SIGOPS Europe 2004

Adaptive Time-Critical Resource Management Using Time/Utility Functions

Binoy Ravindran, Peng Li, Haisang Wu
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{binoy,peli2,hswu02}@vt.edu

E. Douglas Jensen
The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

ABSTRACT

This position paper makes the case that time/utility functions (or TUFs) and utility accrual optimization criteria constitutes, arguably, the most effective and broadest approach for adaptive, time-critical resource management. A TUF, which is a generalization of the classical deadline constraint, specifies the utility of completing an application activity as an application/ situation-specific function of that activity's completion time. With TUF time constraints, timeliness optimization criteria can be specified in terms of accrued activity utilities. Such utility accrual (or UA) criteria facilitate design of resource management algorithms that are adaptive in the sense that they allocate resources in a mission-oriented way i.e., in the best interests of the application's mission. Further, they gracefully degrade timeliness performance during overloads and gracefully improve performance otherwise. Such timeliness adaptivity is not possible with traditional real-time resource management techniques. We overview past and recent UA algorithms that illustrate this. We also identify emerging challenges.

1. INTRODUCTION

Time-critical resource management is fundamentally concerned with satisfying application time constraints. The most widely studied time constraint is the deadline. A deadline 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. Further, completing the activity after the deadline accrues less utility (zero or sometimes infinitively negative utility). With deadline time constraints, one can specify timeliness optimality criteria such as "meet all deadlines," "minimize maximum lateness," and "minimize deadline-miss rate" and use traditional real-time scheduling algorithms [7] to achieve them.

There two fundamental problems with deadline-based timeliness optimality criteria. First, not all activities in many non-trivial real-time systems have the same functional utility. Typically, some activities may have higher utility than some others and this relative utility can dynamically change. Further, the relative utility of activities are often not directly related to their urgency. For example, the most urgent activity may not have the highest utility at a given time.

This orthogonality of urgency and utility becomes a fundamental issue during resource constrained situations (which may be either be transient or permanent), when sufficient resources are not available for satisfying a collective timeliness optimality criteria such as meeting all deadlines. Many real-time systems are subject to overload situations, typically due to unpredictable event arrivals or execution-time overruns. Thus, during overload situations, the system is confronted with the decision of "shedding" a subset of activities, which is problematic if the scheduling optimality criteria is deadline-based as deadlines only represent urgency and not utility.

Secondly, deadlines imply the attainment of uniform utility for activity completion anytime before the deadline time. This semantics becomes problematic for specifying time constraints, where the utility attained for activity completion *varies* (e.g., increases, decreases) with activity completion time. For example, converting such timing constraints to deadlines will violate their *non-uniform* timeliness semantics.

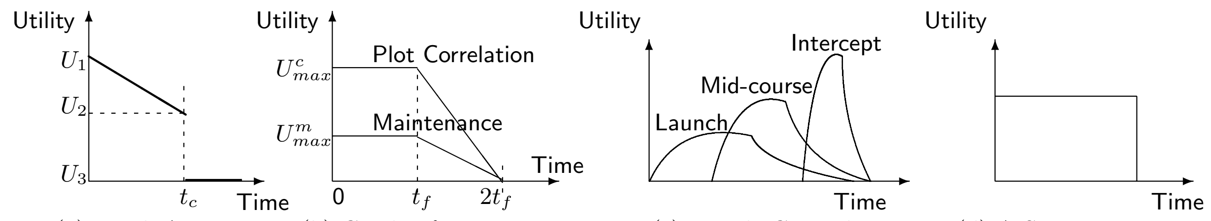
1.1 TUFs and UA Criteria

Jensen's time/utility functions [3] generalizes the deadline constraint. A TUF specifies the utility to the system that results from the completion of an activity as a function of the activity's completion time. A time-utility function (abbreviated here as TUF) specifies the utility to the system of completing an application activity as an application- or situation-specific function of *when* that activity completes.

Figure 1 shows examples of time constraints specified using TUFs. Figures 1(a), 1(b), and 1(c) show time constraints of two significant, real-time applications specified using TUFs. The applications include: (1) the AWACS (Airborne WArning and Control System) surveillance mode tracker system [1] built by The MITRE Corporation and The Open Group (TOG); and (2) a coastal air defense system [9] built by General Dynamics (GD) and Carnegie Mellon University (CMU).

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

.



(a) Track Asso. (b) Corrln. & Maint. (c) Missile Control (d) A Step TUF

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

TUF of the missile control activity dynamically changes.

The classical deadline constraint is a downward "step" shaped TUF. This is shown in Figure 1(d).

Note that a TUF precisely captures the semantics of time constraints with non-uniform (and uniform) timeliness semantics. Further, TUFs decouple activity importance and urgency. Furthermore, predicates that are defined using TUFs of all application activities allow the specification of adaptive timeliness optimality criteria. An example such criterion is to *maximize, accrued* (*e.g., total*) *utility*. With utility accrual predicates, one can design scheduling and resource management algorithms that are precisely driven by the predicates. We call such algorithms, utility accrual (abbreviated as UA) algorithms.

In many situations, an UA algorithm that seeks to maximize total summed utility can facilitate adaptation through graceful performance degradation and performance improvement. For example, when resources are sufficient, such an algorithm will inherently schedule activities and allocate resources such that the maximum possible total utility is obtained. Furthermore, when resources are insufficient to achieve the maximum possible total utility, the algorithm will schedule activities and allocate resources such that as many "high" utility activities complete, as close to their optimal completion times as possible, thereby maximizing the total summed utility.

Of course, maximizing the total summed utility may not always be appropriate. For example, an UA algorithm might complete (say) two activities, yielding utilities that sum to more than the utility that could have otherwise been obtained from another activity. Now, in some situations, completing more activities may be preferable to completing fewer activities, or completing activities having higher maximum utilities may be preferable to maximizing the overall sum, or activities with higher maximum utility may be more "important" regardless of the overall sum. Thus, in such situations, appropriate utility accrual predicates must be formulated (e.g., maximize number of activity-completions as a primary objective, while maximizing total summed utility as a secondary goal). Subsequently, UA algorithms that are driven by such predicates must then be designed.

The class of UA algorithms thus facilitate the construction of adaptive resource management services with the following distinguishing characteristics: (1) they allow time-critical activities including those that have non-uniform timeliness semantics; (2) they dynamically adapt to workload and execution environment uncertainties by gracefully degrading and gracefully improving performance; and (3) they dynamically differentiate importance of activities, irrespective of their urgency.

Thus, we argue that TUF/UA models constitute, arguably, the most effective and broadest approach for adaptive, time-critical resource management.

We now overview four UA scheduling algorithms. These include: (1) Locke's Best Effort Scheduling Algorithm (or LBESA) [8]; (2) Dependent Activity Scheduling Algorithm (or DASA) [2]; (3) Utility Accrual Packet scheduling Algorithm (or UPA) [10]; and (4) Generic Utility Scheduling algorithm (GUS) [5]. These are discussed in Sections 2, 3, 4, and 5, respectively. In Section 6, we report experimental measurements that compare the performance of these algorithms on a POSIX real-time OS implementation. We conclude the paper and identify emerging challenges in TUF/UA research in Section 7.

2. LBESA

The LBESA algorithm [8] is the first known UA scheduling algorithm. The algorithm considers a task model, where tasks are subject to time constraints expressed using almost arbitrarily shaped TUFs, have no resource dependencies, and have variabilities in their execution times, which are stochastically expressed using random variables. Given such a model, the algorithm seeks to schedule tasks so that the accrued utility can be maximized.

A TUF considered by LBESA can be decreasing, constant, and even increasing (e.g., those shown in Figure 1). Further, each TUF is associated with a "deadline" time, which is defined as the latest time after which its utility drops below an application-specified percentage of its maximal utility (e.g., 90% of the maximal utility). This definition of "deadline" can handle TUFs whose utilities delay very slowly.

Two key observations drive the design of LBESA: (1) If the processor is under-loaded, an Earliest-Deadline-First (or EDF) schedule can satisfy all deadlines [7]; and (2) When an overload occurs, a decreasing "value density" order — value density of a task is defined as the attained task utility over its remaining execution time — can yield good performance.

Based on these two observations, LBESA employs an additional heuristic to handle overload: When an overload occurs, tasks are rejected in non-decreasing order of value densities until the remaining task set can meet all its deadlines (or is "feasible"). Then, the earliest deadline task in the remaining task set is selected by mimicking an EDF scheduler. The rational of rejecting tasks in non-decreasing order of value density is to minimize utility loss as much as possible.

Since task execution tines are stochastically described, LBESA computes the *expected* task utility, which, in general, requires integral over a timer interval. Further, for tasks with increasing TUFs, LBESA computes a timer instant, called *earliest starting time* (EST), after which the task can accrue at least some predefined utility. Thus, if such a task is released before its EST, the algorithm ensures that they are not eligible for scheduling until their EST.

LBESA detects overloads, in principle, by comparing task processor time demand against available time until the task deadline. However, due to the randomness of task execution times, the algorithm computes the *probability* of processor overload and compares that with a threshold probability. The processor is determined to be overloaded when the computed probability exceeds the threshold value.

3. DASA

DASA advances LBESA by considering tasks with dependencies i.e., tasks that share resources, which may be subject to mutual exclusion constraints, and precedence relationships. However, DASA only allows deterministic task execution times and downward step TUFs, while LBESA allows stochastic task execution times and almost arbitrary TUFs.

DASA makes scheduling decisions using the concept of *potential value density* (or PVD), which is similar to LBESA's value density. The difference is that DASA's PVD considers both the utility of a task itself and utilities of its *dependent* tasks. If the execution of a task T_i needs a resource R, which is currently held by another task T_j, then T_i is said to be dependent on task T_j. Consequently, task T_j has to be scheduled first before task T_i can be scheduled for execution. Thus, the dependent tasks of a task T_i include those tasks that must be executed before T_i such that T_i can access its needed resources and continue its execution.

DASA aborts tasks for improving timeliness performance. If an aborted task possesses shared resources, then extra processor time is needed to release those resources. DASA uses the notion of *abort time* to model the execution time of cleanup operations on the shared resource, such as resetting a shared variable to a safe and consistent value.

For example, suppose that task T_i needs to access resource R, which is currently held by another task T_j. To allow this, the scheduler can schedule task T_j to continue execution until it releases R. Alternatively, the scheduler can abort T_j, secure resource R, conduct cleanup operations on R, and grant it to T_i. DASA aborts task T_j if T_j's remaining execution time is longer than its abort time. Otherwise, T_j is executed normally. The rational behind this heuristic is to release resource R as soon as possible.

Let $T_i.Dep(t)$ be the set of dependent tasks of task T_i and let $R_j(t)$ be the remaining execution time of a task T_j (or abortion time of T_j if it is aborted). Then, the potential value density of T_i at time t is computed as $PVD_i(t) = \dfrac{U_i + \sum_{T_k \in T_i.Dep} U'_k}{R_i(t) + \sum_{T_k \in T_i.Dep} R_k(t)}$. Note that U'_k is zero if task T_k is aborted and is the full utility of T_k, otherwise.

At each scheduling event, DASA examines tasks in the task ready queue in decreasing order of their PVD values. The algorithm then inserts each task into a tentative schedule at its deadline-position and checks the feasibility of the schedule. Tasks are maintained in increasing deadline-order in the tentative schedule. If inserting a task into the tentative schedule results in an infeasible schedule, then the algorithm removes the task from the schedule.

DASA repeats this process until all tasks in the ready queue have been examined. The algorithm then selects the earliest deadline task in the tentative schedule (which will be at the "head" of the schedule) as the next task to be executed.

Note that if all task deadlines can be satisfied, then DASA's output will be the same as that of EDF. Thus, DASA yields the same timeliness optimality of EDF under identical conditions [7] i.e., for a set of independent periodic tasks, which are subject to step TUFs, and when there is no overload, DASA is optimal with respect to meeting all deadlines and produces the minimum possible maximum lateness.

4. UPA

UPA is a packet scheduling algorithm that executes at the MAC-layer of system nodes (e.g., hosts, switches) for selecting packets for outbound transmission. The algorithm considers a packet model, where packets have non-increasing, unimodal TUFs and seeks to maximize the sum of packets' attained utilities. Unimodal TUFs are those TUFs for which any decrease in utility cannot be followed by an increase. Figure 1 shows examples. Non-increasing unimodal TUFs are simply those unimodal TUFs for which utility never increases as time advances (see Figures 1(a), 1(b), and 1(d)).

UPA first constructs a tentative schedule by sorting packets in decreasing order of their "return of investments." The return of investment for a packet is the potential timeliness utility that can be obtained by spending a unit amount of network transmission time for the packet. Thus, "high return" packets will appear early in the tentative schedule. The return of investment for a packet is determined by simply computing the ratio of the maximum possible packet utility (specified by the packet TUF) to the packet termination time. In [10], this ratio is called "pseudo-slope," since it only gives an approximate measure of the TUF slope.

From this tentative schedule, packets that are found to be infeasible are moved to the end of the schedule. The algorithm then maximizes the *local* aggregate utility in the resulting schedule by observing that given two schedules $\sigma_a = \langle \sigma_1, p_i, p_j, \sigma_2 \rangle$ and $\sigma_b = \langle \sigma_1, p_j, p_i, \sigma_2 \rangle$ of a packet set $\mathcal{A}$, such that $\sigma_1 \neq 0$,

$\sigma_2 \neq 0$, $\sigma_1 \bigcup \sigma_2 = \mathcal{A} - \{p_i, p_j\}$, and $\sigma_1 \bigcap \sigma_2 = \emptyset$, the scheduling decision at a time t, where $t = \sum_{k \in \sigma_1} l_k$, that will lead to maximum local aggregate utility is determined by computing $\Delta_{i,j}(t)$, where $\Delta_{i,j}(t) = [U_i(t+l_i) + U_j(t+l_i+l_j)] - [U_j(t+l_j) + U_i(t+l_j+l_i)]$. Thus, if $\Delta_{i,j}(t) \geq 0$, then schedule σ_a will yield a higher aggregate utility than σ_b.

UPA maximizes local aggregate utility by examining adjacent pairs of packets in the schedule, computing Δ, and swapping the packets, if the reverse order can lead to higher local aggregate utility. The procedure is repeated until no swaps are required. The packet that appears first in the resulting schedule is then selected for transmission.

5. GUS

GUS is a task scheduling algorithm that combines the models of LBESA and DASA. It considers tasks that have dependencies due to shared resources under a single-unit resource request model, and have their time constraints specified using arbitrarily shaped TUFs. Given such a model, the scheduling objective is to maximize the sum of tasks' attained utilities.

Similar to DASA, the key concept of GUS is the Potential Utility Density (or PUD) metric. The PUD of a task simply measures the amount of value (or utility) that can be accrued per unit time by executing the task and the task(s) that it depends upon. The PUD therefore, essentially measures the "return of investment" for the task. Further, by considering the dependent tasks in computing the PUD, GUS explicitly accounts for task dependencies.

However, unlike LBESA or DASA, a TUF in GUS need not have a deadline time. Thus, GUS does not explicitly perform a feasibility test to determine whether or not a task's deadline can be satisfied. Rather, it implicitly checks the feasibility of a task set by ensuring the task set can yield positive utility. In fact, in the case of arbitrary TUFs, a deadline order or consequent deadline-based feasibility check may not be applicable. Another difference between GUS and DASA lies on GUS' more general resource access model.

GUS considers the "greedy" strategy, i.e., selecting a task and its dependents, whose execution will yield the maximum utility-increase per unit time (i.e., maximum PUD).

For step TUFs, the PUD of a task can be determined as the ratio of the sum of the utility of the task and all tasks in the dependency chain to their total execution time. However, it is difficult to similarly compute the PUD for non-step TUFs, because a task can yield different utilities at different completion times. Further, a task may not finish its execution at its optimal time and thereby can accrue sub-optimal (positive) utility.

To allow arbitrary TUFs, the philosophy of GUS is to regard the TUF as an application-specified "black box" in the following sense: The black box simply accepts a task completion time and returns an utility value. Therefore, to compute the PUD of a task T at time t, the algorithm considers the expected completion time(s) of the task (denoted as t_f), and possibly, its dependent tasks if they need to be completed to execute task T. The expected completion times are then "plugged" into individual task TUFs, to compute the sum of the expected utilities by executing the tasks. Once the expected utility U is computed, PUD of task T at time t is calculated as $U/(t_f - t)$.

Thus, when triggered at a scheduling event — which includes task arrival and departure, and resource request and release — GUS first computes the task PUDs. The largest PUD task and its dependents are then collected and inserted into a tentative schedule. The procedure is repeated until all tasks are added into the tentative schedule, or the execution of any of the remaining tasks will not produce any positive utility.

6. EXPERIMENTAL COMPARISONS

We implemented all four UA algorithms discussed here and the D^{over} algorithm [4]. D^{over} is optimal in the sense that it has the highest possible competitive ratio among all on-line algorithms for some restricted cases. All algorithms were implemented in a scheduling framework called *meta-scheduler* [6]. The meta-scheduler is an application-level framework for implementing UA scheduling algorithms on POSIX real-time operating systems (RTOSes), without modifying the underlying OS. We use QNX Neutrino 6.2 as the underlying RTOS.

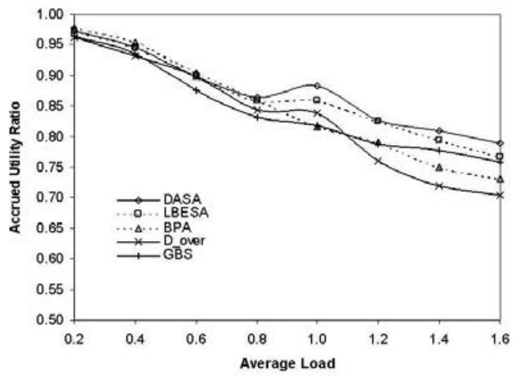


Figure 2: Performance of Algorithms Under Step TUFs and No Dependencies

Our first experiment considers tasks with step TUFs and no dependencies, which is applicable for all five algorithms. From Figure 2, we observe that the performance of the five 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. Further, we observe that GUS [1] performs worse than DASA and LBESA, but better than UPA and D^{over}.

Our second experiment compares the performance of DASA with that of GUS for tasks with step TUFs

[1]GUS was called "GBS" and UPA was called "BPA" then.

and dependencies. 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. Figure 3, however, shows that GUS actually performs better than DASA during light workload situations. Performance of the two schedulers are very close during overloaded situations. This is because in our particular implementation, the GUS scheduler incurs smaller overhead than the DASA scheduler.

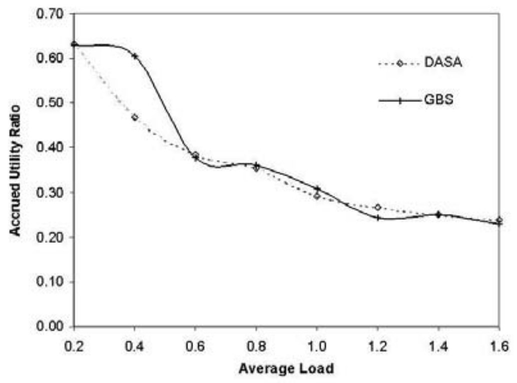


Figure 3: DASA versus GUS Under Step TUFs and Dependencies

Besides step TUFs, LBESA and GUS can also handle almost arbitrary TUFs, and GUS can further deal with dependent tasks. From Figure 4, we observe that LBESA and GBS exhibit close performance in terms of accrued utility ratio.

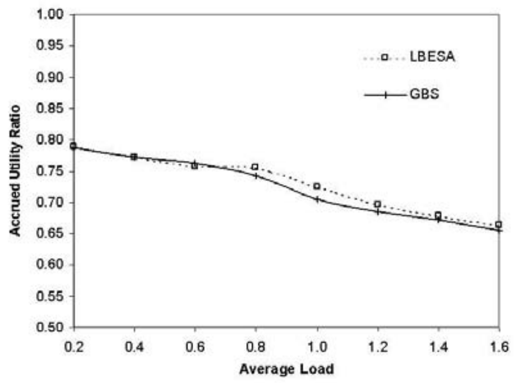


Figure 4: LBESA versus GUS Under Arbitrary TUFs and No Dependencies

7. CONCLUSIONS, CHALLENGES

Thus, we conclude that UA algorithms: (1) yield the timeliness-optimality of traditional real-time scheduling algorithms (e.g., DASA); (2) outperform traditional algorithms during overloads and thus has superior adaptability; (3) allow specification of non-uniform timeliness semantics (e.g., LBESA, UPA, GUS); and (4) has reasonable overhead even at an application-level.

There are many emerging challenges in TUF/UA research. A major challenge is to provide *assurances* on system behavior including that on individual and collective timeliness behavior. Examples include probability distributions on utility attained individually and collectively. None of the existing UA algorithms provide such assurances.

Other challenges include developing a methodology for designing TUFs. TUFs are currently empirically designed by application designers [9], [1]. An analytical foundation that provides assurances on UA-system behavior can help toward a systematic methodology for designing TUFs.

Existing UA algorithms are also restricted on the collective timeliness optimality of maximizing the sum of activities' attained utilities. Other timeliness optimality criteria are possible. Examples include maximizing the weighted sum of activities attained utilities and the number of activity completions before non-zero or non-negative utility times.

Finally, the intersection of UA scheduling and most other resource management problems such as memory management and power management are open.

8. REFERENCES

[1] R. Clark et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, volume 1586, pages 353–362, April 1999.

[2] R. K. Clark. *Scheduling Dependent Real-Time Activities.* PhD thesis, Carnegie Mellon University, 1990. CMU-CS-90-155.

[3] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *IEEE RTSS*, pages 112–122, December 1985.

[4] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for overloaded real-time systems. In *IEEE RTSS*, pages 290–299, December 1992.

[5] P. Li. *A Utility Accrual Scheduling Algorithm for Resource-Constrained Real-Time Activities.* Phd dissertation proposal, Virginia Tech, 2003. Available at `http://www.ee.vt.edu/~realtime/li-proposal03.pdf`.

[6] P. Li, B. Ravindran, et al. Choir: A real-time middleware architecture supporting benefit-based proactive resource allocation. In *IEEE ISORC*, pages 292–299, May 2003.

[7] J. W. S. Liu. *Real-Time Systems.* Prentice Hall, New Jersey, 2000.

[8] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, CMU, 1986. CMU-CS-86-134.

[9] D. P. Maynard et al. An example real-time command, control, and battle management application for alpha. Archons Project TR-88121, CMU, Dec. 1988.

[10] J. Wang and B. Ravindran. Time-utility function-driven switched ethernet: Packet scheduling algorithm, implementation, and feasibility analysis. *IEEE TPDS*, 15(2):119–133, February 2004.