On Scheduling Exception Handlers in Dynamic Real-Time Systems

Binoy Ravindran*, Edward Curley*, and E. Douglas Jensen‡

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

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

Abstract

We consider the problem of scheduling exception handlers in real-time systems that operate under run-time uncertainties including those on execution times, activity arrivals, and failure occurrences. The application/scheduling model includes activities and their exception handlers that are subject to time/utility function (TUF) time constraints and an utility accrual (UA) optimality criterion. A key underpinning of the TUF/UA scheduling paradigm is the notion of "best-effort" where high importance activities are always favored over low importance ones, irrespective of activity urgency. (This is in contrast to classical admission control models which favor feasible completion of admitted activities over admitting new ones, irrespective of activity importance.) We consider a transactional style activity execution paradigm, where handlers that are released when their activities fail (e.g., due to time constraint violations) abort the failed activities after performing recovery actions. We present a scheduling algorithm called *Handler-assured Utility accrual Algorithm* (or HUA) for scheduling activities and their handlers. We show that HUA's properties include bounded-time completion for handlers and bounded loss of the best-effort property. Our implementation experience on a RTSJ (Real-Time Specification for Java) Virtual Machine demonstrates the algorithm's effectiveness.

I. Introduction

Embedded real-time systems that are emerging in many domains such as robotic systems in the space domain (e.g., NASA/JPL's Mars Rover [4]) and control systems in the defense domain (e.g., airborne trackers [2]) are fundamentally distinguished by the fact that they operate in environments with dynamically uncertain properties. These uncertainties include transient and sustained resource overloads due to context-dependent activity execution times, and non-deterministically distributed activity arrivals and failure occurrences (which may also cause overloads). Nevertheless, such systems require the strongest possible assurances on activity timeliness behavior. Another important distinguishing feature of most of these systems that are of interest to us is their relatively long activity execution time magnitudes, compared to conventional real-time subsystems—e.g., from milliseconds to minutes.

When resource overloads occur, meeting time constraints of all activities is impossible as the demand exceeds the supply. The urgency of an activity is sometimes orthogonal to the relative importance

of the activity—-e.g., the most urgent activity may be the least important, and vice versa; the most urgent may be the most important, and vice versa. Hence when overloads occur, completing the most important activities irrespective of activity urgency is often desirable. Thus, a clear distinction has to be made between urgency and importance during overloads. (During underloads, such a distinction generally need not be made, especially if all time constraints are deadlines, as algorithms that can meet all deadlines exist for those situations—e.g., EDF [6].)

Deadlines by themselves cannot express both urgency and importance. Thus, we consider the abstraction of *time/utility functions* (or TUFs) [7] that express the utility of completing an activity as a function of that activity's completion time. We specify a deadline as a binary-valued, downward "step" shaped TUF; Figure 1(a) shows examples. Note that a TUF decouples importance and urgency—i.e., urgency is measured on the X-axis, and importance is denoted (by utility) on the Y-axis. TUFs usually have a *termination time* — the latest time after which the function is not defined. For downward step TUFs, this time generally is the function's discontinuity point.

Some real-time systems also have activities with *non-deadline* time constraints, such as those where the utility attained for activity completion *varies* (e.g., decreases, increases) with completion time. Figures 1(b)–1(c) show examples from two defense applications [2], [12].

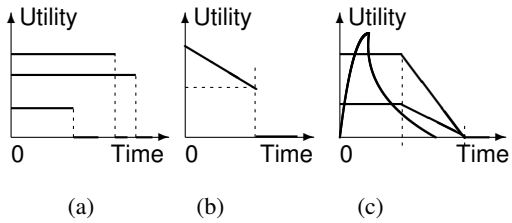

Fig. 1. Example TUF Time Constraints: (a) Step TUFs; (b) TUF of an AWACS [2]; and (c) TUFs of a Coastal Air defense System [12].

When activity time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued activity utility—e.g., maximizing the total activity accrued utility. Such criteria are called *utility accrual* (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that optimize UA criteria are called UA sequencing algorithms (see [15] for example algorithms). UA criteria may also include other factors—e.g., dependencies that may arise between activities due to synchronization.

UA algorithms that maximize total utility under downward step TUFs (e.g., [3], [11], [19]) default to EDF during underloads, since EDF satisfies all deadlines during underloads. Consequently, they obtain the optimum total utility during underloads. During overloads, they inherently favor more important activities over less important ones (since more utility can be attained from the former), irrespective of activity urgency, and thus exhibit adaptive behavior and graceful timeliness degradation. This behavior of UA algorithms is called "best-effort" [11] in the sense that the algorithms strive their best to feasibly complete as many high importance activities — as specified by the application through TUFs — as possible.[1] Consequently, high importance activities that arrive at any time always have a very high

[1]Note that the term "best effort" as used in the context of networks actually is intended to mean "least effort."

likelihood for successful completion (irrespective of their urgency). Note also that EDF's optimal timeliness behavior is a special-case of UA scheduling.

A. *Contributions: Scheduling Exception Handlers with Timing Assurances*

When a failure occurs to an activity in such dynamic systems (e.g., an execution overrun causing a time constraint violation, logical error), an application-supplied exception handler is often immediately released for execution. Such handlers may have time constraints themselves and will compete for the processor along with other activities. Under a termination model, when a handler executes (not necessarily when it is released), it will abort the failed activity after performing transactional-style recovery actions that are necessary to avoid inconsistencies (e.g., [17]). After a handler completes its execution, the application may desire to resume the execution of the failed activity's logic—e.g., the parent activity (or another activity that was waiting for the completion) of the failed activity creates a new child activity to resuscitate the failed activity's logic.

Scheduling of the handlers (along with their activities) must contribute to system-wide timeliness optimality. Untimely handler execution can degrade timeliness optimality—e.g.: high urgency handlers are delayed by low urgency non-failed activities, thereby delaying the resumption of high urgency failed activities; high urgency, non-failed activities are delayed by low urgency handlers.

A straightforward approach for scheduling handlers is to conceptually model them as normal activities, insert them into the ready queue when activities arrive, and schedule them along with the normal activities, according to a discipline that provides acceptably optimal system-wide timeliness. This should be possible, as handlers are like normal activities, with similar scheduling parameters (e.g., execution time, time constraints). However, doing so causes some serious difficulties:

(1) Constructing a schedule that includes an activity and its handler implies that the activity *and* the handler will be dispatched for execution according to their order in the schedule. This is not true, as the handler needs to be dispatched only if and when the activity fails;

(2) When an activity is released for execution, which is a scheduling event, it is immediately ready for execution. However, its handler is released for execution only if and when the activity fails. Thus, constructing a schedule at an activity's release time such that it also includes the activity's handler will require a prediction of when the handler will be ready for execution in the future — a potentially impossible problem as there is no way to know if an activity will fail.

These problems can possibly be alleviated by considering an activity's failure time as a scheduling event and constructing a schedule that includes the activity's handler at that time. Doing so means that there is no way to know whether or not the handler can feasibly complete, satisfying its time

constraint, until the activity fails. In fact, it is quite possible that when the activity fails, the scheduler may discover that the handler is infeasible due to an overload — e.g., there are more activities than can be feasibly scheduled, and there exists a schedule of activities excluding the handler from which more utility can be attained than from one including the handler.

Another strategy that avoids this predicament and has been very often considered in the past (e.g., [1], [13], [18]) is classical *admission control*: When an activity arrives, check whether a feasible schedule can be constructed that includes all the previously admitted activities and their handlers, besides the newly arrived one and its handler. If so, admit the activity and its handler; otherwise, reject. But this will cause the very fundamental problem that is solved by UA schedulers through its best-effort decision making—i.e., a newly arriving activity is rejected because it is infeasible, despite that activity being the most important. In contrast, UA schedulers will feasibly complete the high importance newly arriving activity (with high likelihood), at the expense of not completing some previously arrived ones, since they are now less important than the newly arrived.

Note that this problem does not occur in hard real-time systems (i.e., those that are assured to meet all deadlines) because the arrival and execution behaviors of activities are statically known. Thus, activities and their handlers are statically scheduled to ensure that all deadlines are met; if no feasible schedule exists, the application is redesigned until one exists [8].

Thus, scheduling handlers to ensure their system-wide timely execution in dynamic systems involves an apparently paradoxical situation: an activity may arrive at any (unknown) time; in the event of its failure, which is unknown until the failure occurs, a handler is immediately released, and as strong assurances as possible must be provided for the handler's feasible completion.

We precisely address this problem in this paper. We consider real-time activities that are subject to TUF time constraints. Activities may have arbitrary arrival behaviors and failure occurrences. Activities may synchronize their execution for serially sharing non-CPU resources, causing dependencies. For such a model, we consider the scheduling objective of maximizing the total utility accrued by all activities on one processor. This problem is $\mathcal{NP}$-hard. We present a polynomial-time heuristic algorithm called the *Handler-assured Utility accrual Algorithm* (or HUA).

We show that HUA ensures that handlers of activities that encounter failures during their execution will complete within a bounded time. Yet, the algorithm retains the fundamental best-effort property of UA algorithms with bounded loss—i.e., a high importance activity that may arrive at any time has a very high likelihood for successful completion. HUA also exhibits other properties including optimal total utility for a special case, deadlock-freedom, and correctness. Our implementation experience of HUA on a RTSJ Virtual Machine demonstrates the algorithm's effectiveness.

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

The rest of the paper is organized as follows: Section II outlines our activity model and state the scheduling objectives. We present HUA in Section III and establish the algorithm's properties in Section IV. Section V reports our implementation experience. We conclude the paper in Section VI.

II. Models and Objectives

A. Threads and Scheduling Segments

Our basic scheduling entity is the thread abstraction. Thus, the application consists of a set of threads, denoted $T_i, i \in \{1, 2, ..., n\}$. Threads can arrive arbitrarily and be preempted arbitrarily.

A thread can be subject to time constraints. A time constraint usually has a "scope"—a segment of the thread control flow that is associated with a time constraint [14]. We call such a scope a "scheduling segment." 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."

A thread presents an execution time estimate of its scheduling segment to the scheduler when it enters that segment. This time estimate is not the worst-case; it can be violated at run-time (e.g., due to context dependence) and can cause CPU overloads.

B. Resource Model

Threads can access non-CPU resources including physical (e.g., disks) and logical (e.g., locks) resources. Resources can be shared, and can be subject to mutual exclusion constraints.

Similar to resource access models for fixed-priority scheduling [16] and that for TUF/UA scheduling [3], [10], we consider a single-unit resource model. Thus, only a single instance is present for each resource and a thread must explicitly specify the resource that it needs.

A thread may request multiple shared resources during its lifetime. The requested time intervals for holding resources may be nested or disjoint. We assume that a thread explicitly releases all granted resources before the end of its execution.

C. Timeliness Model

A thread's time constraints are specified using TUFs. A TUF is always associated with a thread scheduling segment and is presented by the thread to the scheduler when the thread enters that segment. We focus on *non-increasing* unimodal TUFs, as they encompass the majority of the time constraints of interest to us. Figures 1(a), 1(b), and two TUFs in Figure 1(c) show examples.

Each TUF has an *initial* time and a *termination* time, which are the earliest and the latest times for which the TUF is defined, respectively. We assume that the initial time is the thread release time; thus a thread's absolute and relative termination times are the same. In this paper, we also assume that the termination time of a downward step TUF is its discontinuity point.

D. Exceptions and Abortion Model

An exception handler is assumed to be associated with each scheduling segment of a thread. We consider a termination model for failures that are encountered during thread executions including time-constraint violations and logical errors. When a thread segment encounters such a failure during its execution, an exception is raised, and the segment's handler is immediately released.

When the handler executes (not necessarily when it is released), it will abort the thread after performing transactional-style (e.g., [17]) compensations and recovery actions that are necessary to avoid inconsistencies—e.g., rolling back, rolling forward, or making other compensations to logical and physical resources that are held by the failed thread to safe states. Often, the handler will also perform actions that are required to ensure the safety and stability of the external state.

A handler also has a time constraint, which is specified using a TUF. The handler's TUF's initial time is the time of failure of the handler's thread. The handler's TUF's termination time is relative to its initial time. Thus, a handler's absolute and relative termination times are *not* the same.

A handler also specifies an execution time estimate. This estimate along with the handler's TUF are described by the handler's thread when the thread enters the corresponding scheduling segment.

To summarize, when a thread enters a scheduling segment, it presents the following scheduling parameters to the scheduler: (1) execution time estimate of the scheduling segment; (2) time constraint of the segment (described using a TUF); (3) execution time estimate of the segment's exception handler; and (4) time constraint of the handler (described using a TUF).

A thread is assumed to present these scheduling parameters to the scheduler through a scheduling API that it invokes when entering a scheduling segment. Example such scheduling APIs include Real-Time CORBA 1.2's [14] `begin_scheduling_segment` API and [9]'s `REQ_CPU` API that are invoked by distributable threads and normal threads to enter a scheduling segment, respectively.

Handlers are not allowed to mutually exclusively access non-CPU resources. Violation of a handler's absolute termination time will cause the immediate execution of system recovery code, which will recover thread's held resources and return the system to a consistent and safe state.

E. Scheduling Objectives

Our goal is to design a scheduling algorithm that maximizes the sum of the utility accrued by all the threads as much as possible. For downward step TUFs, maximizing the total utility subsumes meeting all TUF termination times as a special case. For non-step TUFs, this is not the case, as different utilities can be accrued depending upon the thread completion time, even when the TUF termination time is met. When all termination times are met for downward step TUFs (possible during underloads), the total accrued utility is the optimum possible. During overloads, for step and non-step TUFs, the goal is to maximize the total utility as much as possible.

Further, the completion time of handlers must be bounded. Moreover, the algorithm must exhibit the best-effort property of UA algorithms (described in Section I) to the extent possible.

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].

III. HUA Scheduling Algorithm

A. Basic Rationale

Since the task model is dynamic—i.e., when threads will arrive, how long they will execute, which set of resources will be needed by which threads, the length of time for which those resources will be needed, the order of accessing the resources are all statically unknown, future scheduling events[2] such as new thread arrivals and new resource requests cannot be considered at a scheduling event. Thus, a schedule must be constructed solely exploiting the current system knowledge.

Since the primary scheduling objective is to maximize the total utility, a reasonable heuristic is a "greedy" strategy: Favor "high return" threads over low return ones, and complete as many of them as possible before thread termination times, as early as possible (since TUFs are non-increasing).

The potential utility that can be accrued by executing a thread defines a measure of its "return on investment." We measure this using a metric called the *Potential Utility Density* (or PUD). A thread's PUD measures the utility that can be accrued per unit time by immediately executing the thread and those thread(s) that it (directly or transitively) depends upon for locked resources.

Since the best-case failure scenario is the absence of failure for the thread and all of its dependents, the corresponding PUD can be obtained as the total utility accrued by executing the thread and its dependents divided by the aggregate execution time spent for executing the thread and its dependents. The PUD for the worst-case failure scenario (one where the thread and all of its dependents fail)

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

can be obtained as the total utility accrued by executing the handler of the thread and that of its dependents divided by the aggregate execution time spent for executing the thread, its handler, the thread's dependents, and the handlers of the dependents.[3] The thread PUD can now be measured as the minimum of these two PUDs, as that represents the worst-case.

Thus, HUA examines threads for potential inclusion in a feasible schedule in the order of decreasing PUDs. For each thread, the algorithm examines whether the thread and its handler, along with the thread's dependents and their handlers, can be feasibly completed. If infeasible, the thread, its handler, the dependents, and their handlers are rejected. The process is repeated until all threads are examined, and the schedule's first thread is dispatched. Rejected threads are reconsidered for scheduling at subsequent scheduling events, until their termination times expire.

This process ensures that the threads included in the schedule at any given time have feasible handlers, thereby ensuring that when those threads encounter failures *during* execution, their handlers are assured to complete. Note that no such assurances are afforded to thread failures that are encountered otherwise—e.g., when termination times of threads that are rejected at a scheduling event eventually expire. Handlers for those failures are executed in a best-effort manner—i.e., in accordance with their potential contribution to the total utility (at termination time expirations).

Handler Feasibility. Feasibility of a thread can be tested by verifying whether the thread can complete before its termination time. For a handler, feasibility means whether it can complete before its *absolute* termination time, which is the time of thread failure plus the handler's termination time. Since the thread failure time is impossible to predict, possible choices for the handler's absolute termination time include: (A) predicted thread completion time (in the current schedule) plus the handler's termination time; and (B) thread's termination time plus the handler's termination time.

The difference between A and B is in the delay suffered by the handler before its execution begins (A incurs less delay than B). Delaying the handler's start time (until its latest start time) potentially allows threads that may arrive later but with an earlier termination time than that of the handler to be feasibly scheduled. Thus, B is more appropriate from the standpoint of maximizing total utility.

There is always the possibility that a new thread T_i may arrive after the failure of another thread T_j but before the completion of T_j's handler. As per the best-effort philosophy, T_i must immediately be afforded the opportunity for feasible execution, in accordance with its potential contribution to the total utility. However it is possible that a schedule that includes T_i may not include T_j's handler. Since T_j's handler cannot be rejected, as that will violate the commitment made to T_j, the only option left

[3]Note that, in the worst-case failure scenario, utility is accrued only for executing the thread handlers; no utility is gained for executing the threads themselves, though execution time is spent for executing the threads and the handlers.

is to not consider T_i for execution until T_j's handler completes, consequently degrading the best-effort property. In Section IV, we quantify this loss, and thereby establish the tradeoff between bounding handler completion times and the loss of the best-effort property.

We now overview the algorithm, and subsequently describe each of its components in detail.

B. Overview

HUA's scheduling events include the arrival of a thread, completion of a thread or a handler, a resource request, a resource release, and the expiration of a TUF termination time. To describe HUA, we define the following variables and auxiliary functions:

- $\mathcal{T}_r$ is the current set of unscheduled threads. $T_i \in \mathcal{T}_r$ is a thread. T_i^h denotes T_i's handler.
- σ is the ordered schedule. $\sigma(i)$ denotes the thread occupying the i^{th} position in schedule σ.
- $U_i(t)$ denotes T_i's TUF; $U_i^h(t)$ denotes T_i^h's TUF.
- $T_i.X$ is T_i's termination time. $T_i.ExecTime$ is T_i's estimated remaining execution time. $T_i.Dep$ is T_i's dependency list.
- H is the set of handlers that are *released* for execution, ordered by non-decreasing handler termination times. A handler is said to be released for execution when the handler's thread fails. $H = \emptyset$ if all released handlers have completed.
- Function `updateReleaseHandlerSet()` inserts a handler T_i^h into H if the scheduler is invoked due to a thread T_i's failure; deletes a handler T_i^h from H if the scheduler is invoked due to T_i^h's completion. Insertion of T_i^h into H is at the position corresponding to T_i^h's termination time.
- `Owner(`R`)` denotes the threads that are currently holding resource R; `reqRes(`T`)` returns the resource requested by T.
- `headOf(`σ`)` returns the first thread in σ.
- `sortByPUD(`σ`)` returns a schedule ordered by non-increasing thread PUDs. If two or more threads have the same PUD, then the thread(s) with the largest $ExecTime$ will appear before any others with the same PUD.
- `Insert(`T`,`σ`,`I`)` inserts T in the ordered list σ at the position indicated by index I; if entries in σ exists with the index I, T is inserted before them. After insertion, T's index in σ is I.
- `Remove(`T`,`σ`,`I`)` removes T from ordered list σ at the position indicated by index I; if T is not present at the position in σ, the function takes no action.
- `lookup(`T`,`σ`)` returns the index value of the first occurrence of T in the ordered list σ.
- `feasible(`σ`)` returns a boolean value indicating schedule σ's feasibility. σ is feasible, if the predicted completion time of each thread T in σ, denoted $T.C$, does not exceed T's termination

time. $T.C$ is the time at which the scheduler is invoked plus the sum of the $ExecTime$'s of all threads that occur before T in σ and $T.ExecTime$.

```
1:  input: T_r, H; output: selected thread T_exe;
2:  Initialization: t := t_cur; σ := ∅;
3:  updateReleaseHandlerSet ();
4:  for each thread T_i ∈ T_r do
5:      if feasible(T_i) =false then
6:          reject(T_i);
        else
7:          T_i.LUD = min( U_i(t+T_i.ExecTime)/T_i.ExecTime , U_i^h(t+T_i.ExecTime+T_i^h.ExecTime)/(T_i.ExecTime+T_i^h.ExecTime) );
8:          T_i.Dep := buildDep(T_i);
9:  for each thread T_i ∈ T_r do
10:     T_i.PUD:=calculatePUD(T_i, t);
11: σ_tmp :=sortByPUD(T_r);
12: for each thread T_i ∈ σ_tmp from head to tail do
13:     if T_i.PUD > 0 then
14:         σ := insertByETF(σ, T_i);
15:     else break;
16: HandlerIsMissed := false ;
17: if H ≠ ∅ then
18:     for each thread T^h ∈ H do
19:         if T^h ∉ σ then
20:             HandlerIsMissed := true;
21:             break;
22: if HandlerIsMissed := true then
23:     T_exe :=headOf(H);
    else
24:     T_exe:=headOf(σ);
25: return T_exe;
```

Algorithm 1: HUA: High Level Description

Algorithm 1 describes HUA at a high level of abstraction. When invoked at time t_{cur}, HUA first updates the set H (line 3) and checks the feasibility of the threads. If a thread's earliest predicted completion time exceeds its termination time, it is rejected (line 6). Otherwise, HUA calculates the thread's *Local Utility Density* (or LUD) (line 7), and builds its dependency list (line 8).

The PUD of each thread is computed by the procedure `calculatePUD()`, and the threads are then sorted by their PUDs (lines 10–11). In each step of the *for*-loop from line 12 to 15, the thread

with the largest PUD, its handler, the thread's dependents, and their handlers are inserted into σ, if it can produce a positive PUD. The output schedule σ is then sorted by the threads' termination times by the procedure `insertByETF()`.

If one or more handlers have been released but have not completed their execution (i.e., $H \neq \emptyset$; line 17), the algorithm checks whether any of those handlers are missing in the schedule σ (lines 18–21). If any handler is missing, the handler at the head of H is selected for execution (line 23). If all handlers in H have been included in σ, the thread at the head of σ is selected (line 24).

C. Computing Dependency Lists

HUA builds the *dependency list* of each thread—that arises due to mutually exclusive resource sharing—by following the chain of resource request and ownership.

```
1: input: Thread T_k; output: T_k.Dep ;
2: Initialization : T_k.Dep := T_k; Prev := T_k;
3: while reqRes(Prev) ≠ ∅ ⋀
   Owner(reqRes(Prev)) ≠ ∅ do
4:     T_k.Dep :=Owner(reqRes(Prev)) ·T_k.Dep;
5:     Prev := Owner(reqRes(Prev));
```

Algorithm 2: `buildDep(`T_k`)`: Building Dependency List for a Thread T_k

Algorithm 2 shows this procedure for a thread T_k. For convenience, the thread T_k is also included in its own dependency list. Each thread T_l other than T_k in the dependency list has a successor job that needs a resource which is currently held by T_l. Algorithm 2 stops either because a predecessor thread does not need any resource or the requested resource is free. Note that "$\cdot$" denotes an append operation. Thus, the dependency list starts with T_k's farthest predecessor and ends with T_k.

D. Resource and Deadlock Handling

To handle deadlocks, we consider a deadlock detection and resolution strategy, instead of a deadlock prevention or avoidance strategy precisely due to the dynamic nature of the systems of interest — which resources will be needed by which threads, for how long, and in what order are all unknown to the scheduler. Under a single-unit resource request model, the presence of a cycle in the resource graph is the necessary and sufficient condition for a deadlock to occur. Thus, a deadlock can be detected by a straightforward cycle-detection algorithm. Such an algorithm is invoked by the scheduler whenever a thread requests a resource. A deadlock is detected if the new edge resulting from the thread's resource request produces a cycle in the resource graph. To resolve the deadlock, some thread needs

to be aborted, which will result in some utility loss. To minimize this loss, we compute the utility that a thread can potentially accrue by itself if it were to continue its execution, which is measured by its LUD (line 7, Algorithm 1). HUA aborts that thread in the cycle with the lowest LUD.

E. Computing Thread PUD

Procedure `calculatePUD()` (Algorithm 3) accepts a thread T_i (with its dependency list) and the current time t_{cur}. It determines T_i's PUD, by assuming that threads in $T_i.Dep$ and their handlers are executed from the current position (at t_{cur}) in the schedule, while following the dependencies.

1: **input**: T_i, t_{cur}; **output**: $T_i.PUD$;
2: *Initialization* : $t_c := 0$, $t_c^h := 0$, $U := 0$, $U^h := 0$;
3: **for** *each thread* $T_j \in T_i.Dep$, *from tail to head* **do**
4: $\quad t_c := t_c + T_j.ExecTime$;
5: $\quad U := U + U_j(t_{cur} + t_c)$;
6: $\quad t_c^h := t_c^h + T_j^h.ExecTime$;
7: $\quad U^h := U^h + U_j^h(t_{cur} + t_c + t_c^h)$;
8: $T_i.PUD := \min\left(U/t_c, U^h/(t_c + t_c^h)\right)$;
9: **return** $T_i.PUD$;

Algorithm 3: `calculatePUD(`T_i`,`t_{cur}`)`: Calculating the PUD of a Thread T_i

To compute T_i's PUD at time t_{cur}, HUA computes the PUDs for the best-case and worst-case failure scenarios and determines the minimum of the two.

For determining T_i's total accrued utility for the best-case failure-scenario, HUA considers each thread T_j that is in T_i's dependency chain, which needs to be completed before executing T_i. The total expected execution time upon completing T_j is counted using the variable t_c of line 4. With the known expected completion time of each thread, we can derive the expected utility for each thread, and thus obtain the total accrued utility U (line 5) for T_i's best-case failure-scenario.

For determining T_i's total accrued utility for the worst-case failure-scenario, the algorithm counts the total expected execution time upon completing T_j's handler using the variable t_c^h of line 6. The total accrued utility for the worst-case failure scenario U^h can be determined once the thread's completion time followed by its handler's completion time is known (line 7).

The best-case and worst-case failure scenario PUDs can be determined as U and U^h divided by t_c and $t_c + t_c^h$, respectively, and the minimum of the two PUDs is determined as T_i's PUD (line 8).

Note that the total execution time of T_i and its dependents consists of two parts: (1) the time needed to execute the threads that directly or transitively block T_i; and (2) T_i's remaining execution time. According to the process of `buildDep()`, all the dependent threads are included in $T_i.Dep$.

Note that each thread's PUD is calculated assuming that they are executed at the current position in the schedule. This would not be true in the output schedule σ, and thus affects the accuracy of the PUDs calculated. Actually, we are calculating the highest possible PUD of each thread by assuming that it is executed at the current position. Intuitively, this would benefit the final PUD, since `insertByETF()` always selects the thread with the highest PUD at each insertion on σ. Also, the PUD calculated for the dispatched thread at the head of σ is always accurate.

F. Constructing Termination Time-Ordered Feasible Schedules

Algorithm 4 describes `insertByETF()` (invoked in Algorithm 1, line 14). `insertByETF()` updates the tentative schedule σ by attempting to insert each thread, along with its handler, all of the thread's dependent threads, and their handlers into σ. The updated schedule σ is an ordered list of threads, where each thread is placed according to the termination time that it *should* meet.

```
1:  input  : T_i and an ordered thread list σ
2:  output : the updated list σ
3:  if T_i ∉ σ then
4:      Copy σ into σ_tmp: σ_tmp :=σ;
5:      Insert(T_i, σ_tmp, T_i.X);
6:      Insert(T_i^h, σ_tmp, T_i.X + T_i^h.X);
7:      CuTT = T_i.X;
8:      for each thread T_j ∈ {T_i.Dep − T_i} from head to tail do
9:          if T_j ∈ σ_tmp then
10:             TT=lookup(T_j, σ_tmp);
11:             if TT < CuTT then
                    continue;
12:             else
13:                 Remove(T_j, σ_tmp, TT);
14:                 TTh=lookup(T_j^h, σ_tmp);
15:                 Remove(T_j^h, σ_tmp, TTh);
16:         CuTT:=min(CuTT, T_j.X);
17:         Insert(T_j, σ_tmp, CuTT);
18:         Insert(T_j^h, σ_tmp, T_j.X + T_j^h.X);
19:     if feasible(σ_tmp) then
20:         σ := σ_tmp;
21: return σ;
```

Algorithm 4: `insertByETF`(σ, T_i) : Inserting a Thread T_i, T_i's Handler, T_i's Dependents, and their Handlers into a Termination Time-Ordered Feasible Schedule σ

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

A thread may need to meet an earlier termination time in order to enable another thread to meet its termination time. Whenever a thread is considered for insertion in σ, it is scheduled to meet its own termination time. However, all of the threads in its dependency list must execute before it can execute, and therefore, must precede it in the schedule. The index values of the dependent threads may be changed with `Insert()` in line 17 of Algorithm 4.

The variable $CuTT$ keeps track of this information. It is initialized with the termination time of thread T_i, which is tentatively added to the schedule (line 7). Thereafter, any thread in $T_i.Dep$ with a later termination time than $CuTT$ is required to meet $CuTT$ (lines 13; 16–17). If, however, a thread has a tighter termination time than $CuTT$, then it is scheduled to meet that time (line 11), and $CuTT$ is advanced to that time since all threads left in $T_i.Dep$ must complete by then (lines 16–17).

When T_i (or any thread $T_j \in T_i.Dep$) is inserted in σ, its handler T_i^h is immediately inserted to meet a termination time that is equal to T_i's termination time plus T_i^h's (relative) termination time (lines 6, 18). When a thread in $T_i.Dep$ with a later termination time than $CuTT$ is advanced to meet $CuTT$, the thread's handler is also correspondingly advanced (lines 14–15; 18).

Finally, if this insertion (of T_i, its handler, threads in $T_i.Dep$, and their handlers) produces a feasible schedule, then the threads are included in this schedule; otherwise, not (lines 19–20).

Computational Complexity. With n threads, HUA's asymptotic cost is $O(n^2 \log n)$ (for brevity, we skip the analysis). Though this cost is higher than that of many traditional real-time scheduling algorithms, it is justified for applications with longer execution time magnitudes such as those that we focus on here. (Of course, this high cost cannot be justified for every application.)

IV. Algorithm Properties

We first describe HUA's bounded-time completion property for exception handlers:

Theorem 1: If a thread T_i encounters a failure during its execution, then under HUA with zero overhead, its handler T_i^h will complete no later than $T_i.X + T_i^h.X$ time units (barring T_i^h's failure).

Proof: If T_i fails at a time t during its execution, then T_i was included in HUA's schedule constructed at the scheduling event that occurred nearest to t, say at t', since only threads in the schedule are executed (lines 23–25, Algorithm 1). If T_i was in HUA's schedule at t', then both T_i and T_i^h (besides T_i's dependents and their handlers) were feasible at t', since infeasible threads and their handlers (along with their dependents) are rejected by HUA (lines 19–20, Algorithm 4). Thus, T_i^h was scheduled to complete no later than $T_i.X + T_i^h.X$ (lines 6, 18, Algorithm 4). ■

When a thread T_i arrives after the failure of a thread T_j but before the completion of T_j^h, HUA *may* exclude T_i from a schedule until T_j^h completes, resulting in some loss of the best-effort property. To quantify this loss, we define the concept of a *Non Best-effort time Interval* (or NBI):

Definition 1: Consider a scheduling algorithm $\mathcal{A}$. Let a thread T_i arrive at a time t with the following properties: (a) T_i and its handler together with all threads in $\mathcal{A}$'s schedule at time t are not feasible at t, but T_i and its handler are feasible just by themselves;[4] (b) One or more handlers (which were released due to thread failures before t) have not completed their execution at t; and (c) T_i has the highest PUD among all threads in $\mathcal{A}$'s schedule at time t. Now, $\mathcal{A}$'s NBI, denoted $NBI_{\mathcal{A}}$, is defined as the duration of time that T_i will have to wait after t, before it is included in $\mathcal{A}$'s feasible schedule. Thus, T_i is assumed to be feasible together with its handler at $t + NBI_{\mathcal{A}}$.

We now describe the NBI of HUA and other UA algorithms including DASA [3], LBESA [11], and AUA [5] (under zero overhead):

Theorem 2: HUA's worst-case NBI is $t + \max_{\forall T_j \in \sigma_t} \left(T_j.X + T_j^h.X\right)$, where σ_t denotes HUA's schedule at time t. DASA's and LBESA's worst-case NBI is zero; AUA's is $+\infty$.

Proof: The time t that will result in the worst-case NBI for HUA is when $\sigma_t = H \neq \emptyset$. By NBI's definition, T_i has the highest PUD and is feasible. Thus, T_i will be included in the feasible schedule σ, resulting in the rejection of some handlers in H. Consequently, the algorithm will discard σ and select the first handler in H for execution. In the worst-case, this process repeats for each of the scheduling events that occur until all the handlers in σ_t complete (i.e., at handler completion times), as T_i and its handler may be infeasible with the remaining handlers in σ_t at each of those events. Since each handler in σ_t is scheduled to complete by $\max_{\forall T_j \in \sigma_t} \left(T_j.X + T_j^h.X\right)$, the earliest time that T_i becomes feasible is $t + \max_{\forall T_j \in \sigma_t} \left(T_j.X + T_j^h.X\right)$.

DASA and LBESA will examine T_i at t, since a task arrival is always a scheduling event for them. Further, since T_i has the highest PUD and is feasible, they will include T_i in their feasible schedules at t (before including any other tasks), yielding a zero worst-case NBI.

AUA will examine T_i at t, since a thread arrival at any time is a scheduling event under it. However, AUA is a TUF/UA algorithm in the classical admission control mould and will reject T_i in favor of previously admitted threads, yielding a worst-case NBI of $+\infty$. ■

Theorem 3: The best-case NBI of HUA, DASA, and LBESA is zero; AUA's is $+\infty$.

Proof: HUA's best-case NBI occurs when T_i arrives at t and the algorithm includes T_i and all handlers in H in the feasible schedule σ (thus the algorithm only rejects some *threads* in σ_t to construct σ). Thus, T_i is included in a feasible schedule at time t, resulting in zero best-case NBI.

[4]If $\mathcal{A}$ does not consider a thread's handler for feasibility (e.g., [3], [11]), then the handler's execution time is regarded as zero.

The best-case NBI scenario for DASA, LBESA, and AUA is the same as their worst-case. ■

Thus, HUA's NBI interval $[0, \max_{\forall T_j \in \sigma_t} T_j.X + T_j^h.X]$ lies in between that of DASA/LBESA's $[0]$ and AUA's $[+\infty]$. Note that HUA and AUA bound handler completions; DASA/LBESA do not.

HUA produces optimum total utility for a special case:

Theorem 4: Consider a set of independent threads subject to step TUFs. Suppose there is sufficient processor time for meeting the termination-times of all threads and their handlers. Now, a schedule produced by EDF [6] is also produced by HUA, yielding equal totaly utility.

Proof: For a thread T_i without dependencies, $T_i.Dep$ only contains T_i. During underloads, σ from line 14 of Algorithm 1 is termination time-ordered. The TUF termination time that we consider is analogous to the deadline in [6]. From [6], an EDF schedule is optimal (with respect to meeting all deadlines) during underloads. Thus, σ yields the same total utility as EDF. ■

HUA also exhibits non-timeliness properties including freedom from deadlocks, correctness (i.e., the resource requested by a thread selected for execution by HUA is free), and mutual exclusion. These properties are self-evident from the algorithm description. For brevity, we omit their proofs.

V. Implementation Experience

We implemented HUA in a real-time Java platform. This platform consisted of the *meta-scheduler* middleware scheduling framework in [9] implemented atop Apogee's Aphelion Real-Time Java Virtual Machine that is compliant with the Real-Time Specification for Java (RTSJ). Aphelion JVM is an RTSJ-compliant real-time extension of the J9 IBM JVM (version 2.1). This RTSJ platform runs atop the Debian Linux OS (kernel version 2.6.16-2-686) on a 800MHz, Pentium-III processor.

Besides HUA, we implemented DASA and a simplified variant of HUA called HUA-Non-Preemptive (or HUA-NP), for a comparative study. DASA does not consider handlers for scheduling until failures occur. When a thread fails, DASA then considers its handler for scheduling just like a regular thread, resulting in zero NBI. Similar to DASA, HUA-NP also does not consider handlers for scheduling until failures occur. However, when a thread fails, unlike DASA, HUA-NP immediately runs the thread handler non-preemptively till completion, resulting in a worst-case and best-case NBI of one handler execution time. In this way, HUA-NP seeks to accrue as much utility as possible by excluding handlers from schedule construction (and thus is more greedy than HUA), while maintaining an upper bound on handler completion. Thus, DASA and HUA-NP are good candidates for a comparative study as they represent two interesting end points of the NBI-versus-handler-completion-time tradeoff space.

Our test application created several periodic threads that consume a certain amount of processor time, request a shared resource, and periodically check for abort exceptions. Each thread created had

a unique execution time, period, and maximum utility. These parameters were assigned based on three PUD-based thread classes that were used: *high*, *medium*, and *low*. The classes differed in thread execution times, thread periods, and threads PUDs by one order of magnitude. The classes, however, differed in handler execution times, handler periods, and handler PUDs only by a small factor. Within each class, thread execution times and thread PUDs were higher than that of their handler execution times and handler PUDs, respectively, by one order of magnitude. For all the experiments, an even number of threads from each of the three classes were used. Thus, the three classes give the algorithms a rich mixture of thread properties to exhibit their NBI and handler completion behaviors.

Our metrics to evaluate HUA included the NBI, Handler Completion Time (HCT), Accrued Utility Ratio (AUR), and Deadline Miss Ratio (DMR). HCT is the duration between a handler's completion time and it's release time. AUR is the ratio of the total accrued utility to the maximum possible total utility (possible if every released thread completes before its termination time). DMR is the ratio of the number of threads that missed their termination times to the number of released threads.

We manipulated five variables during our experiments: (1) the percentage of failed threads, (2) system load caused by normal tasks, (3) system load caused by handlers, (4) the ratio of handler execution time to normal task execution time, and (5) the number of shared resources within the system. The variables affect the system's "stress factor" and influence the four metrics.

We measured the four metrics under a constant value for these variables, except for the failure percentage, which was varied between 0% and 95%. To vary the failure percentage, the set of threads that must fail for a given percentage must be repeatable. However, to have a repeatable set of discrete failures (i.e., not a random distribution), the actual percentage of failures may be slightly off from the predicted value—e.g., if an experiment had 50 threads and 25% of them needed to be failed, it is impossible to fail 12.5 threads; thus the failure percentage would be 24% or 26%.

Normal task load was 150%, handler load was 90%, and the ratio of handler execution to normal execution was 50%. We first focused on zero shared resources and then considered shared resources.

Figure 2 shows the measured NBI of DASA, HUA-NP, and HUA under increasing number of failures. We observe that HUA provides a smaller NBI than DASA and HUA-NP. HUA has a smaller NBI than DASA because DASA is unlikely to execute low-PUD threads like handlers. Thus, it is likely to keep them pending and incur a non-zero NBI due to scheduler overhead when a high PUD thread arrives. HUA has a smaller NBI than HUA-NP because HUA-NP will always have a non-zero NBI when a high PUD thread arrives during its non-preemptive handler execution. However, the only time HUA will have a non-zero NBI is when a high PUD thread arrives with such little slack that the pending handlers cannot fit within that slack.

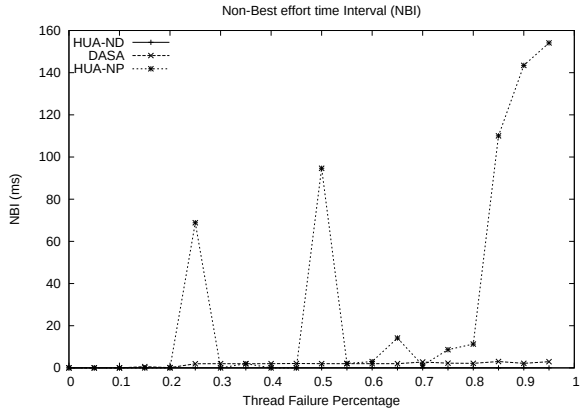

Fig. 2. Non-Best effort time Interval (NBI)

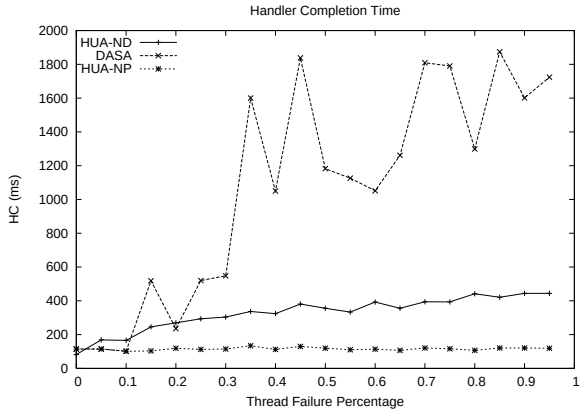

Fig. 3. Handler Completion Time

Figure 3 shows the average HCTs for HUA, DASA, and HUA-NP. In general, DASA's HCTs are highest and rather inconsistent, HUA-NP's are smallest and very consistent, and HUA's are somewhat consistent, but always within a certain bound. As DASA was not designed to bound HCTs, it makes sense that its HCTs would be larger than the other two algorithms. Likewise, it makes sense that HUA-NP would have the least average HCT as the handler is run to completion when it is released. To allow more threads to be scheduled, HUA does not immediately run the handler when it is released. This delay in running the handler causes HUA's average HCT to be higher than HUA-NP's. However, as HUA is designed to provide a finite bound on HCT, it will generally be smaller than DASA's.

Figures 2 and 3 also indicate that the trends acquired from our experiments display a less than smooth response to changes in failure percentage. (Figures 4 and 5 also display this behavior.) This is likely due to the way the failure percentage is varied. Since the set of threads that fail for a given failure percentage is not a strict subset of the set of threads that fail for a larger failure percentage, it is possible that lower-PUD threads may fail at higher failure percentages. Thus, algorithms like DASA and HUA-NP may find a more beneficial schedule at higher failure percentages.

Figure 4 shows how the AUR of each algorithm is affected under increasing failures. In general, HUA will have a lower AUR as it reserves a portion of its schedule for handlers which generally have lower PUDs or may not even execute. However, as can be seen from the figure, HUA has an AUR that is comparable to, if not better than that of DASA and HUA-NP for this thread set. This is because DASA only analyzes handlers that have been released. This limits DASA's ability to discern whether it would be more beneficial to abort the thread and run its handler instead. As HUA has no such limitation, it can better decide whether to run the thread or abort the thread and run its handler.

Figure 5 displays the measured DMR under increasing failures. As the number of failures increases, the number of termination times (or deadlines) missed also increases. This is due to the added load

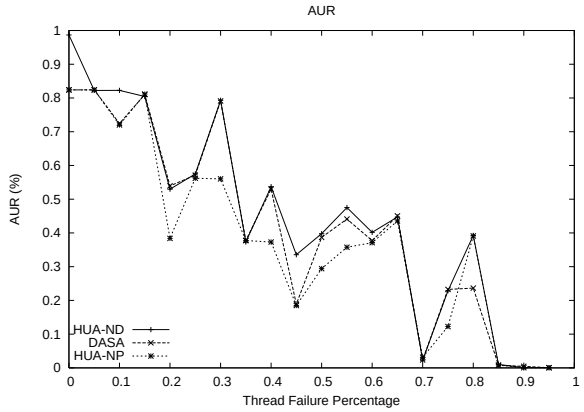

Fig. 4. Accrued Utility Ratio

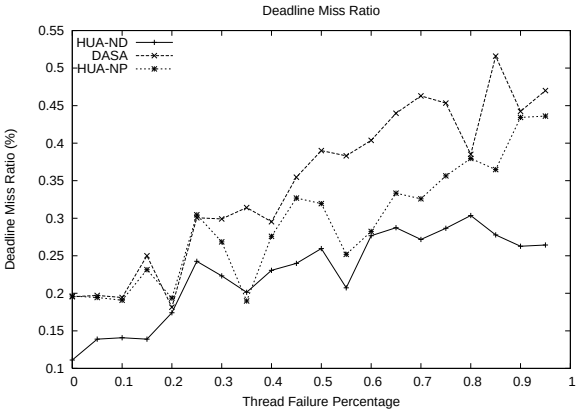

Fig. 5. Deadline Miss Ratio

that handlers put on the system. In the case of DASA, this load is completely unforeseen and as the handlers have less PUD than most normal threads, DASA may never schedule them causing their termination times to be missed. Thus, DASA is affected most by increased failures. While the handler load is also unanticipated for HUA-NP, the effects are mitigated somewhat due to HUA-NP's non-preemptive handler execution property. Because HUA takes the handler load into consideration when forming a schedule, the extra load on the system affects HUA the least.

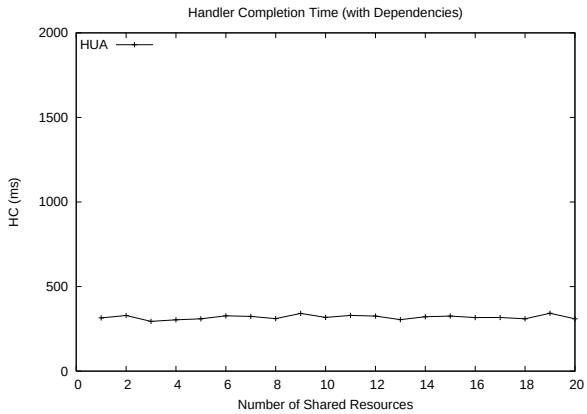

Fig. 6. Handler Completion Time with Dependencies

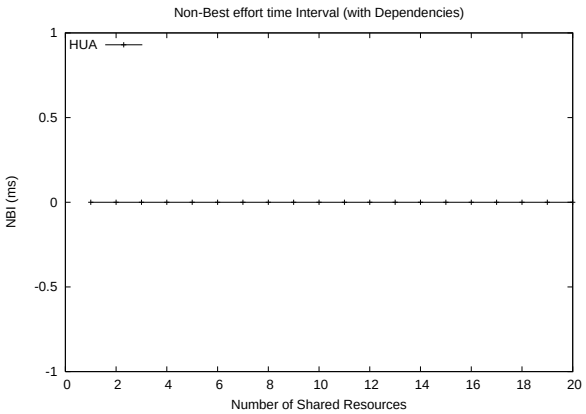

Fig. 7. Non-Best effort time Interval (NBI) with Dependencies

Figure 6 and Figure 7 show the average HCT and average NBI of HUA under increasing number of shared resources. From the figures, we observe that HUA's HCT and NBI are unaffected by dependencies that arise between threads due to shared resources.

VI. Conclusions and Future Work

We present a real-time scheduling algorithm called HUA. The algorithm's application model includes threads and their exception handlers with TUF time constraints, and a transactional-style execution paradigm where handlers abort the failed threads after performing recovery actions. Threads

may serially share non-CPU resources. We show that HUA bounds (A) the completion times of handlers that are released for threads which fail during execution, and (B) the time interval for which a high importance thread arriving during overloads has to wait to be included in a feasible schedule. Our implementation on a RTSJ Virtual Machine demonstrates HUA's effectiveness.

Property (A) is potentially unbounded for best-effort algorithms, and property (B) is potentially unbounded for admission control algorithms. By bounding (A) and (B), HUA conceptually places itself between these two models, allowing for applications to exploit the tradeoff space.

Directions for future work include extending the results to [20]'s variable cost function model for thread/handler execution times, multiprocessor scheduling, and scheduling [14]'s distributable threads.

REFERENCES

[1] A. Bestavros and S. Nagy. Admission control and overload management for real-time databases. In *Real-Time Database Systems: Issues and Applications*, chapter 12. Kluwer Academic Publishers, 1997.

[2] R. Clark et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, pages 353–362, April 1999.

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

[4] R. K. Clark et al. Software organization to facilitate dynamic processor scheduling. In *IEEE WPDRTS*, April 2004.

[5] E. Curley, J. S. Anderson, B. Ravindran, and E. D. Jensen. Recovering from distributable thread failures with assured timeliness in real-time distributed systems. In *IEEE SRDS*, pages 267–276, 2006.

[6] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quaterly*, 21:177–185, 1974.

[7] E. D. Jensen et al. A time-driven scheduling model for real-time systems. In *IEEE RTSS*, pages 112–122, December 1985.

[8] N. Kandasamy, J. P. Hayes, and B. T. Murray. Scheduling algorithms for fault tolerance in real-time embedded systems. In D. R. Avresky, editor, *Dependable Network Computing*, chapter 18. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

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

[10] P. Li, H. Wu, B. Ravindran, and E. D. Jensen. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. *IEEE Transactions on Computers*, 55(4):454 – 469, April 2006.

[11] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986.

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

[13] S. Nagy and A. Bestavros. Admission control for soft-transactions in accord. In *IEEE RTAS*, page 160, 1997.

[14] OMG. Real-time corba 2.0: Dynamic scheduling specification. Technical report, Object Management Group, September 2001.

[15] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *IEEE ISORC*, pages 55 – 60, May 2005.

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

[17] N. R. Soparkar et al. *Time-Constrained Transaction Management*. Kluwer Academic Publishers, 1996.

[18] H. Streich. Taskpair-scheduling: An approach for dynamic real-time systems. *Mini and Microcomputers*, 17(2):77–83, 1995.

[19] H. Wu et al. Utility accrual scheduling under arbitrary time/utility functions and multiunit resource constraints. In *IEEE RTCSA*, pages 80–98, August 2004.

[20] H. Wu et al. Utility accrual real-time scheduling under variable cost functions. In *IEEE RTCSA*, pages 213–219, 2005.