Utility Accrual Real-Time Scheduling for Multiprocessor Embedded Systems

Hyeonjoong Cho[a] Binoy Ravindran[b] E. Douglas Jensen[c]

[a]*Dept. of Computer and Information Science, Korea University, South Korea*

[b]*ECE Dept., Virginia Tech., Blacksburg, VA 24061, USA*

[c]*The MITRE Corporation Bedford, MA 01730, USA*

Abstract

We present the first *Utility Accrual* (or UA) real-time scheduling algorithm for multiprocessors, called *global Multiprocessor Utility Accrual scheduling algorithm* (or gMUA). The algorithm considers an application model where real-time activities are subject to time/utility function time constraints, variable execution time demands, and resource overloads where the total activity utilization demand exceeds the total capacity of all processors. We consider the scheduling objective of (1) probabilistically satisfying lower bounds on each activity's maximum utility, and (2) maximizing the system-wide, total accrued utility. We establish several properties of gMUA including optimal total utility (for a special case), conditions under which individual activity utility lower bounds are satisfied, a lower bound on system-wide total accrued utility, and bounded sensitivity for assurances to variations in execution time demand estimates. Finally, our simulation experiments validate our analytical results and confirm the algorithm's effectiveness.

Key words: Time utility function, utility accrual, multiprocessor systems, statistical assurance, real-time, scheduling

Email addresses: `raycho@korea.ac.kr` (Hyeonjoong Cho), `binoy@vt.edu` (Binoy Ravindran), `jensen@mitre.org` (E. Douglas Jensen).

1 Introduction

1.1 Utility Accrual Real-Time Scheduling

There are embedded real-time systems in many domains, such as robotic systems in the space domain (e.g., NASA/JPL's Mars Rover [1]) and control systems in the defense domain (e.g., airborne trackers [2]), which 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 arbitrary activity arrival patterns. Nevertheless, such systems desire the strongest possible assurances on activity timeliness behavior. Another important distinguishing feature of almost all of these systems is their relatively long task execution time magnitudes—e.g., in the order of milliseconds to minutes.

When resource overloads occur, meeting deadlines of all activities is impossible as the demand exceeds the supply. The urgency of an activity is typically orthogonal to its relative importance—e.g., the most urgent activity can be the least important, and vice versa; the most urgent can 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 under-loads, such a distinction need not be made, because deadline-based scheduling algorithms such as EDF are optimal (on one processor) [3].

Deadlines by themselves cannot express both urgency and importance. Thus, we consider the abstraction of time/utility functions (or TUFs) [4] that express the utility of completing an application 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 as a deadline on the X-axis, and importance is denoted by utility on the Y-axis.

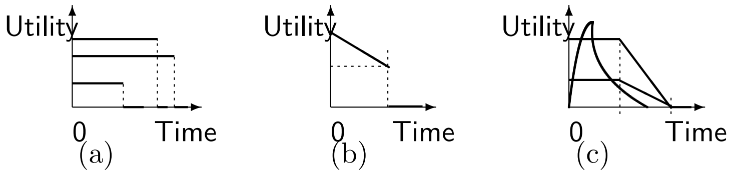


Fig. 1. Example TUF Time Constraints: (a) Step TUFs; (b) AWACS TUF [2]; and (c) Coastal Air defense TUFs [5]

Many embedded real-time systems also have activities that are subject to *non-deadline* time constraints, such as those where the utility attained for activity completion *varies* (e.g., decreases, increases) with completion time. This is in contrast to deadlines, where a positive utility is attained for completing the activity anytime before the deadline, after which zero, or infinitely negative utility is attained. Figures 1(a)-1(c) show examples of such time constraints from two real applications (see [2] and references therein for application details). For example, in [2], Clark *at. al.* discuss an AWACS tracker application which collects radar sensor reports, identifies airborne objects (or "track objects") in them, and associates those objects to track states that are maintained in a track database. Here, each job of a track association task has a TUF time constraint, and all jobs of the same task have the same time constraint. The tracker is routinely overloaded, so the collective timeliness optimality criterion is to meet as many job deadlines (or termination times) as possible, and to maximize the total utility obtained from the completion of as many jobs as possible. Another example timeliness requirement is found in a NASA/JPL Mars Science Lab Rover application—scheduling of processor cycles for the Mission Data System (MDS) [1]. Additional key features of this application include transient and permanent processor cycle overloads, and activity time scales (e.g., frequency of constructing MDS schedules) that are of the order of minutes.

When activity time constraints are specified using TUFs, which subsume deadlines, the scheduling criteria are based on accrued utility, such as maximizing sum of the activities' attained utilities. We call such criteria, *utility accrual* (or UA) criteria, and scheduling algorithms that optimize them UA scheduling algorithms.

On single processors, UA algorithms that maximize accrued utility for downward step TUFs (see algorithms in [6]) default to EDF during under-loads, since EDF satisfies all deadlines during under-loads. Consequently, they obtain the maximum possible accrued utility during under-loads. During overloads, they favor more important activities (since more utility can be attained from them), irrespective of urgency. Thus, deadline scheduling's optimal timeliness behavior is a special case of UA scheduling.

1.2 Real-time Scheduling on Multiprocessors

Multiprocessor architectures—e.g., Symmetric Multi-Processors (SMPs), Single Chip Heterogeneous Multiprocessors (SCHMs)—are recently becoming more attractive for embedded systems because they are significantly decreasing in price. This makes them very desirable for embedded system applications with high computational workloads, where additional, cost-effective processing capacity is often needed. Responding to this trend, (real-time operating system) RTOS vendors are increasingly providing multiprocessor platform support—e.g., QNX Neutrino is now available for a variety of SMP chips [7]. But this exposes the critical need for real-time scheduling for multiprocessors—

a comparatively undeveloped area of real-time scheduling which has recently received significant research attention, but is not yet well supported by the RTOS products. Consequently, the impact of cost-effective multiprocessor platforms for embedded systems remains nascent.

One highly developed class of multiprocessor scheduling algorithms is static scheduling of a program represented by a directed task graph on a multiprocessor system to minimize the program completion time [8]. In contrast to that, the class of multiprocessor scheduling algorithms our work is in seeks to satisfy tasks' completion time constraints (such as deadlines) by performing dynamic (i.e., run-time) task assignment to processors and dynamic scheduling of the tasks on each processor.

One unique aspect of multiprocessor scheduling is the degree of run-time migration that is allowed for job instances of a task across processors (at scheduling events). Example migration models include: (1) *full migration*, where jobs are allowed to arbitrarily migrate across processors during their execution. This usually implies a global scheduling strategy, where a single shared scheduling queue is maintained for all processors and a system-wide scheduling decision is made by a single (global) scheduling algorithm; (2) *no migration*, where tasks are statically (off-line) partitioned and allocated to processors. At run-time, job instances of tasks are scheduled on their respective processors by processors' local scheduling algorithm, like single processor scheduling; and (3) *restricted migration*, where some form of migration is allowed—e.g., at job boundaries.

Carpentar *et al.* [9] have catalogued multiprocessor real-time scheduling algorithms considering the degree of job migration. The Pfair class of algorithms [10] that allow full migration have been shown to achieve a schedulable utilization bound (below which all tasks meet their deadlines) that equals the total capacity of all processors—thus, they are theoretically optimal. However, Pfair algorithms incur significant overhead due to their quantum-based scheduling approach [11]. Thus, scheduling algorithms other than Pfair (e.g., global EDF) have also been studied though their schedulable utilization bounds are lower.

Global EDF scheduling on multiprocessors is subject to the "Dhall effect" [12], where a task set with total utilization demand arbitrarily close to one cannot be scheduled so as to satisfy all deadlines. To overcome this, researchers have studied global EDF's behavior under restricted individual task utilizations. For example, on M processors, Srinivasan and Baruah show that when the maximum individual task utilization, u_{max}, is bounded by $M/\left(2M-1\right)$, EDF's utilization bound is $M^2/\left(2M-1\right)$ [13]. In [14], Goossens *et. al* show that EDF's utilization bound is $M-\left(M-1\right)u_{max}$. This work was later extended by Baker for the more general case of deadlines less than or equal to periods in [15]. In [16], Bertogna *et al.* show that Baker's utilization bound does not dominate the bound of Goossens *et. al*, and vice versa.

While most of these past works focus on the hard real-time objective of always meeting all deadlines, recently there have been efforts that consider the soft real-time objective of bounding the tardiness of tasks. In [17], Srinivasan and Anderson derive a tardiness bound for a suboptimal Pfair scheduling algorithm. In [25], for a restricted migration model where migration is allowed only at job boundaries, Andersen *et. al* present an EDF-based partitioning scheme and scheduling algorithm that ensures bounded tardiness. In [11], Devi and Anderson derive the tardiness bounds for global EDF when the total utilization demand of tasks may equal the number of available processors.

1.3 Contributions

In this paper, we consider the problem of global UA scheduling on an SMP system with M number of identical processors in environments with *dynamically uncertain* properties. By dynamic uncertainty, we mean operating in environments where arrival and execution behaviors of tasks are subject to run-time uncertainties, causing resource overloads. Nonetheless, such systems desire the strongest possible assurances on task timing behaviors—both that of individual activities behavior and that of collective, system-wide behavior. Statistical assurances are appropriate for these systems.

Multiprocessor scheduling should determine which processor each task should be assigned to for execution (the allocation problem) and in which order tasks should start execution at each processor (the scheduling problem). Our scheduling algorithm does both.

Real-time scheduling for multiprocessors is categorized into: global scheduling, where all jobs are scheduled together based on a single queue for all processors; partitioned scheduling, where tasks are assigned to processors, and each processor is scheduled separately. We consider global multiprocessor scheduling (that allows full migration as opposed to partitioned scheduling) because of its improved schedulability and flexibility [18]. Further, in many embedded architectures (e.g., those with no cache), its migration overhead has a lower impact on performance [16]. Moreover, applications of interest to us [1,2] are often subject to resource overloads, during when the total application utilization demand exceeds the total processing capacity of all processors. When that happens, we hypothesize that global scheduling has greater scheduling flexibility, resulting in greater accrued activity utility, than does partitioned scheduling.

We consider repeatedly occurring application activities (e.g., tasks) that are subject to TUF time constraints, variable execution times, and overloads. To account for uncertainties in activity execution behaviors, we employ a stochastic model for activity demand and execution. Activities repeatedly arrive with a known minimum inter-arrival time. For such a model, our objective is to: (1) provide statistical assurances on individual activity timeliness behavior including probabilistically-satisfied lower bounds on each activity's maximum utility; (2) provide assurances on system-level timeliness behavior including an assured lower bound on the sum of the activities' attained utilities; and (3) maximize the sum of activities' attained utilities.

This problem has not been studied in the past and is $\mathcal{NP}$-hard. We present a polynomial-time, heuristic algorithm for the problem called the *global Multiprocessor Utility Accrual (gMUA) scheduling algorithm*. We establish several properties of gMUA including optimal total utility for the special case of downward step TUFs and application utilization demand not exceeding global EDF's utilization bound—conditions under which individual activity utility lower bounds are satisfied, and a lower bound is established on system-wide total accrued utility. We also show that the algorithm's assurances have bounded sensitivity to variations in execution time demand estimates, in the sense that the assurances hold as long as the variations satisfy a sufficient condition that we present. Further, we show that the algorithm is robust against a variant of the Dhall effect.

Thus, the contribution of this paper is the gMUA algorithm. We are not aware of any other efforts that solve the problem solved by gMUA.

The rest of the paper is organized as follows: Section 2 describes our models and scheduling objective. In Section 3, we discuss the rationale behind gMUA and present the algorithm. We describe the algorithm's properties in Section 4. We report our simulation-based experimental studies in Section 5. The paper concludes in Section 6.

2 Models and Objective

2.1 Activity Model

We consider the application to consist of a set of tasks, denoted $\mathbf{T}$=$\{T_1, T_2, ..., T_n\}$. Each task T_i has a number of instances, called jobs, and these jobs may be released either periodically or sporadically with a known minimal inter-arrival time. The j^{th} job of task T_i is denoted as $J_{i,j}$. The period or minimal inter-arrival time of a task T_i is denoted as p_i.

We initially assume that all tasks are independent—i.e., they do not share any resource or have any precedences.

The basic scheduling entity that we consider is the job abstraction. Thus, we use J to denote a job without being task specific, as seen by the scheduler at any scheduling event.

A job's time constraint is specified using a TUF. Jobs of the same task have the same TUF. We use $U_i()$ to denote the TUF of task T_i. Thus, completion of the job $J_{i,j}$ at time t will yield an utility of $U_i(t)$.

TUFs can be classified into unimodal and multimodal functions. Unimodal TUFs are those for which any decrease in utility cannot be followed by an increase. Figure 1 shows examples. TUFs which are not umimodal are multimodal. In this paper, we focus on *non-increasing* unimodal TUFs, as they encompass majority of the time constraints in our motivating applications.

Each TUF U_i of $J_{i,j}$ has an initial time $t^I_{i,j}$ and a termination time $t^X_{i,j}$. Initial and termination times are the earliest and the latest times for which the TUF is defined, respectively. We assume that $t^I_{i,j}$ is the arrival time of job $J_{i,j}$, and $t^X_{i,j} - t^I_{i,j}$ is the period or minimal inter-arrival time p_i of the task T_i. If $J_{i,j}$'s $t^X_{i,j}$ is reached and execution of the corresponding job has not been completed, an exception is raised, and the job is aborted.

2.2 Job Execution Time Demands

We estimate the statistical properties, e.g., distribution, mean, variance, of job execution time demand rather than the worst-case demand because: (1) applications of interest to us [1,2] exhibit a large variation in their *actual workload.* Thus, the statistical estimation of the demand is much more stable and hence more predictable than the actual workload; (2) worst-case workload is usually a very conservative prediction of the actual workload [19], resulting in resource over-supply; and(3) allocating execution times based on the statistical estimation of tasks' demands can provide statistical performance assurances, which is sufficient for our motivating applications.

Let Y_i be the random variable of a task T_i's execution time demand. Estimating the execution time demand distribution of the task involves two steps: (1) profiling its execution time usage, and (2) deriving the probability distribution of that usage. A number of measurement-based, off-line and online profiling mechanisms exist (e.g., [20]). We assume that the mean and variance of Y_i are finite and determined through either online or off-line profiling.

We denote the *expected* execution time demand of a task T_i as $E(Y_i)$, and the variance on the demand as $Var(Y_i)$.

2.3 Statistical Timeliness Requirement

We consider a task-level statistical timeliness requirement—each task must attain some percentage of its maximum possible utility with a certain probability. For a task T_i, this requirement is specified as $\{\nu_i, \rho_i\}$, which implies that T_i must attain at least ν_i percentage of its maximum possible utility with the probability ρ_i. This is also the requirement for each job of T_i. Thus, for example, if $\{\nu_i, \rho_i\} = \{0.7, 0.93\}$, then T_i must attain at least 70% of its maximum possible utility with a probability no less than 93%. For step TUFs, ν can only take the value 0 or 1. Note that the objective of always satisfying all task deadlines is the special case of $\{\nu_i, \rho_i\} = \{1.0, 1.0\}$.

This statistical timeliness requirement on the utility of a task implies a corresponding requirement on the range of its job *sojourn times.* [1] Since we focus on non-increasing unimodal TUFs, upper-bounding job sojourn times will lower-bound job and task utilities.

2.4 Scheduling Objective

We consider a two-fold scheduling criterion: (1) assure that each task T_i attains the specified percentage ν_i of its maximum possible utility with at least the specified probability ρ_i; and (2) maximize the system-level total accrued utility. We also desire to obtain a lower bound on the system-level total accrued utility. When it is not possible to satisfy ρ_i for each task (e.g., due to overloads), our objective is to maximize the system-level total accrued utility.

This problem is $\mathcal{NP}$-hard because it subsumes the $\mathcal{NP}$-hard problem of scheduling dependent tasks with step TUFs on one processor [21].

[1] A job's sojourn time is defined as the time interval from the job's release to its completion.

3 The gMUA Algorithm

3.1 Bounding Accrued Utility

Let $s_{i,j}$ be the sojourn time of the j^{th} job of task T_i. Task T_i's statistical timeliness requirement can be represented as $Pr(U_i(s_{i,j}) \geq \nu_i \times U_i^{max}) \geq \rho_i$, where U_i^{max} is the maximum value of $U_i()$. Since TUFs are assumed to be non-increasing, it is sufficient to have $Pr(s_{i,j} \leq D_i) \geq \rho_i$, where D_i is the upper bound on the sojourn time of task T_i. We call D_i the "critical time" hereafter, and it is calculated as $D_i = U_i^{-1}(\nu_i \times U_i^{max})$, where $U_i^{-1}(x)$ denotes the inverse function of TUF $U_i()$. Thus, T_i is (probabilistically) assured to attain at least the utility percentage $\nu_i = U_i(D_i)/U_i^{max}$, with the probability ρ_i.

3.2 Bounding Utility Accrual Probability

Since task execution time demands are stochastically specified (through means and variances), we need to determine the actual execution time that must be allocated to each task, such that the desired utility attained probability ρ_i is satisfied. Further, this execution time allocation must account for the uncertainty in the execution time demand specification (i.e., the variance factor).

Given the mean and the variance of a task T_i's execution time demand Y_i, by a one-tailed version of the Chebyshev's inequality, when $y \geq E(Y_i)$, we have:

$$Pr[Y_i < y] \geq \frac{(y - E(Y_i))^2}{Var(Y_i) + (y - E(Y_i))^2} \tag{1}$$

From a probabilistic point of view, Equation 1 is the direct result of the cumulative distribution function of task T_i's execution time demands—i.e., $F_i(y) = Pr[Y_i \leq y]$. Recall that each job of task T_i must attain ν_i percentage of its maximum possible utility with a probability ρ_i. To satisfy this requirement, we let $\rho_i' = Pr[Y_i < C_i] = \frac{(C_i - E(Y_i))^2}{Var(Y_i) + (C_i - E(Y_i))^2} \geq \rho_i$ and obtain the minimum required execution time $C_i = E(Y_i) + \sqrt{\frac{\rho_i' \times Var(Y_i)}{1 - \rho_i'}}$.

Thus, the gMUA algorithm allocates C_i execution time units to each job $J_{i,j}$ of task T_i, so that the probability that job $J_{i,j}$ requires no more than the allocated C_i execution time units is at least ρ_i—i.e., $Pr[Y_i < C_i] \geq \rho_i' \geq \rho_i$. We set $\rho_i' = (\max\{\rho_i\})^{\frac{1}{n}}, \forall i$ to satisfy requirements given by ρ_i. Supposing that each task is allocated C_i time within its p_i, the actual demand of each task often varies. Some jobs of the task may complete their execution before using up their allocated time, and the others may not. gMUA probabilistically schedules the jobs of a task T_i to provide assurance probability ρ'_i ($\geq \rho_i$) as long as they are satisfying a certain feasibility test.

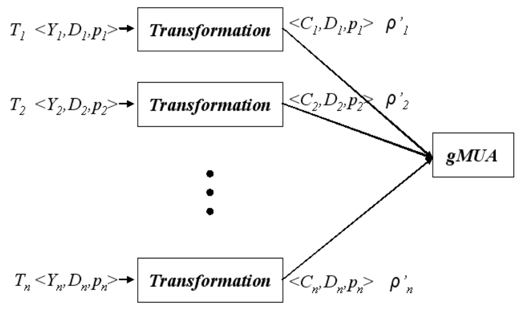


Fig. 2. Transformation Array and gMUA

Figure 2 shows our method of transforming the stochastic execution time demand ($E(Y_i)$ and $Var(Y_i)$) into execution time allocation C_i. The values after the transformation are utilized as reference parameters by gMUA, the following scheduling algorithm. Note that this transformation is independent of our proposed scheduling algorithm.

3.3 Algorithm Description

gMUA's scheduling events include job arrival and job completion. To describe gMUA, we define the following variables and auxiliary functions:

- ζ_r: current job set in the system, including running jobs and unscheduled jobs.
- σ_{tmp}, σ_a: a temporary schedule; σ_m: schedule for processor m, where $m \leq M$.
- $J_k.C(t)$: J_k's remaining allocated execution time.
- `offlineComputing()` is computed at time $t = 0$ once. For a task T_i, it computes C_i as $C_i = E(Y_i) + \sqrt{\frac{\rho_i \times Var(Y_i)}{1-\rho_i}}$.
- `UpdateRAET(`ζ_r`)` updates the remaining allocated execution time of all jobs in the set ζ_r.
- `feasible(`σ`)` returns a boolean value denoting schedule σ's feasibility; `feasible(`J_k`)` denotes job J_k's feasibility. For σ (or J_k) to be feasible, the predicted completion time of each job in σ (or J_k), must not exceed its critical time.
- `sortByECF(`σ`)` sorts jobs of σ in the order of earliest critical time first.
- `findProcessor()` returns the ID of the processor on which the currently assigned tasks have the shortest sum of allocated execution times.
- `append(`J_k`,`σ`)` appends job J_k at the end of schedule σ.
- `remove(`J_k`,`σ`)` removes job J_k from schedule σ.
- `removeLeastPUDJob(`σ`)` removes job with the least *potential utility density* (PUD) from schedule σ. PUD is the ratio of the expected job utility (ob-

tained when job is immediately executed to completion) to the remaining job allocated execution time—i.e., the PUD of a job J_k is $\frac{U_k(t+J_k.C(t))}{J_k.C(t)}$. Thus, PUD measures the job's "return on investment." This function returns the removed job.

- `headOf(`σ_m`)` returns the set of jobs that are at the head of schedule σ_m, $1 \leq m \leq M$.

Algorithm 1: gMUA

```
1  Input  : $\mathbf{T}$={$T_1$,...,$T_N$}, $\zeta_r$={$J_1$,...,$J_n$}, M:# of processors
2  Output : array of dispatched jobs to processor $p$, $Job_p$
3  Data: {$\sigma_1$, ..., $\sigma_M$}, $\sigma_{tmp}$, $\sigma_a$
4  offlineComputing($\mathbf{T}$);
5  Initialization: {$\sigma_1$, ..., $\sigma_M$} = {0, ..., 0};
6  UpdateRAET($\zeta_r$);
7  for $\forall J_k \in \zeta_r$ do
8      $J_k.PUD = \frac{U_k(t+J_k.C(t))}{J_k.C(t)}$;
9  $\sigma_{tmp}$ = sortByECF( $\zeta_r$ );
10 for $\forall J_k \in \sigma_{tmp}$ from head to tail do
11     if $J_k.PUD > 0$ then
12         $m$ = findProcessor();
13         append($J_k$, $\sigma_m$);
14 for $m = 1$ to $M$ do
15     $\sigma_a$ = null;
16     while !feasible( $\sigma_m$) and !IsEmpty( $\sigma_m$ ) do
17         t = removeleastPUD( $\sigma_m$ );
18         append( $t$, $\sigma_a$ );
19     sortByECF( $\sigma_a$ );
20     $\sigma_m$ += $\sigma_a$;
21 {$Job_1$, ..., $Job_M$} = headOf( {$\sigma_1$, ..., $\sigma_M$} );
22 return {$Job_1$, ..., $Job_M$};
```

A description of gMUA at a high level of abstraction is shown in Algorithm 1. The procedure `offlineComputing()` is included in line 4, although it is executed only once at $t = 0$. When gMUA is invoked, it updates the remaining allocated execution time of each job. The remaining allocated execution times of running jobs are decreasing, while those of unscheduled jobs remain constant. The algorithm then computes the PUDs of all jobs.

The jobs are then sorted in the order of earliest critical time first (or ECF), in line 9. In each step of the `for` loop from line 10 to line 13, the job with the earliest critical time is selected to be assigned to a processor. The processor that yields the shortest sum of allocated execution times of all jobs in its local schedule is selected for assignment (procedure `findProcessor()`). The rationale for this choice is that the shortest summed execution time processor results in the nearest scheduling event after assigning each job, and therefore, it establishes the same schedule as global EDF does. Then, the job J_k with the earliest critical time is inserted into the local schedule σ_m of the selected processor m.

In the `for`-loop from line 14 to line 20, gMUA attempts to make each local schedule feasible by removing the lowest PUD job. In line 16, if σ_m is not feasible, then gMUA removes the job with the least PUD from σ_m until σ_m becomes feasible. All removed jobs are temporarily stored in a schedule σ_a and then appended to each σ_m in ECF order. Note that simply aborting the removed jobs may result in decreased accrued utility. This is because the algorithm may decide to remove a job which is estimated to have a longer allocated execution time than its actual one, even though it may be able to attain utility. For this case, gMUA gives the job another chance to be scheduled instead of aborting it, which eventually makes the algorithm more robust. Finally, each job at the head of $\sigma_m, 1 \leq m \leq M$ is selected for execution on the respective processor.

gMUA's time cost depends upon that of procedures `sortByECF()`, `findprocessor()`, `append()`, `feasible()`, and `removeLeastPUDJob()`. With n tasks, `sortByECF()` costs $O(nlogn)$. For SMPs with a restricted number of processors, `findprocessor()`'s costs $O(M)$. While `append()` costs $O(1)$ time, both `feasible()` and `removeLeastPUDJob()` cost $O(n)$. The `while`-loop in line 16 iterates at most n times, for an entire loop cost of $O(n^2)$. Thus, the algorithm costs $O(Mn^2)$. However, M of contemporary SMPs is usually small (e.g., 16) and bounded with respect to the problem size of number of tasks. Thus, gMUA costs $O(n^2)$.

gMUA's $O(n^2)$ cost is similar to that of many past UA algorithms [6]. Our prior implementation experience with UA scheduling at the middleware level has shown that the overhead is in the magnitude of sub-milliseconds [22] (sub-microsecond overheads may be possible at the kernel level). We anticipate a similar overhead magnitude for gMUA. Though this cost is higher than that of many traditional algorithms, the cost is justified for applications with longer execution time magnitudes such as those that we focus on here.

In traditional small scale static hard real-time subsystems, the task time scales are usually microseconds to milliseconds to even a few seconds. For those time scales, the time overhead of a scheduling algorithm must be proportionately low. The time overhead of our many TUF/UA algorithms precludes software implementations of them from being used in systems with mS-S time scales—although hardware (e.g., gate array) implementations can be used even down there.

But our TUF/UA scheduling algorithms are intended for the many important time-critical applications in the gap between: classical real-time's mS-S time scales; and the classical logistics, job shop, etc. systems' many minutes to many hours time scales where proportionately higher cost general scheduling theory, evolutionary algorithms, and linear programming are used.

The systems in that gap have time scales in the 1 S to few minutes range but their timeliness requirements are no less–and often more (cf. military combat systems)—challenging and mission-critical than those of classical real-time subsystems. These systems have adequate time for TUF/UA scheduling, and those of interest to us need the benefits of it. [2]

4 Algorithm Properties

4.1 Timeliness Assurances

We establish gMUA's timeliness assurances under the conditions of (1) independent tasks that arrive periodically, and (2) task utilization demand satisfies any of the feasible utilization bounds for global EDF (GFB, BAK, BCL) in [16].

Theorem 1 (Optimal Performance with Downward Step Shaped TUFs) *Suppose that only downward step shaped TUFs are allowed under conditions (1) and (2). Then, a schedule produced by global EDF is also produced by gMUA, yielding equal accrued utilities. This is a critical time-ordered schedule.*

PROOF. We prove this by examining Algorithm 1. In line 9, the queue σ_{tmp} is sorted in a non-decreasing critical time order. In line 12, the function `findProcessor()` returns the index of the processor on which the summed execution time of assigned tasks is the shortest among all processors. Assume that there are n tasks in the current ready queue. We consider two cases: (1) $n \leq M$ and (2) $n > M$. When $n \leq M$, the result is trivial—gMUA's schedule of tasks on each processor is identical to that produced by EDF (every processor has a single task or none assigned). When $n > M$, task T_i $(M < i \leq n)$ will be assigned to the processor whose tasks have the shortest summed execution time. This implies that this processor will have the earliest completion for all assigned tasks up to T_{i-1}, so that the event that will assign T_i will occur by

[2] When UA scheduling is desired with lower overhead, solutions and tradeoffs exist. Examples include linear-time stochastic UA scheduling [26], and using special-purpose hardware accelerators for UA scheduling (analogous to floating-point co-processors) [23].

this completion. Note that tasks in σ_{tmp} are selected to be assigned to processors according to ECF. This is precisely the global EDF schedule, as we consider a TUFs critical time in UA scheduling to be the same as a deadline in traditional hard real-time scheduling. Under conditions (1) and (2), EDF meets all deadlines. Thus, each processor always has a feasible schedule, and the `while`-block from line 16 to line 18 will never be executed. Thus, gMUA produces the same schedule as global EDF.

Some important corollaries about gMUA's timeliness behavior can be deduced from EDF's behavior under conditions (1) and (2).

Corollary 2 *Under conditions (1) and (2), gMUA always completes the allocated execution time of all tasks before their critical times.*

Theorem 3 (Statistical Task-Level Assurance) *Under conditions (1) and (2), gMUA meets the statistical timeliness requirement $\{\nu_i, \rho_i\}$ for each task T_i.*

PROOF. From Corollary 2, all allocated execution times of tasks can be completed before their critical times. Further, based on the results of Equation 1, among the actual processor time of task T_i's jobs, at least ρ_i' $(\geq \rho_i)$ of them have lesser actual execution time than the allocated execution time. Thus, gMUA can satisfy at least ρ_i critical times—i.e., the algorithm attains ν_i utility with a probability of at least ρ_i.

Theorem 4 (System-Level Utility Assurance) *Under conditions (1) and (2), if a task T_i's TUF has the highest height U_i^{max}, then the system-level utility ratio, defined as the utility accrued by gMUA with respect to the system's maximum possible utility, is at least* $\frac{\rho_1\nu_1U_1^{max}/P_1+\ldots+\rho_n\nu_nU_n^{max}/P_n}{U_1^{max}/P_1+\ldots+U_n^{max}/P_n}$.

PROOF. We denote the number of jobs released by task T_i as m_i. Each m_i is computed as $\frac{\Delta t}{p_i}$, where Δt is a time interval. Task T_i can attain at least ν_i percentage of its maximum possible utility with the probability ρ_i. Thus, the ratio of the system-level accrued utility to the system's maximum possible utility is $\frac{\rho_1\nu_1U_1^{max}m_1+\ldots+\rho_n\nu_nU_n^{max}m_n}{U_1^{max}m_1+\ldots+U_n^{max}m_n}$. Thus, the formula comes to $\frac{\rho_1\nu_1U_1^{max}/P_1+\ldots+\rho_n\nu_nU_n^{max}/P_n}{U_1^{max}/P_1+\ldots+U_n^{max}/P_n}$.

4.2 Dhall Effect

The *Dhall effect* [12] shows that there exists a task set that requires nearly one total utilization demand, but cannot be scheduled to meet all deadlines using global EDF even with infinite number of processors. Prior research has revealed that this is caused by the poor performance of global EDF when the task set contains both high utilization tasks and low utilization tasks together. This phenomenon, in general, can also affect UA scheduling algorithms' performance, and impede such algorithms' ability to maximize the total accrued utility. We discuss this with an example inspired from [24]. We consider the case when the execution time demands of all tasks are constant with no variance, and gMUA estimates them accurately.

Example A. Consider $M + 1$ periodic tasks that are scheduled on M processors using global EDF. Let task T_i, where $1 \leq i \leq M$, have $p_i = D_i = 1, C_i = 2\epsilon$, and task T_{M+1} have $P_{M+1} = D_{M+1} = 1 + \epsilon, C_{M+1} = 1$. We assume that each task T_i has a downward step shaped TUF with height h_i and task T_{M+1} has a downward step shaped TUF with height H_{M+1}. When all tasks arrive at the same time, tasks T_i will execute immediately and complete their execution 2ϵ time units later. Task T_{M+1} then executes from time 2ϵ to time $1 + 2\epsilon$. Since task T_{M+1}'s critical time—we assume here it is the same as its period—is $1 + \epsilon$, it begins to miss its critical time. By letting $M \to \infty$, $\epsilon \to 0$, $h_i \to 0$ and $H_{M+1} \to \infty$, we have a task set whose total utilization demand is near one and the maximum possible accrued utility is infinite, but which finally accrues zero utility even with infinite number of processors.

We call this phenomenon the *UA Dhall effect*. Conclusively, one of the reasons why global EDF is inappropriate as a UA scheduling algorithm is that it suffers this effect. However, gMUA overcomes this phenomenon.

Example B. Consider the same scenario as in Example A, but now, let the task set be scheduled by gMUA. In Algorithm 1, gMUA first tries to schedule tasks like global EDF, but it will fail to do so as we saw in Example A. When gMUA finds that T_{M+1} will miss its critical time on processor m (where $1 \leq m \leq M$), the algorithm will select a task with lower PUD on processor m for removal. On processor m, there should be two tasks, T_m and T_{M+1}. T_m is one of T_i where $1 \leq i \leq M$. When letting $h_i \to 0$ and $H_{M+1} \to \infty$, the PUD of task T_m is almost zero and that of task T_{M+1} is infinite. Therefore, gMUA removes T_m and eventually accrues infinite utility as expected.

In the case when the *Dhall effect* occurs, we can establish the *UA Dhall effect* by assigning extremely high utility to the task that will be selected and miss its deadline using global EDF. This implies that the scheduling algorithm suffering from the *Dhall effect* will likely suffer from *UA Dhall effect*, when it schedules the tasks that are subject to TUF time constraints.

The fact that gMUA is more robust against the *UA Dhall effect* than is global EDF can be observed in our simulation experiments (see Section 5).

4.3 Sensitivity of Assurances

gMUA is designed under the assumption that tasks' expected execution time demands and the variances of the demands—i.e., the algorithm inputs $E(Y_i)$ and $Var(Y_i)$—are correct. However, it is possible that these inputs may have been miscalculated (e.g., due to errors in application profiling) or that the input values may change over time (e.g., due to changes in the application's execution context). To understand gMUA's behavior when this happens, we assume that the expected execution time demands, $E(Y_i)$'s, and their variances, $Var(Y_i)$'s, are erroneous, and develop the sufficient condition under which the algorithm is still able to meet $\{\nu_i, \rho_i\}$ for all tasks T_i.

Let a task T_i's correct expected execution time demand be $E(Y_i)$ and its correct variance be $Var(Y_i)$, and let an erroneous expected demand $E'(Y_i)$ and an erroneous variance $Var'(Y_i)$ be specified as the input to gMUA. Let the task's statistical timeliness requirement be $\{\nu_i, \rho_i\}$. We show that if gMUA can satisfy $\{\nu_i, \rho_i\}$ with the correct expectation $E(Y_i)$ and the correct variance $Var(Y_i)$, then there exists a sufficient condition under which the algorithm can still satisfy $\{\nu_i, \rho_i\}$ even with the incorrect expectation $E'(Y_i)$ and incorrect variance $Var'(Y_i)$.

Theorem 5 () *Assume that gMUA satisfies $\{\nu_i, \rho_i\}, \forall i$, under correct, expected execution time demand estimates, $E(Y_i)$'s, and their correct variances, $Var(Y_i)$'s. When incorrect expected values, $E'(Y_i)$'s, and variances, $Var'(Y_i)$'s, are given as inputs instead of $E(Y_i)$'s and $Var(Y_i)$'s, gMUA satisfies $\{\nu_i, \rho_i\}, \forall i$, if $E'(Y_i) + (C_i - E(Y_i))\sqrt{\frac{Var'(Y_i)}{Var(Y_i)}} \geq C_i, \forall i$, and the task execution time allocations, computed using $E'(Y_i)$'s and $Var'(Y_i)$, satisfy any of the feasible utilization bounds for global EDF.*

PROOF. We assume that if gMUA has correct $E(Y_i)$'s and $Var(Y_i)$'s as inputs, then it satisfies $\{\nu_i, \rho_i\}, \forall i$. This implies that the C_i's determined by Equation 1 are feasibly scheduled by gMUA satisfying all task critical times:

$$\rho_i = \frac{(C_i - E(Y_i))^2}{Var(Y_i) + (C_i - E(Y_i))^2}. \tag{2}$$

However, gMUA has incorrect inputs, $E'(Y_i)$'s and $Var'(Y_i)$, and based on those, it determines C_i's by Equation 1 to obtain the probability $\rho_i, \forall i$:

$$\rho_i = \frac{(C_i' - E'(Y_i))^2}{Var'(Y_i) + (C_i' - E'(Y_i))^2}. \tag{3}$$

Unfortunately, C_i' that is calculated from the erroneous $E'(Y_i)$ and $Var'(Y_i)$ leads gMUA to another probability ρ_i' by Equation 1. Thus, although we expect assurance with the probability ρ_i, we can only obtain assurance with the probability ρ_i' because of the error. ρ' is given by:

$$\rho_i' = \frac{(C_i' - E(Y_i))^2}{Var(Y_i) + (C_i' - E(Y_i))^2}. \tag{4}$$

Note that we also assume that tasks with C_i' satisfy the global EDF's utilization bound; otherwise gMUA cannot provide the assurances. To satisfy $\{\nu_i, \rho_i\}, \forall i$, the actual probability ρ_i' must be greater than the desired probability ρ_i. Since $\rho_i' \geq \rho_i$,

$$\frac{(C_i' - E(Y_i))^2}{Var(Y_i) + (C_i' - E(Y_i))^2} \geq \frac{(C_i - E(Y_i))^2}{Var(Y_i) + (C_i - E(Y_i))^2}.$$

Hence, $C' \geq C_i$. From Equations 2 and 3,

$$C_i' = E'(Y_i) + (C_i - E(Y_i))\sqrt{\frac{Var'(Y_i)}{Var(Y_I)}} \geq C_i. \tag{5}$$

Corollary 6 () *Assume that gMUA satisfies $\{\nu_i, \rho_i\}, \forall i$, under correct, expected execution time demand estimates, $E(Y_i)$'s, and their correct variances, $Var(Y_i)$'s. When incorrect expected values, $E'(Y_i)$'s, are given as inputs instead of $E(Y_i)$'s but with correct variances $Var(Y_i)$, gMUA satisfies $\{\nu_i, \rho_i\}, \forall i$, if $E'(Y_i) \geq E(Y_i), \forall i$, and the task execution time allocations, computed using $E'(Y_i)$'s, satisfy the feasible utilization bound for global EDF.*

PROOF. This can be proved by replacing $Var'(Y_i)$ with $Var(Y_i)$ in Equation 5.

Corollary 6, a special case of Theorem 5, is intuitively straightforward. It essentially states that if overestimated demands are feasible, then gMUA can still satisfy $\{\nu_i, \rho_i\}, \forall i$. Thus, it is desirable to specify larger $E'(Y_i)$s as input to the algorithm when there is the possibility of errors in determining the expected demands, or when the expected demands may vary with time.

Corollary 7 () *Assume that gMUA satisfies $\{\nu_i, \rho_i\}, \forall i$, under correct, expected execution time demand estimates, $E(Y_i)$'s, and their correct variances, $Var(Y_i)$'s. When incorrect variances, $Var'(Y_i)$'s, are given as inputs instead of correct $Var(Y_i)$'s but with correct expectations $E(Y_i)$'s, gMUA satisfies $\{\nu_i, \rho_i\}, \forall i$, if $Var'(Y_i) \geq Var(Y_i), \forall i$, and the task execution time allocations, computed using $E'(Y_i)$'s, satisfy the feasible utilization bound for global EDF.*

PROOF. This can be proved by replacing $E'(Y_i)$ with $E(Y_i)$ in Equation 5.

5 Experimental Evaluation

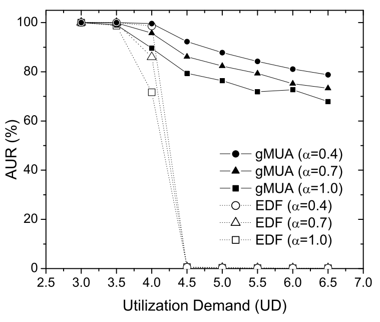
We conducted simulation-based experimental studies to validate our analytical results, and to compare gMUA's performance with global EDF. We take the global EDF algorithm as our counterpart since it is one of the global scheduling algorithms, like gMUA, that allows full migration and does not depend on time quanta. Note that gMUA uses deadline time constraints as opposed to gMUA which uses TUF time constraints. We consider two cases: (1) the demand of all tasks is constant (i.e., zero variance) and gMUA exactly estimates their execution time allocation, and (2) the demand of all tasks statistically varies and gMUA probabilistically estimates the execution time allocation for each task. The former experiment is conducted to evaluate gMUA's generic performance as opposed to EDF, while the latter is conducted to validate the algorithm's assurances. The experiments focus on the no dependency case (i.e., each task is assumed to be independent of others).

5.1 Performance with Constant Demand

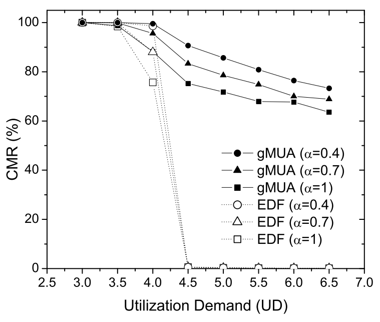
We consider an SMP machine with four processors. A task T_i's period $p_i (= t_i^X)$ and its expected execution times $E(Y_i)$ are randomly generated with uniform distribution in the range [1,30] and $[1, \alpha \cdot p_i]$, respectively, where α is defined as $max\{\frac{C_i}{p_i} | i = 1, ..., n\}$ and $Var(Y_i)$ are zero. According to [14], EDF's feasible utilization bound depends on α as well as the number of processors. It implies that no matter how many processors the system has, there exist task sets with *total utilization demand* (UD) close to one, which cannot be scheduled to satisfy all deadlines using EDF. Generally, the performance of global scheduling algorithms tends to decrease when α increases.

We consider two TUF shape patterns: (1) a homogeneous TUF class in which all tasks have downward step shaped TUFs, and (2) a heterogeneous TUF class, including downward step, linearly decreasing, and parabolic shapes. Each TUF's height is randomly generated in the range [1,100].

The number of tasks is determined depending on the given UD and the α value. We vary the UD from 3 to 6.5, including the case where it exceeds the number of processors. We set α to 0.4, 0.7, and 1. For each experiment, more than 1,000,000 jobs are released. To see the generic performance of gMUA, we assume $\{\nu_i, \rho_i\} = \{0, 1\}$.



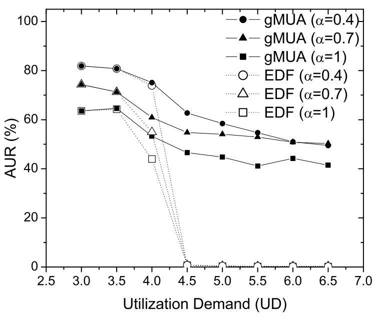
(a) AUR



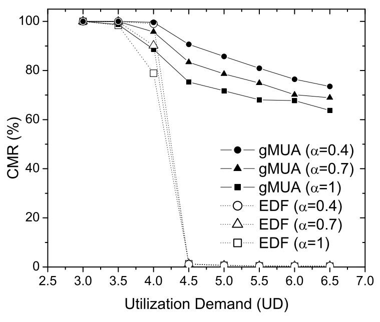
(b) CMR

Fig. 3. Performance Under Constant Demand, Step TUFs

Figures 3 and 4 show the *accrued utility ratio* (AUR) and *critical-time meet ratio* (CMR) of gMUA and EDF, respectively, under increasing UD (from 3.0 to 6.5) and for the three α values. AUR is the ratio of total accrued utility to the total maximum possible utility, and CMR is the ratio of the number of jobs meeting their critical times to the total number of job releases. For a task with a downward step TUF, its AUR and CMR are identical. But the system-level AUR and CMR can be different due to the mix of different task utilities.



(a) AUR



(b) CMR

Fig. 4. Performance Under Constant Demand, Heterogeneous TUFs

When all tasks have downward step TUFs and the total UD satisfies the global EDF's feasible utilization bound, gMUA performs exactly the same as EDF. This validates Theorem 1.

EDF's performance drops sharply after $UD = 4.0$ (for downward step TUFs), which corresponds to the number of processors in our experiments. This is due to EDF's *domino effect* (originally identified for single processors) that occurs here, when UD exceeds the number of processors. On the other hand, the performance of gMUA gracefully degrades as UD increases and exceeds 4.0, since gMUA selects as many feasible, higher PUD tasks as possible, instead of simply favoring earlier deadline tasks.

Observe that EDF begins to miss deadlines much earlier than when $UD = 4.0$, as indicated in [16]. Even when $UD < 4.0$, gMUA outperforms EDF in both AUR and CMR. This is because gMUA is likely to find a feasible or at least better schedule even when EDF cannot find a feasible one, as we have seen in Section 4.2.

We also observe that α affects the AUR and CMR of both EDF and gMUA. Despite this effect, gMUA outperforms EDF for the same α and UD for the reason that we describe above.

We observe similar and consistent trends for tasks with heterogeneous TUFs in Figure 4. The figure shows that gMUA is superior to EDF with heterogeneous TUFs and when UD exceeds the number of processors.

5.2 Performance with Statistical Demand

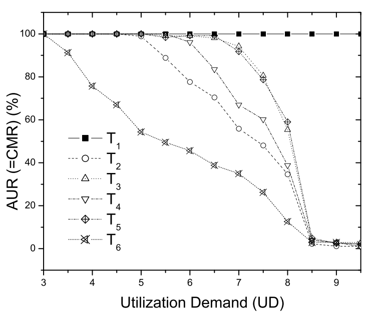
We now evaluate gMUA's statistical timeliness assurances. The task settings used in our simulation study are summarized in Table 1. The table shows the task periods and the maximum utility (or U_{max}) of the TUFs. For each task T_i's demand Y_i, we generate normally distributed execution time demands. Task execution times are changed along with the total UD. We consider both homogeneous and heterogeneous TUF shapes as before.

Table 1
Task Settings

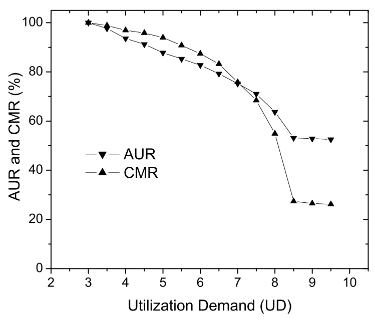
Task	p_i	U_i^{max}	ρ_i	$E(Y_i)$	$Var(Y_i)$
T_1	25	400	0.96	3.15	0.01
T_2	28	100	0.96	13.39	0.01
T_3	49	20	0.96	18.43	0.01
T_4	49	100	0.96	23.91	0.01
T_5	41	30	0.96	14.98	0.01
T_6	49	400	0.96	24.17	0.01

Figure 5(a) shows AUR and CMR of each task under increasing total UD of gMUA. For a task with downward step TUFs, task-level AUR and CMR are identical, as satisfying the critical time implies the attainment of a constant utility. But the system-level AUR and CMR are different as satisfying the critical time of each task does not always yield the same amount of utility.

Figure 5(a) shows that all tasks scheduled by gMUA accrue 100% AUR and CMR within the global EDF's bound (i.e., UD<≈2.5 here), thus satisfying



(a) Task-level



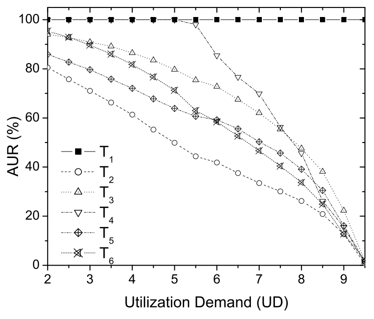
(b) System-level

Fig. 5. Performance Under Statistical Demand, Step TUFs

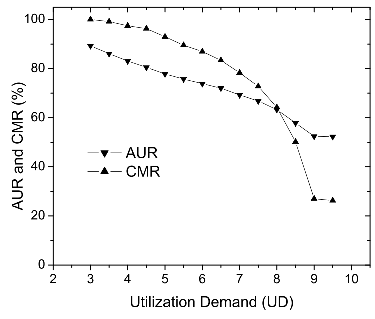
the desired $\{\nu_i, \rho_i\} = \{1, 0.96\}, \forall i$. This validates Theorem 3.

Under the condition beyond what Theorem 3 indicates, gMUA achieves graceful performance degradation in both AUR and CMR in Figure 5(b), as the previous experiment in Section 5.1 implies. In Figure 5(a), gMUA achieves 100% AUR and CMR for T_1 over the whole range of UD. This is because T_1 has a downward step TUF with higher height. Thus, gMUA favors T_1 over others to obtain more utility when it cannot satisfy the critical time of all tasks.

According to Theorem 4, the system-level AUR must be at least 96%. (For each task $T_i, \nu_i = 1$, because all TUFs are downward step shaped.) We observe that AUR and CMR of gMUA under the condition of Theorem 4 are above 99.0%. This validates Theorem 4.



(a) Task-level



(b) System-level

Fig. 6. Performance Under Statistical Demand, Heterogenous TUFs

A similar trend is observed in Figure 6 for heterogeneous TUFs. We assign downward step TUFs for T_1 and T_4, linearly decreasing TUFs for T_2 and T_5, and parabolic TUFs for T_3 and T_6. For each task T_i, ν_i is set as $\{1.0, 0.1, 0.1, 1.0, 0.1, 0.1\}$.

According to Theorem 4, the system-level AUR must be at least $0.96 \times$

$(400/25+100\times0.1/28+20\times0.1/49+100/49+30\times0.1/41+400\times0.1/49)/(400/25+100/28+20/49+100/49+30/41+400/49) = 62.5\%$. In Figure 6, we observe that the system-level AUR under gMUA is above 62.5%. This further validates Theorem 4 for non step-shaped TUFs. We also observe that the system-level AUR and CMR of gMUA degrade gracefully, since gMUA favors as many feasible, high PUD tasks as possible.

6 Conclusions and Future Work

We present a global TUF/UA scheduling algorithm for SMPs, called gMUA. The algorithm applies to tasks that are subject to TUF time constraints, variable execution time demands, and resource overloads. gMUA has the two-fold scheduling objective of probabilistically satisfying utility lower bounds for each task, and maximizing the accrued utility for all tasks.

We establish that gMUA achieves optimal accrued utility for the special case of downward step TUFs and when the total task utilization demand does not exceed global EDF's feasible utilization bound. Further, we prove that gMUA probabilistically satisfies task utility lower bounds, and lower bounds the system-wide accrued utility. We also show that the algorithm's utility lower bound satisfactions have bounded sensitivity to variations in execution time demand estimates, and that the algorithm is robust against a variant of the Dhall effect. When task utility lower bounds cannot be satisfied (due to increased utilization demand), gMUA maximizes the accrued utility.

Our simulation experiments validate our analytical results and confirm the algorithm's effectiveness. Our method of transforming task stochastic demand into actual task execution time allocation is independent of gMUA, and can be applied in other algorithmic contexts, where similar (stochastic scheduling) problems arise.

Some aspects of our work lead to directions for further research. Examples include relaxing the sporadic task arrival model to allow a stronger adversary (e.g., the unimodal arbitrary arrival model), allowing greater task utilizations for satisfying utility lower bounds, and reducing the algorithm overhead.

Acknowledgements

This work was supported by the U.S. Office of Naval Research under Grant N00014-00-1-0549 and by The MITRE Corporation under Grant 52917. E.

Douglas Jensen was supported by the MITRE Technology Program. A preliminary version of this paper appeared as “On Multiprocessor Utility Accrual Real-Time Scheduling With Statistical Timing Assurances,” H. Cho, H. Wu, B. Ravindran, and E. D. Jensen, *2006 IFIP International Conference on Embedded And Ubiquitous Computing (EUC)*, August 01-04, 2006, Seoul, Korea.

References

[1] R. K. Clark, E. D. Jensen, N. F. Rouquette, ”Software organization to facilitate dynamic processor scheduling,” *IEEE WPDRTS*, pp.122, April, 2004.

[2] R. K. Clark, E. D. Jensen, and others, ”An Adaptive, Distributed Airborne Tracking System,” *IEEE WPDRTS*, pp.353-362, April, 1999.

[3] C. L. Liu, J. W. Layland, ”Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” *Journal of the ACM*, vol.20, no.1, pp.46-61, 1973.

[4] E. D. Jensen, C. D. Locke, H. Tokuda, ”A Time-Driven Scheduling Model for Real-Time Systems”, *IEEE RTSS*, pp.112-122, Dec., 1985.

[5] D. P. Maynard, S. E. Shipman and others, ”An Example Real-Time Command, Control, and Battle Management Application for Alpha,” Archons Project TR 88121, CMU CS Dept., Dec. 1988.

[6] B. Ravindran, E. D. Jensen, P. Li, ”On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource Management,” , *IEEE ISORC*, pp.55-60, May, 2005.

[7] QNX, ”Symmetric Multiprocessing”, `http://www.qnx.com/products/rtos/smp.html`, Last accessed October 2005.

[8] Yu-Kwong Kwok and Ishfaq Ahmad, ”Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors,” ACM Computing Surveys, vol. 31, no. 4, pp. 406-471, December 1999.

[9] J. Carpenter, S. Funk and others, ”A categorization of real-time multiprocessor scheduling problems and algorithms”, *Handbook on Scheduling Algorithms, Methods, and Models*, pp.30.1-30.19, Chapman Hall/CRC, 2004.

[10] S. Baruah, N. Cohen, C. G. Plaxton, D. Varvel, ”Proportionate Progress: A Notion of Fairness in Resource Allocation,” Algorithmica, vol.15, pp.600, 1996.

[11] U. C. Devi, J. Anderson, ”Tardiness Bounds for Global EDF Scheduling on a Multiprocessor,” *IEEE RTSS*, pp.330-341, 2005.

[12] S. K. Dhall, C. L. Liu, ”On a real-time scheduling problem,” *Operations Research*, vol.26, no.1, 1978.

[13] A. Srinivasan, S. Baruah, "Deadline-based Scheduling of Periodic Task Systems on Multiprocessors", *Information Processing Letters*, pp.93-98, Nov. 2002.

[14] J. Goossens, S. Funk, S. Baruah, "Priority-Driven Scheduling of Periodic Tasks Systems on Multiprocessors," *Real-Time Systems*, vol.25, no.2-3, pp.187-205, 2003.

[15] T. P. Baker, "Multiprocessor EDF and Deadline Monotonic Schedulability Analysis," *IEEE RTSS*, pp.120-129, Dec., 2003.

[16] M. Bertogna, M. Cirinei, G. Lipari, "Improved Schedulability Analysis of EDF on Multiprocessor Platforms," *IEEE ECRTS*, pp.209-218, 2005.

[17] A. Srinivasan, J. Anderson, "Efficient scheduling of soft real-time applications on multiprocessors," *IEEE ECRTS*, pp.51-59, July, 2003.

[18] Philip Holman, James H. Anderson, "Adapting Pfair Scheduilng for Symmetric Multiprocessors", *Journal of Embedded Computing*, to appear.

[19] H. Aydin, R. Melhem, D. Mosse, P. Mejia-Alvarez, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," *IEEE RTSS*, pp.95-105, Dec., 2001.

[20] X. Zhang, Z. Wang, and others, "System support for automated profiling and optimization," *ACM SOSP*, pp.15-26, Oct., 1997.

[21] R. K. Clark, "Scheduling Dependent Real-Time Activities", Carnegie Mellon University, 1990.

[22] P. Li, B. Ravindran, and others, "A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems," *IEEE Trans. Software Engineering*, vol.30, no.9, pp.613-629, Sept., 2004.

[23] J. D. Northcutt, "Mechanisms for Reliable Distributed Real-Time Operating Systems – The Alpha Kernel," Academic Press, 1987.

[24] O. U. P. Zapata, P. M. Alvarez, "EDF and RM Multiprocessor Scheduling Algorithms: Survey and Performance Evaluation," `http://delta.cs.cinvestav.mx/~pmejia/multitechreport.pdf`, Last accessed October 2005.

[25] J. Anderson, V. Bud, U. C. Devi, "An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems," *IEEE ECRTS*, pp.199-208, July, 2005.

[26] P. Li, B. Ravindran, "A POSIX RTOS-based Middleware for Soft Real-Time Distributed Systems," *Proceedings of The IEEE International Parallel and Distributed Processing Symposium*, April, 2004.