by various scheduling algorithms.

For performance comparison, we considered DASA [5], LBESA [6], CMA (named after the authors of the algorithm, i.e., Chen and Muhlethaler's Algorithm) [7], and D^{over} [10] algorithms. Each input to a static simulation experiment is a randomly generated 9-task set with certain distributions, including uniform distribution, normal distribution, and exponential distribution. Moreover, we normalize the accrued utility with that acquired by the schedule that is produced by exhaustive search. Note that the exhaustive search knows the resource request patterns in advance, but unlike GUS, it does not split a thread into several segments within the same schedule. We call such a performance metric as *normalized accrued utility* or "normalized AU" for short. Furthermore, a single data point was obtained as the average of 500 independent experiments.

TABLE I: Simulation Parameters

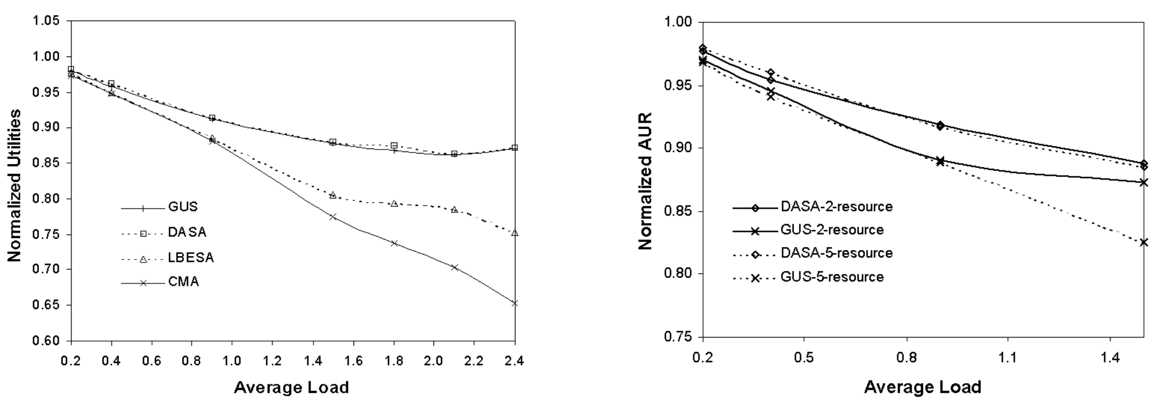
Distribution	ExecTime	Static Deadline	InterArr Time	Laxity
uniform	U[0.05, $2C_{avg}$]	U[0.01, $2D_{avg}$]	U[0.01, $2C_{avg}/\rho$]	N[0.25, 0.25]
normal	N[C_{avg}, C_{avg}]	N[D_{avg}, D_{avg}]	N[C_{avg}/ρ, 0.5]	N[0.25, 0.25]
exponential	E[C_{avg}]	E[D_{avg}]	E[C_{avg}/ρ]	N[0.25, 0.25]

Let C_{avg} be the average task execution time and let D_{avg} be the average task "deadline" that is defined as the latest time point after which the task utility is always zero [6]. Given a 9-task set, the average load ρ can be determined as $\rho = 9C_{Avg}/D_{avg}$. In all the simulation

[6]We assume that a deadline point is the same as the termination time of a TUF. This deadline definition is used through all our experimental results presented in Sections IX and X.

experiments, we chose C_{avg} as 0.5 sec. In Table I, we show the simulation parameters [7], where parameters for the static simulations are shown in columns 2 and 3 of the table.

Excluding the D^{over} algorithm, our first static simulation experiments involve all other four scheduling algorithms for independent tasks with step time/utility functions. The maximum values of the TUFs abide normal distribution N[10, 10]. The D^{over} algorithm requires a timer for *Latest Start Time* [10]. Thus, it can only be dynamically simulated. In Figure 6(a), we show the performance of the algorithms under uniform distributions. As shown in the figure, DASA and GUS have very close performance for the entire load range and perform the best. On the other hand, CMA and LBESA exhibits the worst performance among all algorithms. Our experiments with normal and exponential distributions showed consistent results. Thus, we conclude that GUS, in general, has close performance to DASA for independent task sets with step TUFs.



(a) Without Dependencies

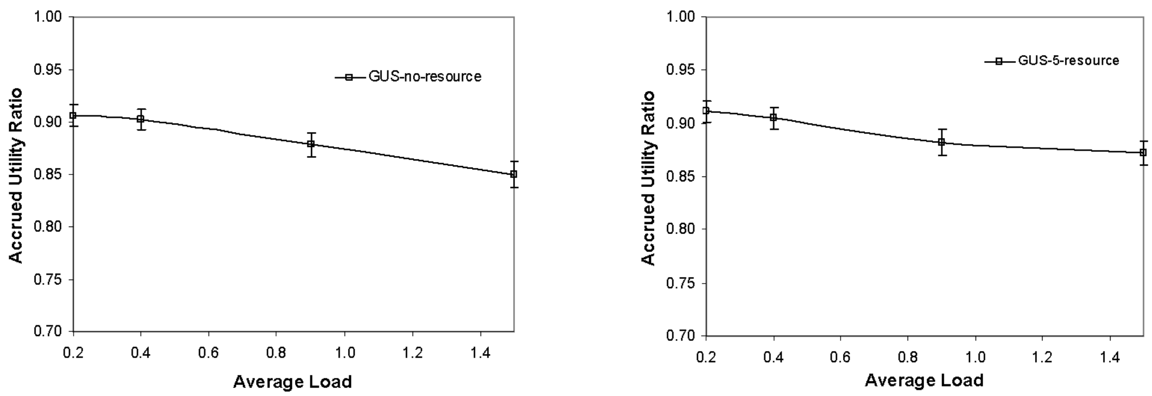
(b) With Dependencies

Fig. 6: Normalized Performance of Algorithms Under Step TUFs and Uniform Distributions

For dependent task sets in the static simulation, we randomly generate resource dependency graphs. Once a generated graph is verified to be deadlock-free, it is sent as the input of the simulator. We show the simulation results of DASA and GUS schedulers for uniform distributions in Figure 6(b), as an example. Observe that GUS performs worse than DASA in the figure. Furthermore, the performance gap increases when more resources are present in the system. We attribute this worse performance of GUS to the

[7] U$[a, b]$ denotes a random variable that is uniformly distributed between a and b. N$[a, b]$ specifies a normal distribution with mean value of a and variance of b. E$[a]$ is an exponential distribution with mean value of a.

fact that GUS ignores tasks deadlines in making scheduling decisions. Though deadline order may not be appropriate for arbitrarily shaped TUFs (see Section IV), it can be beneficial for tasks with step TUFs. On the other hand, DASA only outperforms GUS by no more than 10% in terms of normalized AUR, even if the system is overloaded and has five shared resources.



(a) Without Dependencies

(b) With Dependencies

Fig. 7: Performance of GUS Under Arbitrarily Shaped TUFs and Uniform Distributions

In Figures 7(a) and 7(b), we show the performance of GUS for tasks with arbitrarily shaped TUFs, which are represented by 3^{rd}-order polynomials in the experiments. [8] The coefficients of the 3^{rd}-order polynomials are randomly generated between [-10, 10], which we believe can provide good approximation of "arbitrary" TUFs. As shown in the two figures, GUS does not suffer abrupt performance degradation with or without shared resources. Furthermore, if the system is not heavily overloaded, GUS can accrue most of the utility (over 80%) that can be accrued by the exhaustive-search scheduler. Simulation results for normal and exponential distributions are similar.

Our major objective in conducting dynamic simulations is to compare the performance of GUS with the D^{over} algorithm, which cannot be investigated in static simulations. Each dynamic simulation experiment generates a stream of 1000 tasks.

Given the task average execution time C_{avg} and a load factor ρ, the average task inter arrival time can be calculated as C_{avg}/ρ. This average task inter arrival is applied for

[8]The error bar around each data point represents the 90% confidence interval of that data point, and each single data point is an average of 500 independent experimental results.

different distributions, as shown in column 4 of Table I. Furthermore, the task laxity is modeled as a random variable with normal distribution N[0.25, 0.25] (Table I, column 5). Note that Laxity is defined as $StaticDeadline - ExecTime$. To compare the algorithm performance, we consider independent tasks with step TUFs. Again, the maximum values of the TUFs abide normal distribution N[10, 10].

We show the performance of the five scheduling algorithms for step TUFs in terms of Accrued Utility Ratio (AUR) in Figure 8. AUR is defined as the accrued utility divided by the maximal utility that can possibly be accrued from the tasks. As shown in the figure, the D^{over} algorithm performs the worst among all six algorithms, while DASA, CMA, and GUS have close performance. The poor performance of D^{over} is because the algorithm rejects more high utility tasks than it should, to guarantee the worst-case performance (see Chapter 8, Section 8.4.2 in [17]). Furthermore, the optimality of D^{over} only applies to tasks whose utilities are proportional to their execution times, which is not the case in our experiments.

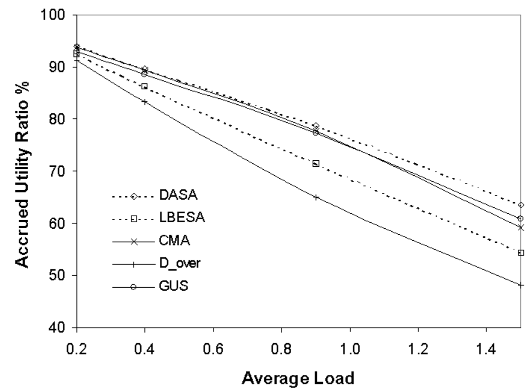


Fig. 8: Algorithm AURs: Step TUFs, No Dependencies, and Uniform Distributions

X. Implementation Results

We implemented GUS and several other scheduling algorithms (used in the simulation studies) in a scheduling framework called *meta-scheduler* [18]. The meta-scheduler is an application-level framework for implementing utility accrual scheduling algorithms on POSIX-compliant operating systems [16], without modifying the operating system. Our major motivation for considering the meta-scheduler framework (as opposed to using an OS kernel environment) is because, it is significantly easier to implement and debug scheduling algorithms in the meta-scheduler framework. In the meanwhile, the meta scheduler maintains reasonably small overhead (from 10 usec to a few hundred usec in a common hardware platform). The experimental system is a Toshiba Satellite 1805-S254 laptop (one 1GHz Intel Pentium III processor, 256KB L2 cache, and 256MB SDRAM) running QNX Neutrino 6.2.1 real-time operating system.

During each experiment, 100 tasks are generated with randomly distributed parameters such as task execution times and deadlines. For example, the worst-case execution times (WCETs) of tasks [9] are exponentially distributed with a mean of 500 msec. Given a workload ρ, we calculate the mean inter-arrival time as the mean execution time divided by ρ. In addition, the laxity of a task is uniformly distributed between 50 msec and 1 sec. The task relative deadline is then the sum of the task execution time and laxity.

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

We implemented four scheduling algorithms including DASA, LBESA, D^{over}, and GUS as part of our experimental study. The CMA algorithm that we had considered in the simulation studies was not implemented because it requires significant amount of memory space and CPU time for median number of ready tasks.

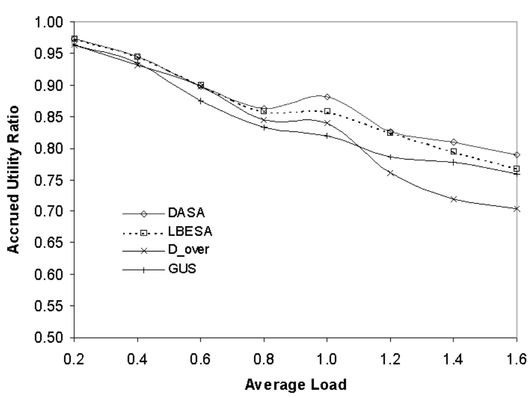


Fig. 9: Algorithm AURs: Step TUFs, No Dependencies, and Exponential Distributions

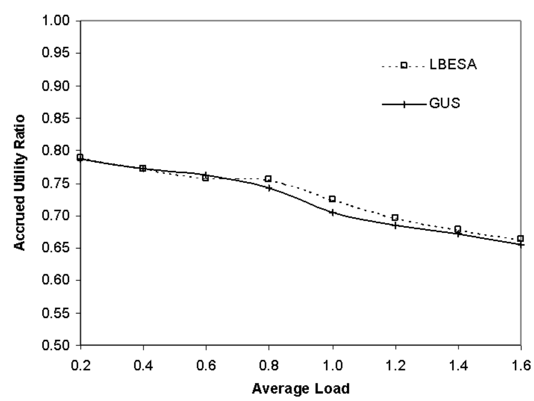


Fig. 10: Algorithm AURs: Arbitrary TUFs, No Dependencies, and Exponential Distributions

Since all algorithms with the exception of GUS, have certain restrictions, e.g., can only handle certain shapes of TUFS, we only compare performance of the algorithms for the

[9]WCETs are equal to exact execution times in our experiments.

cases that they apply to. For example, Figure 9 shows the AURs of all four scheduling algorithms for task sets with step TUFs but no dependencies. From the figure, we observe that the performance of the four algorithms do not significantly differ for light load and medium load conditions (workload is less than 0.8). However, DASA and LBESA show superior performance for heavy and overloaded situations. Figure 10 shows the AURs of GUS and LBESA for task sets with arbitrary TUFs but no dependencies. As shown in the figure, GUS performs very close to LBESA during both under-loads and overloads.

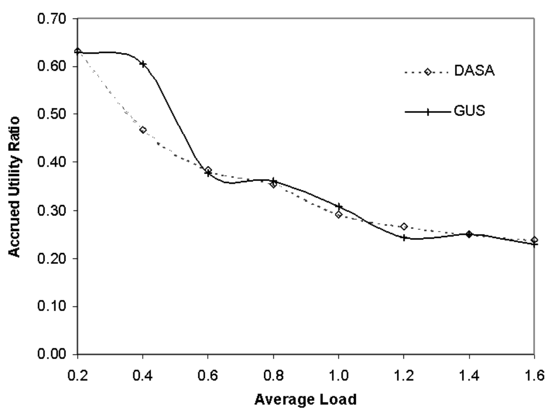


Fig. 11: Algorithm AURs: Step TUFs, With Dependencies, and Exponential Distributions

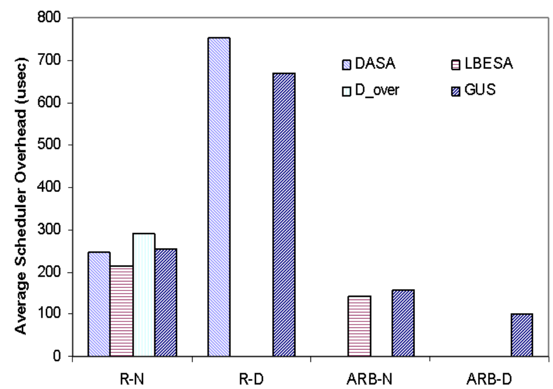


Fig. 12: Average Scheduling Overhead

In addition, we show the performance comparison of DASA and GUS for task sets with step TUFs and dependencies in Figure 11. Our experiments in this class used three shared resources as an example scenario. We would expect DASA to outperform GUS for this class of experiments, because GUS neither conducts a feasibility test, nor considers the deadline order for scheduling the feasible task subset. The figures, however, show that GUS actually performs better than DASA during light workload situations. Performance of the two schedulers are very close during overloaded situations. We conjecture that the better performance of GUS is because of its lower actual computational cost (for this particular implementation), in spite of its $O(n^3)$ complexity (DASA's complexity is $O(n^2)$). In Figure 12, we provide further evidence for our conjecture by measuring scheduling overheads. Note that "R-N" in the figure stands for step TUFs without dependencies; "R-D" is for step TUFs with dependencies. Similarly, "ARB-N" and "ARB-D" are for arbitrary TUFs without dependencies and with dependencies, respectively.

We also conduct experiments to evaluate the heuristic of k-step PUDs as described in Section VI. When $k = 1$, this heuristic is equal to that of GUS to pick the task with the largest PUD. The heuristic of k-step PUDs is applicable only at situations without resource dependencies among tasks, since only in this case the output tasks are ordered by non-increasing PUDs. We choose arbitrary TUFs and vary k from 1 to 5. Figure 13 shows the AURs of GUS with different values of k. From the figure, k-step PUDs ($k > 1$) yield no better performance than 1-step PUD. Specifically, during very low loads, k-step PUDs ($k > 1$) perform almost the same as 1-step PUD, because only very short task queues are generated during very low loads, and often the task queue length is less than k. This makes the k-step permutation meaningless. During high loads, k-step PUDs ($k > 1$) can check more possible orders of tasks, so as to minimize the utility loss for the other (delayed) tasks. But k-step PUDs require additional CPU overhead for the permutation operation, which imposes negative effects on the performance. Thus, during high loads, k-step PUDs ($k > 1$) perform no better than 1-step PUD; especially when $k = 5$, the additional overhead degrades the performance to be worse than the others. Therefore, the heuristic of GUS to pick the largest PUD task is both performance-effective and cost-effective.

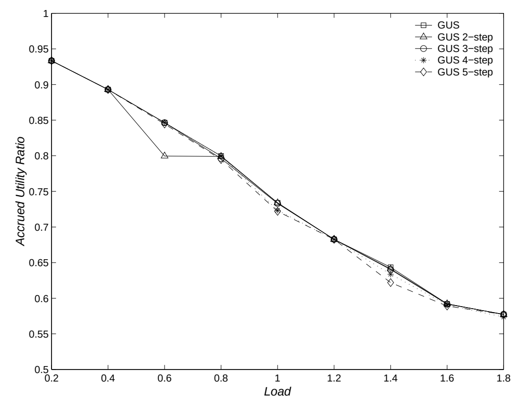


Fig. 13: Algorithm AURs: Comparison of k-step PUDs with Arbitrary TUFs and Uniform Distributions

XI. Related Work

Uni-processor real-time scheduling algorithms can be broadly classified into two categories: (1) deadline scheduling and (2) overload scheduling. Algorithms in the first category generally seek to satisfy all hard deadlines, if possible. Examples of deadline scheduling algorithms include the RMA algorithm [1] and the EDF scheduling algorithm [19]. These algorithms are extended and varied to deal with other deadline-based optimality criteria, such as the (m, k) *firm guarantee* presented in [20], lock-based real-time resource access using the Priority Inheritance Protocol [8], the Priority Ceiling Protocol [8], and the Stack Resource Policy [9]. In contrast, algorithms in the second category deal with deadlines as well as non deadline time constraints such as non-step TUFs, wherever proper.

Many existing algorithms for overload scheduling consider step TUFs, and mimic the behavior of EDF during under-loads as closely as possible. Furthermore, they seek to optimize other performance metrics during overloads, since all deadlines cannot be satisfied during overloads. One important metric is the sum of utility (or "value") that is accrued by all tasks. In [21], the authors show that, for restrictive task sets (i.e., task utilities are proportional to task execution times), the upper bound on the competitive factor of any on-line scheduling algorithm is $1/(1+\sqrt{k})^2$, where k is the *importance ratio* of the task set. This upper bound is achieved by the D^{over} algorithm [10]. However, the optimal competitive ratio does not imply the best performance for D^{over} for a broad range of workload we considered in the experiments.

Besides the optimal D^{over} algorithm, heuristic algorithms have also been developed for effective scheduling during overloaded situations. In [6], Locke presents the LBESA algorithm that uses the notion of value density, which is complemented with feasibility tests. Locke's work is extended by several others including [11], [22]. Performance of the algorithms presented in [11] may be better than LBESA's, but in general, is very close.

Apart from the step TUF model with one segment of execution per task, the concept of "imprecise computations" has also been proposed in the literature as an effective technique to handle overloads [23]. For example, the algorithm presented in [24] consider this model. Furthermore, the work on feedback control theory scheduling assumes the presence of N versions of the same task ($N \geq 2$) [25].

Our work fundamentally differs from all the aforementioned algorithms in that we consider arbitrarily-shaped time/utility functions *and* mutual exclusion resource constraints. All the previously mentioned algorithms, except for the LBESA algorithm, only consider step TUFs. Furthermore, LBESA does not consider mutual exclusion constraints.

In [26], the authors present the concept of "timeliness-functions." In [22], the same authors show that scheduling the task with the highest Dynamic Timeliness-Density (DTD) is more effective than scheduling the highest value density task. The DTD heuristic is echoed in a special case of GUS, where tasks do not share resources.

Non-increasing TUFs have been explored in the context of non-preemptive scheduling of independent activities, such as CMA [7]. We show the comparison of GUS with CMA in Section IX. Again, GUS allows preemption, arbitrary time/utility functions (including non-increasing functions), and mutual exclusion resource dependencies.

In [12], Strayer presents a framework for scheduling using "importance functions." An importance function can take arbitrary shapes, and has the similar meaning as a time/utility function. However, no new scheduling algorithms are presented in [12]. Furthermore, the task model considered in [12] do not consider resource dependencies.

In the context of overload scheduling, little work considers shared resources that have mutual exclusion constraints. DASA [5] considers shared resources with mutual exclusion constraints, but only for step time/utility functions. However, GUS allows arbitrarily shaped TUFs, whereas DASA is restricted to step functions. To the best of our knowledge, DASA and GUS are the only two algorithms that schedule both CPU cycles and other shared resources while allowing time constraints to be expressed using TUFs.

There are however, a significant number of algorithms that can simultaneously manage multiple shared resources (either multiple units of the same resource or multiple resources). An example algorithm is the Q-RAM model [27]. Similar to the imprecise computation, the Q-RAM model assumes that each task can be executed in a number of ways, where different executions require different amount of shared resources, but yield different utilities. Furthermore, it assumes that the utility of a task depends on the resources allocated to it, which is fundamentally different from our UA model. In Section II, we provide motivation for our TUF/UA model by summarizing two significant demonstration applications that were successfully implemented using our UA model.

XII. Conclusions and Future Work

Our simulation results and implementation measurements show that the GUS algorithm has comparable performance with algorithms such as DASA, LBESA, CMA and D^{over} for all the application scenarios that they apply to. Experiments reveal that the accrued utility ratio of GUS is within roughly 5% of that of any other algorithm if not better. However, GUS can handle task sets with arbitrarily shaped TUFs and mutual exclusion resource constraints; none of the existing algorithms can schedule such a task set. This is the major contribution of the GUS algorithm. Furthermore, we establish several fundamental timeliness and non-timeliness properties of GUS.

Several aspects of GUS are interesting directions for further study. One direction is to develop a stochastic version of GUS—one that considers task models with stochastically specified task properties including that for execution times. Another very interesting future direction is to extend GUS for distributed scheduling for satisfying end-to-end time constraints in real-time distributed systems.

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] "Multi-platform radar technology insertion program," http://www.globalsecurity.org/intell/systems/mp-rtip.htm/.

[3] "Bmc3i battle management, command, control, communications and intelligence," http://www.globalsecurity.org/space/systems/bmc3i.htm/.

[4] E. D. Jensen, C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1985, pp. 112–122.

[5] R. K. Clark, "Scheduling dependent real-time activities," Ph.D. dissertation, Carnegie Mellon University, 1990, CMU-CS-90-155.

[6] C. D. Locke, "Best-effort decision making for real-time scheduling," Ph.D. dissertation, Carnegie Mellon University, 1986, CMU-CS-86-134.

[7] K. Chen and P. Muhlethaler, "A scheduling algorithm for tasks described by time value function," *Journal of Real-Time Systems*, vol. 10, no. 3, pp. 293–312, 1996.

[8] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[9] T. P. Baker, "Stack-based scheduling of real-time processes," *Journal of Real-Time Systems*, vol. 3, no. 1, pp. 67–99, March 1991.

[10] G. Koren and D. Shasha, "D-over: An optimal on-line scheduling algorithm for overloaded real-time systems," in *Proc. IEEE Real-Time Systems Symposium*, December 1992, pp. 290–299.

[11] D. Mosse, M. E. Pollack, and Y. Ronen, "Value-density algorithm to handle transient overloads in scheduling," in *Proc. Euromicro Conference on Real-Time Systems*, June 1999, pp. 278–286.

[12] W. T. Strayer, "Function-driven scheduling: A general framework for expressing and analysis of scheduling," Ph.D. dissertation, University of Virginia, May 1992, department of Computer Science.

[13] D. P. Maynard, S. E. Shipman, R. K. Clark, J. D. Northcutt, R. B. Kegley, B. A. Zimmerman, and P. J. Keleher, "An example real-time command, control, and battle management application for alpha," CMU Computer Science Dept., Tech. Rep., December 1988, archons Project Technical Report 88121.

[14] R. Clark, E. D. Jensen, A. Kanevsky, J. Maurer, P. Wallace, T. Wheeler, Y. Zhang, D. Wells, T. Lawrence, and P. Hurley, "An adaptive, distributed airborne tracking system," in *Proceedings of IEEE International Workshop on Parallel and Distributed Real-Time Systems*, ser. Lecture Notes in Computer Science, vol. 1586. Springer-Verlag, April 1999, pp. 353–362.

[15] E. D. Jensen, "Asynchronous decentralized real-time computer systems," in *Real-Time Computing*, W. A. Halang and A. D. Stoyenko, Eds. Springer Verlag, October 1992.

[16] IEEE and OpenGroup, "The open group base specifications issue 6 (ieee std. 1003.1)," http://www.opengroup.org/onlinepubs/007904975/nframe.html, 2003.

[17] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Boston: Kluwer Academic Publishers, 1997.

[18] P. Li, B. Ravindran, J. Wang, and G. Konowicz, "Choir: A real-time middleware architecture supporting benefit-based proactive resource allocation," in *Proc. of IEEE International Symposium on Object-oriented Real-time distributed Computing*, May 2003, pp. 292–299.

[19] W. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quaterly*, vol. 21, pp. 177–185, 1974.

[20] P. Ramanathan, "Overload management in real-time control applications using the (m, k) firm guarantee," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 549–559, 1999.

[21] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha, "On-line scheduling in the presence of overload," in *Proc. IEEE Annual Symposium on Foundations of Computer Science*, October 1991, pp. 101–110.

[22] S. A. Aldarmi and A. Burns, "Dynamic value-density for scheduling real-time systems," in *Proc. of Euromicro Conference on Real-Time Systems*, June 1999, pp. 270–277.

[23] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise computations," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 83–94, Janurary 1994.

[24] D. M. P. Mejia-Alvarez, R. Melhem, "An incremental approach to scheduling during overloads in real-time systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, December 2000, pp. 283–293.

[25] C. Lu, J. Stakovic, G. Tao, and S. Son, "Feedback control real-time scheduling: Framework, modeling and algorithms," *Journal of Real-Time Systems*, vol. 23, no. 1/2, pp. 85–126, July 2002.

[26] S. A. Aldarmi and A. Burns, "Time-cognizant value functions for dynamic real-time scheduling," Department of Computer Science, The University of York, U.K., Tech. Rep., 1998, yCS-306.

[27] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for qos management," in *Proceedings of The IEEE Real-Time Systems Symposium*, 1997, pp. 298–307.