Stochastic, Utility Accrual Real-Time Scheduling with Task-Level and System-Level Timeliness Assurances

Peng Li[‡], Hyeonjoong Cho[⋆], Binoy Ravindran[⋆], and E. Douglas Jensen[†]

[‡]Microsoft Corporation
Redmond, WA 98052, USA
pengli@microsoft.com

[⋆]ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{hjcho,binoy}@vt.edu

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

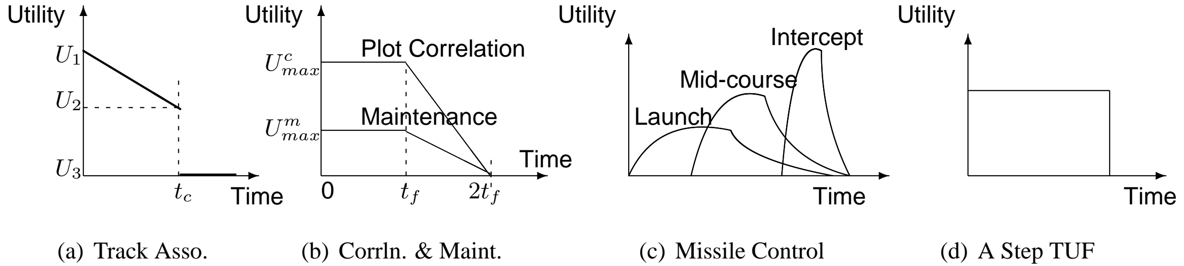
Abstract

Heuristic algorithms have enjoyed increasing interests and success in the context of Utility Accrual (UA) scheduling. However, few analytical results, such as bounds on task-level and system-level accrued utilities are known. In this paper, we propose the S-UA algorithm that can provide probabilistic bounds on task-level accrued utilities. Lower bound on system-level accrued utility ratio (AUR) is also derived and maximized by S-UA.

1 Introduction

Dynamic, adaptive, real-time embedded control systems at any level(s) of an enterprize—e.g., devices in the defense domain such as multi-mode phased array radars and battle management have emerged in many application domains. 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.

Jensen's time/utility functions [16] (or TUFs) allow the semantics of soft time constraints to be precisely specified. A TUF specifies the utility to the system resulting from the completion of an activity as a function of the its completion time. 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 [12] built by The MITRE Corporation and The Open Group (TOG); and (2) a coastal air defense system [23] built by General Dynamics (GD) and Carnegie Mellon University (CMU).



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

Figure 1. Example Timing Constraints Specified Using Time/Utility Functions

When timing constraints are expressed with TUFs, the scheduling optimality criteria are based on factors in terms of maximizing accrued utility from those activities—e.g., maximizing the sum, or the expected sum, of the activities' attained utilities. Such criteria are called *Utility Accrual* (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that consider UA criteria are called UA sequencing algorithms. Several UA scheduling algorithms are presented in the literature, such as [19, 22, 13, 18].

Regardless of the recent advance of UA scheduling theories and algorithms, most of the algorithms are heuristics and do not provide any performance assurance, such as bounds on task-level and system-level accrued utilities. The best known analytical result is the upper bound on competitive ratio established by Baruah et. al. [6]. However, this upper bound is only valid for tasks with step TUFs, where utilities are equal to tasks' computation times.

The problem of performance assurance is further complicated by the fact that systems of interests are highly dynamic and they can not be described using deterministic task models. For example, task execution times may be highly dependent on their operational environment and thus worst-case execution time (WCET) analysis is infeasible. Furthermore, many such systems are driven by external events and do not have known minimal inter-arrival times.

The aforementioned non-determinism naturally lends itself to a stochastic task model, where task execution times and inter-arrival times are modeled by random variables. Task may share resources that can only be exclusively accessed. Furthermore, an end-user is allowed to specify a set of Assurance Probability (AP). Give such models, we develop an algorithm, called S-UA that seeks to accrue the highest possible system-wide utility as well as providing performance assurance for each task with at least its specified assurance probability, i.e., an assured utility and corresponding assurance probability $p \geq AP$. Lower bound on system-level accrued utility ratio (AUR) is also derived and maximized by S-UA.

The rest of the paper is organized as follows. We first introduce the models and objective in Section 2. Section 3 provides an overview of the S-UA algorithm, including usage and basic design approach. In Section 4, we discuss details of the S-UA algorithm. Several important properties of the algorithm are established in Section 5. We then compare the performance of S-UA with several well-known heuristics

in Section 6. Finally, Sections 7 and 8 discuss related efforts and conclude this paper with its contributions and future work.

2 Models and Objectives

2.1 Task Model and Notations

We consider a system that includes a set of tasks $\mathbf{T} = \{T_1, T_2, ..., T_n\}$. A task T_i has a number of instances, denoted as $\{\tau_{i,1}, \tau_{i,2}, ...\}$ and these instances may be released periodically or aperiodically (without known minimal inter arrival time). An instance of a task is also called a *job*.

We model task execution times and inter-arrival times as random variables. To capture the behavior of these random variables, we assume that we have some knowledge of the their stochastic attributes. In the strongest case, we have the full knowledge of their distributions, which can be approximated by a measurement of histogram. In a weaker case, we know moments of task parameters, such as the first moment (mean) and the second moment (variance) of task execution times and inter-arrival times. These stochastic attributes of the random variables can be obtained either by on-line traffic measurement or by off-line traffic profiling.

Tasks may use shared resources guarded by semaphores. We assume that for any task T_i, the execution time of using any shared resource is upper bounded by B_i. This is because a critical section is usually small enough to allow Worst Case Execution Time (WCET) analysis.

2.2 Time/Utility Functions and Application QoS Requirements

A TUF for task T_i, denoted as $U_i(t)$ has an initial time I_i (a typical value of I_i is zero) and a termination time X_i. Before the initial time, the TUF is undefined; after the termination times, the TUF is always zero. Note that $\forall j, \tau_{i,j}$ have the same TUF. Thus, the TUF for job $\tau_{i,j}$ is defined on the interval of $[I_i + a_{i,j}, X_i + a_{i,j}]$, where $a_{i,j}$ is the arrival time of job $\tau_{i,j}$. Furthermore, we focus on non-negative, non-increasing TUFs in this work, as they enjoy broad applicability in many domains, such as multimedia, and command and control. However, this work can be extended to non-negative unimodal TUFs by postponing the execution of a task with a increasing TUF [22]. For a non-negative, non-increasing TUF $U_i(t), U_i(t) \geq 0, \forall t \in [I_i, X_i]$, and $\forall t_1, t_2 \in [I_i, X_i], t_1 < t_2, U_i(t_1) \geq U_i(t_2)$.

Besides a TUF, an end-user is also allowed to specify an acceptable *Assurance Probability* for each task T_i, denoted as AP_i. A high AP_i demands high assurance, which may yield a low assured utility, and vice versa. Thus, the S-UA algorithm allows an interactive, off-line negotiation procedure that can trade assured utility with assurance probability. We present the negotiation procedure in Section 3.1.

2.3 Optimization Objective

The objective of the S-UA algorithm is two-fold: (1) to maximize the sum of accrued utilities from all tasks, and (2) to accrue an assured utility U_i^A for each task T_i with at least its specified assurance probability AP_i, if possible.

More specifically, let $s_{i,j}$ be the sojourn time of an arbitrary instance of a task T_i. We should have $\Pr(u_{i,j} \geq U_i^A) \geq AP_i$, where $u_{i,j}$ is the accrued utility of job $\tau_{i,j}$, i.e., $u_{i,j} = U_i(s_{i,j})$. Subject to this assurance requirement, the algorithm seeks to maximize an important utility accrual criterion, called "Accrued Utility Ratio" (AUR) that is defined as the following:

$$AUR = \lim_{m\to\infty} \left[\sum_{i=1}^{n} \sum_{j=1}^{m} u_{i,j} \Big/ m \sum_{i=1}^{n} U_i^{\max} \right], \tag{2.1}$$

where U_i^{max} is the maximal possible utility accrued by task T_i.

Since the system is random in nature, the sojourn time of a job $s_{i,j}$ and the corresponding utility $u_{i,j}$ are also drawn from random variables. Let u_i be the random variable for the accrued utility of task T_i. The optimization criterion of AUR can be re-written as:

$$AUR = \sum_{i=1}^{n} \left[\lim_{m\to\infty} \left(\frac{1}{m} \sum_{j=1}^{m} u_{i,j} \right) \right] \Big/ \sum_{i=1}^{n} U_i^{\max} = \sum_{i=1}^{n} E\left(u_i\right) \Big/ \sum_{i=1}^{n} U_i^{\max} \tag{2.2}$$

Thus, the optimization problem essentially requires maximizing the sum of the expected utilities of all tasks. Note that the above design objective differs from those of previous algorithms in that it is also subject to the performance assurance requirement.

3 Overview of the S-UA algorithm

3.1 Usage and Programming Model

The S-UA algorithm can be used for either off-line or on-line assurance. For off-line assurance, a designer gathers information of all tasks and specifies a set of assurance probabilities. Given that, the algorithm returns a set of utilities $\{U_i^A\}$ that are guaranteed with at least their corresponding assurance probabilities. The designer can then examine the set of $\{U_i^A\}$. If the set of $\{U_i^A\}$ is not satisfactory, the designer needs to adjust assurance probabilities—e.g., lower assurance probabilities for some low utility tasks. The procedure of interactive off-line assurance is shown in Algorithm 1.

Another possible scenario of off-line assurance is to specify a set of $\{U_i^A, AP_i, i = 1 \text{ to } n\}$ as input to the S-UA algorithm. The algorithm then determines if the set of requirements can be satisfied, while it seeks to maximize system AUR. If the utilities assured by S-UA are higher than $\{U_i^A\}$, then clearly the

Input : a set of tasks $\mathbf{T} = \{T_1, ..., T_n\}$ with non-increasing TUFs, denoted as $\{U_1, U_2, ..., U_n\}$
Output : a set of assurance probability $\{AP_i\}$ and corresponding utilities $\{U_i^A\}$
repeat
    The designer specifies a set of $\{AP_i\}$;
    S-UA computes the set of assured utilities $\{U_i^A\}$;
until $\{AP_i\}$ *and* $\{U_i^A\}$ *are satisfactory*;

Algorithm 1: Interactive Off-line Assurance

requirements can be satisfied. Thus, given a set of $\{AP_i, i = 1 \text{ to } n\}$, the problem of determining the highest utility that can be assured is the foundation for interactive assurance and negotiation procedure.

The algorithm can also be used for on-line assurance. To support on-line assurance, the S-UA algorithm needs a *Traffic Monitor* component. The *Traffic Monitor* component monitors several events that can trigger the S-UA algorithm: (1) change of task traffic pattern; (2) arrival of a new task; and (3) departure of an existing task. The traffic pattern of interest includes patterns of task execution times and task inter-arrival times. For example, the *Traffic Monitor* can measure and monitor the means and variances of task execution times and inter-arrivals times, which can be used by S-UA.

The S-UA algorithm for on-line assurance can start with *estimated* means and variances of task execution times and inter-arrival times. Once the Traffic Monitor detects the deviation of the estimated pattern from the actual traffic pattern, or the traffic pattern changes at run time, it triggers the S-UA resource allocation algorithm. The S-UA resource allocation algorithm in turn recomputes resource allocation for each task.

3.2 Algorithm Design Approach

Our basic approach to achieve utility assurance is to bound tasks' sojourn times. Once tasks' sojourn times are upper-bounded, their accrued utilities are also lower-bounded, due to the non-increasing nature of TUFs assumed in this paper.

This problem is similar to that in classical deadline-based scheduling theories such as EDF [21], where the goal is to satisfy all task deadlines. However, the difference is that the upper bounds on tasks' sojourn times are not predefined. Rather, the algorithm seeks to bound tasks' sojourn times as short as possible so that the assured utilities can be as high as possible. In the meanwhile, short sojourn time bounds may not be achievable as they may overload the CPU. Thus, the sojourn times bound in this work (similar to "deadline" in EDF) are allocated by the S-UA algorithm in a manner consistent with our optimization goal and constraints. Another major difference is that in this work, task parameters, e.g., execution times and inter-arrival times are stochastically specified. Therefore, unlike the deterministic case, there is *always* a probability of task overrun and overload, though the probability may vary.

To derive the upper bounds on tasks' sojourn times, we consider a "server approach" where each task is associated with a "server." A server carries and manages the portion of CPU bandwidth for a task, determined by a CPU budget and a period. The server approach has been used for handling aperiodic tasks in real-time systems (see [9] for examples of aperiodic server algorithms, such as Sporadic Server and Polling Server). However, our rationale for considering the server approach is not to minimize the *average* response time of aperiodic tasks. Rather, the goal is to utilize the load isolation capability of the server approach, i.e., misbehavior of one task does not affect other tasks. This capability is particularly important in an environment of possible task overrun and overload.

The aforementioned server approach has been used for handling task overrun and overload in the literatures. For example, in [14], Gardner and Liu use a server to schedule portions of tasks that have exceeded their allocated CPU times. In [1, 3, 10], Abeni and Buttazzo proposed an algorithm for scheduling soft real-time tasks in a unpredictable environment, called Constant Bandwidth Server (CBS). In its essence, the CBS algorithm is a way to assign deadlines to task instances (i.e., jobs) so that the bandwidth consumed by a task never exceeds its allocated value. Once jobs are assigned proper deadlines by the CBS algorithm, they are simply scheduled by an EDF scheduler. The CBS approach differs from the previously proposed Total Bandwidth Server algorithm (TBS) in that TBS cannot isolate the effect of a misbehaved job, i.e., jobs using more processor time than their declared WCETs.

In this work, we use the CBS algorithm due to its simplicity in implementation and load isolation capability. Thus, each task T_i is assigned to a CBS server with a CPU budget of C_i and a period of D_i. Since D_i is also the desired sojourn time bound, we call it "critical time" of task T_i. Having stated this server approach, two related problems remain unsolved.

1. How to allocate CPU bandwidth to each task so that we can achieve the highest possible AUR and performance assurance; and
2. Given a set of CBS servers allocated to a set of tasks, how to compute the assurance probabilities and the assured utilities.

We present a solution to the first problem in Section 4. To solve the second problem, we first consider a degenerated problem, where each task has deterministic parameters, i.e., deterministic worst case execution times and minimal inter-arrival times. As such, schedulability of the degenerated problem can be analyzed by feasibility conditions similar to that of EDF algorithm [2].

In [2], the authors model each CBS server as a queueing system that serves instances of a task T_i. Assuming FIFO policy on each CBS server, they also present the schedulability analysis for two special cases: (a) constant execution times and random inter-arrival times, and (2) random execution times and constant inter-arrival times. However, schedulability analysis for the general case and under non-FIFO

policy is not available.

In this work, we propose a simple method to analyze the schedulability of a set of CBS servers serving stochastic tasks. Our method is based on the following observation. If a task T_i's execution time is less than its allocated CPU budget C_i and T_i's inter-arrival time is greater than D_i, then its sojourn time is guaranteed to be bounded by D_i. Since task parameters are random variables, there is a probability that the task's parameters violate its allocated CPU budget and minimal inter-arrival time. For such a case, T_i's sojourn time *may* not be bounded by the server period D_i. Note that conforming with the allocated CPU budget and minimal inter-arrival time is a *sufficient*, but *not necessary* condition to guarantee the sojourn time bound. That is, even if a task violates its allocated CPU bandwidth, it still *may* satisfy its sojourn time bound. However, deriving a *sufficient* and *necessary* condition is intractable. Thus, we decide to allocate resources in a way that satisfies the sufficient condition. Furthermore, conforming a CPU bandwidth C_i/D_i is *not* equivalent to conforming the allocated C_i and D_i, respectively. A counter example is a task having large execution time as well as large inter-arrival time. Still, it may only require a small fraction of CPU bandwidth. If that is the case, the task's execution time itself may be much larger than the task's desired sojourn time bound.

Using the aforementioned sufficient condition, we first allocate a portion of CPU bandwidth to each task, subject to the 100% CPU utilization bound of the CBS algorithm. For each task T_i, the probability of violating its allocated CPU budget and minimal inter-arrival time is then calculated and compared against the user-specified assurance probability AP_i. If the calculated probability is larger than or equal to AP_i, then task T_i is guaranteed to accrue at least the utility of $U_i(D_i)$ with the probability of AP_i. Otherwise, the user is advised to loosen his assurance probability requirements. This procedure is repeated until the user is satisfied. We elaborate the interactive negotiation procedure in Section 3.1.

4 The S-UA Algorithm

4.1 Problem Formulation

The optimization problem introduced in Section 2.3 can be stated as the following:

$$\text{Objective: } \max AUR = \sum_{i=1}^{n} E(u_i) \Big/ \sum_{i=1}^{n} U_i^{\max}$$
$$\text{Subject to: } \Pr(u_i \geq U_i^A) \geq AP_i, \forall i\text{=1 to } n$$

This form is difficult to analyze, as it does not reflect the variables being controlled in the resource allocation procedure. That is, neither the objective function nor the constraints have incorporated the CPU budget C_i and critical time D_i for each task T_i. To do that, we use the following lemmas to establish the relationship between the optimization objective and controlled resource allocation variables.

Lemma 1. *If a task T_i is guaranteed to accrue utility of U_i^A with the probability of at least AP_i, then the mean of its accrued utility, i.e., $E(u_i)$ is at least $U_i^A \times AP_i$.*

Proof. Let $p_i(u)$ be the probability density of task T_i's accrued utility. By the definition of $E(u_i)$ and the assumption of non-negative TUFs, we have:

$$E(u_i) = \int_0^{+\infty} u p_i(u) du = \int_0^{U_i^A} u p_i(u) du + \int_{U_i^A}^{+\infty} u p_i(u) du$$
$$\geq \int_{U_i^A}^{+\infty} u p_i(u) du \geq U_i^A \times \int_{U_i^A}^{+\infty} p_i(u) du$$

Since $\int_{U_i^A}^{+\infty} p_i(u) du$ is the probability that u_i is greater than U_i^A, we have $\int_{U_i^A}^{+\infty} p_i(u) du \geq AP_i$. The lemma thus follows. □

Proposition 2. *The system wide accrued utility ratio (AUR) is lower bounded by $\sum_i \left(U_i^A \times AP_i\right) \Big/ \sum_i U_i^{max}$.*

Proof. By Lemma 1, system-level $AUR = \frac{\sum_i E(u_i)}{\sum_i U_i^{max}} \geq \sum_i \frac{U_i^A \times AP_i}{\sum_i U_i^{max}}$. □

By Proposition 2, one approach to maximize system-wide AUR is to maximize its lower bound, which requires probabilistic bounds on task-level utilities. Notice that task-level probabilistic guarantee is subject to two conditions: (1) the CPU is not overloaded by the output from the S-UA resource allocation, and (2) tasks do not violate their allocated CPU budget times and minimal inter-arrival times. The feasibility of the resource allocation (condition 1) can be verified using the condition presented in [2]. Since task parameters are random variables, the second condition can only be *probabilistically* satisfied. This probability is calculated as $\Pr(T_i \text{ complies with its } C_i \text{ and } D_i) = \Pr(c_i \leq C_i) \times \Pr(d_i \geq D_i)$. Therefore, satisfying condition 2 requires:

$$\Pr(c_i \leq C_i) \times \Pr(d_i \geq D_i) \geq AP_i \tag{4.1}$$

. The above inequality has two controlled variables C_i and D_i. However, a closer look at the relationship between C_i and D_i reveals that, given a D_i, a minimal C_i that satisfies the inequality can be computed.

Assume that the means and variances of execution time and inter arrival time of each task are finite and known. Given a D_i, one can compute $\Pr(d_i \geq D_i)$ by a version of Chebyshev's inequality. Then, the minimal required C_i for the given D_i can be computed as the following:

$$C_i \geq E(c_i) + \sqrt{\frac{Var(c_i)}{1/M_i - 1}}, \tag{4.2}$$

where $M_i = AP_i \times \frac{Var(d_i) + (D_i - E(d_i))^2}{Var(d_i)}$. Note that the computed C_i is the *minimal* required CPU budget time. A larger CPU budget helps to improve $\Pr(c_i \leq C_i)$, but will unnecessarily increase the bandwidth

requirement for task T_i. Thus, given a D_i, the algorithm simply allocates the minimal required C_i from Equation 4.2 to task T_i.

Once the two conditions are probabilistically satisfied, tasks' sojourn times are also probabilistically bounded. Thus, we can transform the optimization problem to the following form:

$$\text{Objective: } \max \lambda = \sum_i (U_i(D_i) \times AP_i)$$
$$\text{Subject to:}$$
$$\begin{cases} C_i \geq E(c_i) + \sqrt{\frac{Var(c_i)}{1/M_i - 1}}, M_i = AP_i \times \frac{Var(d_i) + (D_i - E(d_i))^2}{Var(d_i)}, \forall i = 1 \text{ to } n \\ \sum_{k=1}^{i} \frac{C_k}{D_k} + \frac{B_i}{D_i} \leq 1, \ \forall i = 1 \text{ to } n \end{cases} \tag{4.3}$$

The problem of finding the optimal solution $\{D_i, \forall i\}$ is clearly a non-linear optimization problem, as the constraints are non-linear.

4.2 A Randomized Hill Climbing Search Strategy

To solve such an optimization problem, one approach is to use so called "Hill Climbing" (HLC) search strategy. The HLC strategy is a greedy neighborhood search strategy in the sense that it *always* selects the best "neighbor solution" during each iteration. During each iteration, a HLC strategy may reduce the critical time of a task T_k by a ΔD, if doing so can yield the maximal increase of the objective function λ.

The problem with the simple HLC heuristic is that the search procedure may be trapped in a local optimal point. For example, consider a system of two tasks, both having linearly decreasing TUFs, and the same termination time and AP_i. Assume that $U_1^{max} > U_2^{max}$. Then, during each iteration, $AP_1 \times (U_1(D_1 - \Delta D) - U_1(D_1))$ is *always* greater than $AP_2 \times (U_2(D_2 - \Delta D) - U_2(D_2))$. This inequality eventually leads to the resource allocation result of allocating all CPU bandwidth to task T_1 and completely starving task T_2, which may not be good.

To avoid such a situation, we use a randomized Hill Climbing (RHLC) search strategy. The RHLC search strategy differs the simple HLC strategy in that RHLC *randomly* moves to a "neighbor" solution during each iteration, which may not be the best neighbor solution, or even a worse neighbor solution. Furthermore, the probability of moving to neighbor solution depends on the quality of the solution. In general, a better neighbor solution has a higher probability being selected, and a worse neighbor solution has a lower probability.

Our RHLC strategy is similar to Simulated Annealing (SA) [17] in the sense that both algorithms make random moves to neighbor solutions. However, the SA algorithm contains two nested loops, whereas the RHLC algorithm only has one loop. Thus, our RHLC algorithm is faster than SA in terms of complexity, which is important as the S-UA algorithm may be invoked at run-time to deal with change of traffic

patterns.

The concepts of "temperature" and "cooling" in simulated annealing allow control of diversity at different stages of the search procedure. Initially, the temperature is high enough to allow diversity. With the reduced temperature, the probability of moving to a bad neighbor solution also decreases, as the system is approaching a stable state. For the same purpose, our RHLC heuristic also incorporates a similar cooling process.

Before we describe the resource allocation algorithm in details, we introduce the definitions of "neighbor solutions" and "valid solutions" that are used in the algorithm.

Definition 3. *A valid solution* $\nu = \{D_1, ..., D_n\}$ *satisfies the constraint as defined in inequality 4.3:*

$$\begin{cases} C_i \geq E(c_i) + \sqrt{\frac{Var(c_i)}{1/M_i - 1}}, M_i = AP_i \times \frac{Var(d_i) + (D_i - E(d_i))^2}{Var(d_i)}, \forall i = 1 \textit{ to } n \\ \sum_{k=1}^{i} \frac{C_k}{D_k} + \frac{B_i}{D_i} \leq 1, \ \forall i = 1 \textit{ to } n \end{cases}$$

Given a solution ν and a set of tasks ordered by the critical times as in ν, Algorithm 2 verifies if ν is a valid solution.

Input : a solution ν and a task set $\mathbf{T}$ ordered by increasing critical times defined in ν
Output : true if ν is a valid solution; false otherwise
for $i = 1$ *to* $|\mathbf{T}|$ **do**
  Calculate T_i's CPU budget: $C_i \leftarrow E(c_i) + \sqrt{\frac{Var(c_i)}{1/M_i - 1}}$, where $M_i \leftarrow AP_i \times \frac{Var(d_i) + (D_i - E(d_i))^2}{Var(d_i)}$;
  if $\sum_{k=1}^{i} (\frac{C_k}{D_k}) + \frac{B_i}{D_i} > 1$ **then**
    return false;
return true;

Algorithm 2: `isValid()` Function

Definition 4. *The set of neighbor solutions of a valid solution* ν *is defined as* $\{\nu_i^+ = \nu + \{\overbrace{0, ..., 0}^{1..(i-1)}, \Delta D, \overbrace{0, ..., 0}^{(i+1)...n}\}$, $\nu_i^- = \nu - \{\overbrace{0, ..., 0}^{1..(i-1)}, \Delta D, \overbrace{0, ..., 0}^{(i+1)...n}\}, \forall i = \textit{1 to } n\}$.

The critical allocation algorithm using RHLC heuristic is described in Algorithm 3. The algorithm starts with an initial solution ν^0. During each iteration, it finds the set of valid neighbor solutions of the current solution ν by invoking function `findNeighbors()` (Algorithm 4). Improvement of objective function $\lambda()$ is then calculated for each valid neighbor solution. Similar to simulated annealing, we use an exponential function as the basis for calculating the probabilities. Let ϕ_i be the improvement of $\lambda()$ function by moving to a valid neighbor solution ν_i, i.e., $\phi_i = \lambda(\nu_i) - \lambda(\nu)$ and let $\phi_m = max\{\phi_i, \forall i\}$. The probability of moving to a neighbor solution ν_k at temperature t is computed as $p_k = \gamma_k / \sum_i \gamma_i$, where $\gamma_i = exp(-(\phi_m - \phi_i)/t)$.

Though RHLC heuristic can reduce the possibility of being trapped in a local optimal solution, it can still be stuck on a plateau in the search space. A plateau in the search space corresponds to a solution ν with all $D_i, \forall i$ on segments of constant TUFs. This problem can be solved by simulated annealing with high computation cost. In this work, we augment the RHLC heuristic with tolerance of movements on a plateau. In case that the highest improvement of solution quality is less than a threshold δ, a random movement to a neighbor solution is still conducted until K consecutive no-improvement moves happen.

Input : a set of tasks $\mathbf{T} = \{T_1, ..., T_n\}$ with non-increasing TUFs, denoted as $\{U_1, U_2, ..., U_n\}$ and a set of minimal assurance probability $\{AP_1, AP_2, ..., AP_n\}$
Output : critical time and CPU time budget for each task, denoted as D_i and C_i
Parameters : $\nu^0, t_0, K, \delta, \alpha, \Delta D$;
Set initial solution: $\nu \leftarrow \nu^0$;
Set initial "temperature": $t \leftarrow t_0$;
Set the number of no-improvement moves: $k \leftarrow 0$;
repeat
  Find the set of valid neighbor solutions for ν: $NB \leftarrow \texttt{findNeighbors}(\nu, \mathbf{T})$;
  if $|NB| \geq 1$ **then**
    for *each valid solution* $\nu_i \in NB$ **do**
      Calculate the improvement of objective function: $\phi_i \leftarrow \lambda(\nu_i) - \lambda(\nu)$;
    Let $\phi_m \leftarrow max\{\phi_i, \forall i\}$;
    Let $\gamma_i \leftarrow exp^{-(\phi_m - \phi_i)/t}$;
    Randomly select a solution ν_k with the probability of $p_k \leftarrow \gamma_k / \sum_i \gamma_i$;
    Update the current solution: $\nu \leftarrow \nu_k$;
    Update temperature: $t \leftarrow t \times \alpha$;
    if $\phi_m \leq \delta$ **then**
      Update the number of no-improvement moves: $k \leftarrow k + 1$;
    else
      Reset k: $k \leftarrow 0$;
  else
    break;
until *no more valid neighbor solutions or* $k \geq K$;

Algorithm 3: Critical Time Allocation Algorithm

4.3 Tuning Parameters for Resource Allocation

This section presents the selection of the set of parameters, i.e., $t_0, \nu^0, \Delta D, \delta, K$, and α for Algorithm 3. Our basic approach is to use TUFs to drive the selection procedure.

1. The single requirement for ν^0 is that it is a valid solution, satisfying constraint 4.3. Intuitively, a solution with minimal bandwidth consumption $C_i/D_i, \forall i$ has a good chance of being a valid solution, though not necessarily. Furthermore, a closer look at the relationship between C_i and D_i reveals that C_i is a convex function of D_i and C_i/D_i is minimized when D_i is slightly larger than $E(d_i)$. Thus, in this work, we choose $\nu^0 = \{D_i = E(d_i), \forall i\}$;

```
Input    : a valid solution ν and a task set T
Output   : the set of valid neighbor solutions for ν, denoted as NB
Sort T by increasing critical times as defined in ν;
for i=1 to n do
    Generate a neighbor solution: ν_i^+ = ν + {0,...,0 (1..(i−1)), ΔD, 0,...,0 ((i+1)...n)};
    Let T^+ be the task set ordered by critical times as defined in ν_i^+;
    if isValid(ν_i^+, T^+) then
        Add ν_i^+ into NB: NB ← NB ⋂ {ν_i^+};
    Generate a neighbor solution: ν_i^- = ν − {0,...,0 (1..(i−1)), ΔD, 0,...,0 ((i+1)...n)};
    Let T^- be the task set ordered by critical times as defined in ν_i^-;
    if isValid(ν_i^-, T^-) then
        Add ν_i^- into NB: NB ← NB ⋂ {ν_i^-};
return NB;
```

Algorithm 4: `findNeighbors()` Function

2. ΔD should have comparable time scale with the TUFs. In the meanwhile, it determines the granularity of resource allocation. For such purpose, we choose $\Delta D = max\{X_i, \forall i\}/m$, where $m = 1000$ in out experiments;
3. δ should be related the scope of possible utilities. Thus, we choose $\delta = 5\% \times max\{U_i^{max}, \forall i\}$;
4. The parameter K controls the tolerance of plateau in the search space. A large K tends to explore far neighbor solutions from a plateau solution, and vice versa. Assume tolerating a constant TUF segment of up to the length of $max\{X_i, \forall i\}/2$ is desired. This assumption yields $K = 1/2 \times max\{X_i, \forall i\}/\Delta D$. In the case that $\Delta D = max\{X_i, \forall i\}/1000$, K is computed as 500;
5. The typical range of α is $0.8 \sim 0.99$. In this work, we choose an empirical value of 0.9;
6. Empirical results suggest that t_0 should allow enough diversity in searching the solution space. That is, the initial acceptance ratio for the "best solution" is reasonable, e.g., 0.8. A too high acceptance ratio may reduce the diversity of solutions and thus is not good. In this work, we consider the acceptance ratio at t_0 for a solution with ϕ_m utility improvement is 0.8. Furthermore, we consider all other solutions can not improve the system-wide utility, i.e., all other solutions move on a segment of constant TUF. Therefore, the probability of moving to the best neighbor solution is calculated as $p_m = 1 \Big/ \left[(n-1) \times exp(-\frac{\phi_m}{t_0}) + 1\right] = 0.8$, which yields $t_0 = \phi_m / \left[1 - ln\frac{0.25}{n-1}\right]$.
   Now, the problem is how to compute the maximal utility increase ϕ_m. In general, ϕ_m depends on the shapes of TUF segments along which the solutions, and the step length of ΔD. Therefore, ϕ_m may assume different values at different iterations of the algorithm. To approximate ϕ_m, we use "pseudo slopes" of TUFs. A "pseudo slope" of a non-increasing TUF measures the rate of utility loss, i.e., $pseudo_slope(U_i) = U_i^{max}/X_i$.

Observe that, reducing the critical time of a task T_i that has the highest pseudo slope tends to yield the highest ϕ_i. Therefore, we can approximate ϕ_m as the following:

$\phi_m \approx \phi'_m = max\{U_i^{max}/X_i, \forall i\} \times \Delta D.$

4.4 Scheduling

The performance assurance provided by S-UA relies on the fact that task parameters can probabilistically comply with their allocated values. It also makes the implicit assumption that if a job $\tau_{i,j}$ conforms to its allocated CPU budget and its inter-arrival time, i.e., $c_{i,j} \leq C_i$ and $a_{i,j} - a_{i,j-1} \geq D_i$, it's sojourn time is guaranteed to be not more than D_i. This assumption requires a "memoryless" property of the scheduling algorithm. That is, the performance assurance for a job $\tau_{i,j}$ only depends on $c_{i,j}$ and $(a_{i,j} - a_{i,j-1})$. A previously arrived job that violates its allocated CPU budget and/or inter-arrival time should not affect the performance assurance for $\tau_{i,j}$.

Based on the aforementioned observation, we extend the original CBS scheduling algorithm to satisfy this "memoryless" property. More specifically, if a job arrives with an inter-arrival time shorter than D_i, it is enqueued in a queue of pending jobs. Our rational for not dropping this incoming job is because its performance assurance can still be satisfied, though not guaranteed, for several reasons, e.g., the execution time of the previous jobs are shorter than C_i. This rational is similar to that of Resource Reclaiming in real-time computing literatures. On the other hand, if job $\tau_{i,j}$ arrives with an inter-arrival time $a_{i,j} - a_{i,j-1} \geq D_i$, its performance assurance should be satisfied, provided $c_{i,j} \leq C_i$. Consequently, the execution of the currently running job $\tau_{i,r}$ should not interfere with $\tau_{i,j}$'s performance assurance. Thus, $\tau_{i,r}$ should be preempted by $\tau_{i,j}$. In fact, it can be seen that $\tau_{i,r}$'s sojourn time is greater than D_i at the current time t, as $t = a_{i,j}$ and $a_{i,j} - a_{i,r} \geq D_i$. Though job $\tau_{i,r}$'s performance assurance cannot be satisfied due to its non-conforming parameters, the algorithm still queues it in the pending job queue. By doing so, $\tau_{i,r}$ may accrue some utility later and thus improve the system-wide accrued utility ratio.

We summarize the scheduling rules as the following:

1. A CBS server for task T_i is associated with a budget c_s and an ordered pair (C_i, D_i), where C_i is the maximal budget and D_i is the period of the server. We define server bandwidth $U_s = C_i/D_i$;
2. Every served job $\tau_{i,j}$ is assigned a dynamic deadline $d_{i,j}$ that is equal to the current server deadline $d_{s,k}$;
3. Whenever a served job executes, the budget c_s is decreased by the same amount;
4. When $c_s = 0$, the server budget is recharged to C_i and the server deadline is set to $d_{s,k+1} = d_{s,k} + D_i$;
5. When a job $\tau_{i,j}$ arrives and the server is serving another job $\tau_{i,k}$, where $k < j$ (i.e., is active), the

new job can be managed in two different ways: (1) if $a_{i,j} - a_{i,j-1} < D_i$, $\tau_{i,j}$ is enqueued in a queue of pending jobs that are serviced by FIFO policy, or (2) if $a_{i,j} - a_{i,j-1} \geq D_i$, the currently running job $\tau_{i,r}$ should be preempted by $\tau_{i,j}$;

6. When a job $\tau_{i,j}$ arrives and the server is idle, if $c_s \geq (d_{s,k} - r_{i,j})U_s$ the server generates a deadline $d_{s,k+1} = r_{i,j} + D_i$ and c_s is recharged to C_i, otherwise, $\tau_{i,j}$ is serviced with current server deadline and c_s; and

7. When a job finishes at time t, a pending job $\tau_{i,m}$, if any is selected and served using the current budget and deadline. If there are several pending jobs, $\tau_{i,m}$ should be the one that satisfies $a_{i,m} + D_i > t$ and has the earliest arrival time among all pending jobs.

5 Properties of S-UA Algorithm

In this section, we present a set of important properties of the proposed S-UA algorithm.

Proposition 5. *The S-UA algorithm is deadlock free.*

This is obvious due to the usage of stack resource policy.

Proposition 6. *Suppose the task model degenerates to a sporadic task set with worst-case execution time C_i and minimal inter-arrival time $D_i = X_i$. If $\sum_{k=1}^{i} C_k/D_k + B_i/D_i \leq 1, \forall i = 1$ to n, then S-UA can achieve at least the same lower bound on system-level AUR as that of an EDF scheduler with relative deadlines equal to D_i.*

Proof. For such a task model, the S-UA critical allocation algorithm starts with $\nu = \{X_i, \forall i\}$. Regardless of the random moves during each iteration, the algorithm always moves to a valid neighbor solution $\nu' = \{D'_i, \forall i\}$, where $D'_i \leq X_i$ and can be guaranteed. On the other hand, the feasibility condition of the EDF scheduler only guarantees a sojourn time bound of X_i. By Proposition 2, the lower bound on system AUR achieved by S-UA is at least the same as that of EDF. □

Corollary 7. *For the same task model as in Proposition 6 and step TUFs, S-UA can achieve 100% system-level AUR if $\sum_{k=1}^{i} C_k/D_k + B_i/D_i \leq 1, \forall i = 1$ to n.*

Proof. For step TUFs, an EDF can achieve 100% system-level AUR. Since S-UA is at least as good as EDF, it can also achieve 100% system AUR. □

Proposition 8. *Suppose the termination time of each TUF is less than or equal to the task's average inter-arrival time, i.e., $X_i \leq E(d_i), \forall i$. If $\sum\limits_{i=1}^{n} \frac{E(c_i)}{E(d_i)} > 1$, then there is no valid solution for the constraint defined in Equation 4.3.*

Table 1. Parameters for Critical Time Allocation Algorithm

Parameter	ΔD	t_0	α	K
Value	$0.1\% \times max\{X_i, \forall i\}$	200	0.9	1000

Proof. For any task T_i, assuring any positive utility requires $D_i < X_i \leq E(d_i)$, because $U_i(t) = 0, \forall t \geq X_i$. Furthermore, by inequality 4.2, $C_i \geq E(c_i)$. These two inequalities result in $\frac{C_i}{D_i} > \frac{E(c_i)}{E(d_i)}, \forall i$. Thus, for any solution ν, we have:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} + \frac{B_n}{D_n} > \sum_{i=1}^{n} \frac{E(c_i)}{E(d_i)} + \frac{B_n}{D_n} > 1. \tag{5.1}$$

This inequality leads to the conclusion that ν is not a valid solution. □

6 Performance Evaluation

The previous sections have analytically established the sufficient condition for satisfying user-specified probabilistic assurance, and have presented the S-UA algorithm. However, the performance of the S-UA algorithm may be sacrificed to meet probabilistic assurance. In addition, the run-time overhead of S-UA cannot be obtained through analysis either. Thus, we implemented the S-UA algorithm in the *meta scheduler* scheduling framework [20]. Performance of S-UA and several heuristic UA scheduling algorithms are measured and compared. Empirical values of S-UA run-time overhead is also provided in this section.

6.1 Resource Allocation Overhead

Recall that the S-UA algorithm can be used either off-line, e.g., at design time or on-line at run-time. To do that, overhead of the S-UA algorithm should be as small as possible. Furthermore, overhead of the critical time allocation algorithm (Algorithm 3) clearly dominates that of run-time scheduling (e.g., CBS deadline calculation and EDF scheduling) and thus constitutes the major part of the S-UA algorithm. In fact, our choice of a randomized hill climbing search strategy, as opposed to a full-fledged simulated annealing is motivated by the need of low run-time overhead.

We conducted the overhead measurement on a 450 MHz Pentium II PC, running an implementation of the *meta scheduler* on top of QNX Neutrino 6.2 operating system. Several important parameters for running the resource allocation algorithm are shown in Table 1. All tasks have linearly decreasing TUFs in our experiments.

As shown in Figure 2, the overhead of running Algorithm 3 is roughly tens of milliseconds to a few hundred milliseconds for less than ten tasks, which is considered to be light-weight. Though we do not have measurements for larger problems, we expect that the overhead is still acceptable. This is because S-UA is expected to execute very infrequently. In the event that all task traffic patterns are accurately specified at design time, the S-UA resource allocation algorithm only needs to be invoked at design time. Furthermore, time requirements in systems of interests usually span from milliseconds to seconds, or even hours (e.g., in a several-hour flight mission).

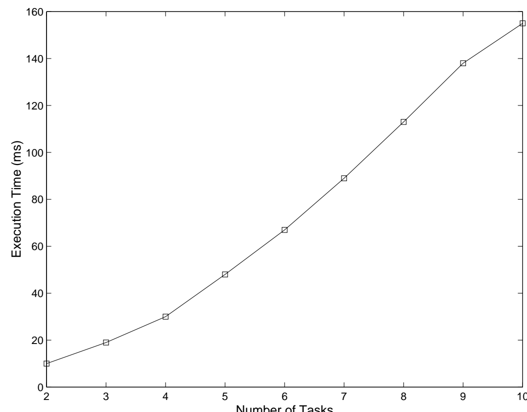


Figure 2. Overhead of Resource Allocation

6.2 Performance Comparison

Performance comparison is conducted between S-UA and a set of heuristic algorithms, which are known to outperform other heuristics for the models they apply. These heuristics include Dependent Activity Scheduling Algorithm (DASA) [13], Generic Utility Scheduling (GUS) [19], and Utility-accrual Packet Scheduling (UPA) [26].

Each experiment contains five tasks with downward step, linear, or right half parabolic TUFs. The maximal utilities of the TUFs vary from 30 to 200, and the termination times fall between 1.4 s to 2.4 s, which are also assumed to be the same as average inter-arrival times. The assurance probabilities start with 0.5 at low load and drop to 0.3 at high load, as a high assurance probability is not possible at high load.

Our first experiment compares the performance of all four algorithms with downward step TUFs and no resource dependencies. As shown in Figure 3, if the load is not too heavy, i.e., average load is less than 0.8, all four algorithms have very close performance. However, when load is heavy, i.e., the average load exceeds 1.0, GUS and DASA exhibit the best performance. This result is consistent with the observation in [19]. The GUS algorithm is a greedy algorithm, and thus has good performance in general. On the contrary, the DASA algorithm assumes and explores the shape information of downward step TUFs. Therefore, DASA has good performance for step TUFs, but not for other functions. The S-UA implicitly uses the shape information of TUFs in critical time allocation.

Based on this observation, we conjecture that S-UA performs better than DASA and UPA for non-step functions. This conjecture is supported by our second experiment with linear TUFs and no dependencies,

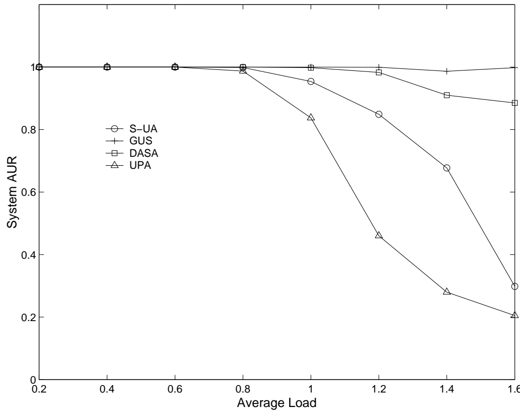


Figure 3. Step TUFs, No Dependencies

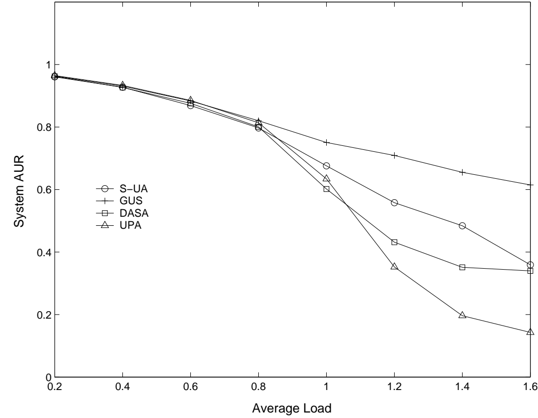


Figure 4. Linear TUFs, No Dependencies

as shown in Figure 4. Observe that all algorithms have lower AURs than their counterpart values for step TUFs in Figure 3, because linear TUFs start decreasing at their initial times. Also observe that S-UA now outperforms DASA for more than 10% system AUR at load 1.2 and 1.4.

We then conduct experiments with a mix set of step, linear, and parabolic TUFs. Since we observe that GUS is always the best for step and non-step TUFs, it should also be the best for tasks with mixed TUFs. Similarly, UPA should have the worst performance. In addition, DASA is known to be better than S-UA for step TUFs, and is worse than S-UA for non-step TUFs. As the end result, DASA and S-UA should have close performance for mixed TUFs. These expectations are confirmed by the experimental results in Figure 5.

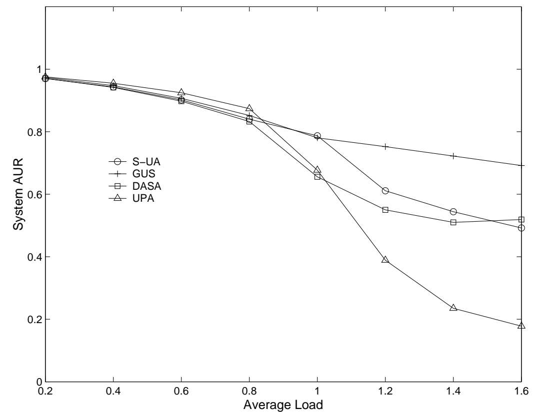


Figure 5. Heterogeneous TUFs, No dependencies

Finally, we also conduct experiments for tasks with resource dependencies. Algorithms that can deal with resource dependencies, such as S-UA and DASA are compared. The experimental results closely mimic those for no-dependency case, and thus are omitted here.

7 Related Work

Majority of the existing UA algorithms consider tasks with downward step TUFs and no dependencies, which are natural extension to the classical deadline time constraints. During underload situations, the classical EDF algorithm lends itself to optimal scheduling. Thus, most of the scheduling algorithms are specifically designed to handle overloads. For example, [24, 8, 4] are equivalent to EDF during

underloads. If the processor is overloaded, the algorithms reject one or more least value-density tasks in the ready queue until the remaining task set becomes feasible.

Recently, several UA scheduling algorithms have been developed to handle non-step TUFs and to allow resource dependencies. Locke's algorithm, called Best Effort (BE) scheduling algorithm [22] is the first known UA scheduling algorithm that can deal with almost arbitrary TUFs, but without resource dependencies. Interestingly, Locke's BE algorithm allows stochastic task execution times and arrivals, but no performance assurance is provided. Clark's DASA algorithm [13] allows resource dependencies but only step TUFs. Later, in [11] and [26], the authors develop non-preemptive, packet scheduling algorithms that can deal with non-increasing TUFs. The GUS algorithm [19] allows more general model of almost arbitrary TUFs and resource dependencies.

Regardless of the advancement of UA scheduling algorithms and theories, there are few known analytical results. One of the best known results is developed by Baruah et. al. [6]. For step TUFs, the authors establish the $1\Big/\left(1+\sqrt{k}\right)^2$ upper bound on the competitive factor of any on-line scheduling algorithm, given the *importance ratio* of k [7]. Note that k is defined as the maximum task value density divided by the minimum task value density among the task set. This upper bound is achieved by the D^{over} algorithm presented in [18]. However, this upper bound is only valid by assuming task utilities are equal to computation times, which is different from out model.

The problem of probabilistic assurance for deadline-driven scheduling is mainly motivated by multimedia applications, where the execution times of periodic tasks, e.g., an MPEG decoder are modeled as random variables. Naturally, the objective is to schedule tasks so that the probability of meeting task deadlines is satisfactory.

In [5], Atlas and Bestavros propose the statistical RMA algorithm that can provide bound on percentages of deadline miss. However, the analysis is valid only for harmonic tasks and is restricted to RMS algorithm. The work done by Gardner [15] provides a more general framework of analyzing percentage of deadline miss for uni-processor systems with or without resource dependencies, and for real-time distributed systems as well.

Several similar techniques are also developed, such as [25]. However, to the best of the authors' knowledge, the problem of probabilistic performance assurance for UA scheduling has not been addressed before.

8 Conclusions and Future Work

Unlike other UA scheduling algorithms, S-UA is the first algorithm that seeks to maximize accrued utility and to provide task-level probabilistic performance assurance. This performance assurance is achieved

through critical time allocation and proper scheduling. Given a set of assurance probabilities $\{AP_i, \forall i\}$, the critical allocation algorithm determines a set of critical times $\{D_i, \forall i\}$ so that the critical times can be satisfied with at least the probabilities of AP_i. In the meanwhile, system-wide AUR can be maximized.

Several aspects of the S-UA algorithm are under investigation. For example, the condition presented in this paper is sufficient but not necessary, and it may be loose for some cases. To strengthen the performance assurance, it is important to provide tighter bound on task-level performance. Deriving sufficient and necessary conditions for some special distributions is another useful direction.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 4–13, December 1998.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of The IEEE Real-Time Systems Symposium*, pages 3–13, December 1998.

[3] L. Abeni and G. Buttazzo. Qos guarantee using probabilistic deadlines. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 242–249, June 1999.

[4] S. Aldarmi and A. Burns. Dynamic value-density for scheduling real-time systems. In *Proceedings of The 11th Euromicro Conference on Real-Time Systems*, pages 270–277, June 1999.

[5] A. Atlas and A. Bestavros. Statistical rate monotonic scheduling. In *Proceedings of The IEEE Real-Time Systems Symposium*, pages 123–132, December 1998.

[6] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, and L. Rosier. On the competitiveness of on-line task real-time task scheduling. In *Proceedings of The 12th IEEE Real-Time Systems Symposium*, pages 106–115, December 1991.

[7] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 101–110, October 1991.

[8] G. Buttazzo and J. Stankovic. Adding robustness in dynamic preemptive scheduling. In D. Fussel and M. Malek, editors, *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, pages 67–88. Kluwer Academic Publishers, October 1995.

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

[10] M. Caccamo, G. Buttazzo, and L. Sha. Handling execution overruns in hard real-time control systems. *IEEE Transactions on Computers*, 51(7):835–849, July 2002.

[11] K. Chen and P. Muhlethaler. A scheduling algorithm for tasks described by time value function. *Journal of Real-Time Systems*, 10(3):293–312, 1996.

[12] 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 The Seventh IEEE*

International Workshop on Parallel and Distributed Real-Time Systems, volume 1586 of *Lecture Notes in Computer Science*, pages 353–362. Springer-Verlag, April 1999.

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

[14] M. Gardner and J. Liu. Performance of algorithms for scheduling real-time systems with overrun and overload. In *Proceedings of Euromicro Conference on Real-Time Systems*, pages 287–296, June 1999.

[15] M. K. Gardner. *Probabilistic Analysis and Scheduling of Critical Soft Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[16] 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.

[17] S. Kirpatrick, C. Gelatt, Jr., and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[18] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for overloaded real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 290–299, December 1992.

[19] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms*. PhD thesis, Virginia Tech, 2004. `http://scholar.lib.vt.edu/theses/available/etd-08092004-230138/`.

[20] 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), September 2004.

[21] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

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

[23] D. P. Maynard, S. E. Shipman, et al. An example real-time command, control, and battle management application for alpha. Archons Project TR-88121, Carnegie Mellon University, December 1988.

[24] 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*, pages 278–286, June 1999.

[25] T. S. Tia, Z. Deng, et al. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of The IEEE Real-Time Technology and Applications Symposium*, pages 164–173, 1995.

[26] J. Wang and B. Ravindran. Time-utility function-driven switched ethernet: Packet scheduling algorithm, implementation, and feasibility analysis. *IEEE Transactions on Parallel and Distributed Systems*, 15(1):1–15, January 2003.