On Multiprocessor Utility Accrual Real-Time Scheduling With Statistical Timing Assurances

Hyeonjoong Cho[1], Haisang Wu[2], Binoy Ravindran[1], and E. Douglas Jensen[3]

[1] ECE Dept., Virginia Tech, Blacksburg, VA 24061, USA {hjcho, binoy}@vt.edu
[2] Juniper Networks, Inc., Sunnyvale, CA 94089, USA, hswu@ieee.org
[3] The MITRE Corporation, Bedford, MA 01730, USA jensen@mitre.org

Abstract. We present the first Utility Accrual (or UA) real-time scheduling algorithm for multiprocessors, called 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 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. Our simulation experiments confirm our analysis and illustrate the algorithm's effectiveness.

1 Introduction

Multiprocessor architectures (e.g., Symmetric Multi-Processors or SMPs, Single Chip Heterogeneous Multiprocessors or SCHMs) are becoming more attractive for embedded systems primarily because major processor manufacturers (Intel, AMD) are making them decreasingly expensive. This makes such architectures very desirable for embedded system applications with high computational workloads, where additional, cost-effective processing capacity is often needed. But this exposes the critical need for multiprocessor real-time scheduling, which has recently received significant attention.

Pfair algorithms [3] have been shown to achieve a schedulable utilization bound (below which all tasks meet their deadlines) that equals the number of processors. Due to their higher overhead, algorithms other than Pfair (e.g., global EDF) have also been studied. With M processors, EDF's utilization bound is shown to be $M-(M-1)\,u_{max}$, where u_{max} is the maximum individual task utilization. This work was later extended for the case of deadlines less than or equal to periods in [2]. In [4], it is shown that the utilization bound in [2] does not dominate the bound in [10], and vice versa.

Timeliness objectives other than the hard real-time objective have also received attention. For example, tardiness bounds are established for a suboptimal Pfair algorithm in [14], an EDF-based partitioning scheme and scheduling algorithm in [1], and global EDF in [8].

1.1 Contributions

In this paper, we consider embedded real-time systems that 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 these systems is their relatively long execution time magnitudes—e.g., in the order of milliseconds to minutes. Some example systems that motivate our work include [6, 7].

When overloads occur, meeting deadlines of all activities is impossible as the demand exceeds the supply. The urgency of an activity is typically orthogonal to the relative importance of the activity—-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 their urgency is often desirable. Thus, a clear distinction has to be made between urgency and importance. During under-loads, such a distinction need not be made, because deadline-based scheduling algorithms such as EDF are optimal (on one processor).

Deadlines by themselves cannot express both urgency and importance. Thus, we consider the time/utility function (TUF) model that express the utility of completing an activity as a function of its completion time [12]. We specify 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.

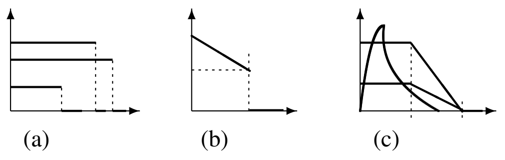


Fig. 1: Example TUFs: (a) Step TUFs; (b) AWACS TUF [6]; and (c) Air defense TUFs [13]

Many 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. Figures 1(a)-1(c) show example TUFs from two real applications. When time constraints are specified using TUFs, the scheduling criteria is 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, as UA scheduling algorithms.

We consider the problem of global UA scheduling on an SMP system with M number of identical processors. We consider global scheduling (as opposed to partitioned scheduling) because of its improved schedulability and flexibility [11]. Further, in many embedded architectures (e.g., those with no cache), its migration overhead has a lower impact on performance [4]. Moreover, applications of interest to us [6, 7] are often subject to resource overloads, during when the total application utilization demand exceed the total processing capacity of all processors. When that happens, we hypothesize that global scheduling can yield greater scheduling flexibility, resulting in greater accrued activity utility, than partitioned scheduling.

We consider repeatedly occurring application activities that are subject to TUF time constraints, variable execution times, and overloads. To account for uncertainties in activity execution behaviors, we consider a stochastic model, where activity execution demand is stochastically expressed. Activities repeatedly arrive with a known minimum inter-arrival time. For such a model, our objective is to provide statistical assurances on activity timeliness behavior.

This problem has not been previously studied. We present a polynomial-time, heuristic algorithm called the *global Multiprocessor Utility Accrual scheduling algorithm* (or gMUA). We show that gMUA achieves optimal total utility (for a special case), probabilistically satisfies individual activity utility lower bounds, and lower bounds the system-wide total accrued utility. Thus, the paper's contribution is the gMUA algorithm. We are not aware of any past 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 and report our simulation 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 **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. All tasks are assumed to be independent. 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. A task T_i's TUF is denoted by $U_i()$; thus job $J_{i,j}$'s completion at time t will yield an utility $U_i(t)$. 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 $I_{i,j}$ and a termination time $X_{i,j}$, which are the earliest and the latest times for which the TUF is defined, respectively. We assume that $I_{i,j}$ is the arrival time of job $J_{i,j}$, and $X_{i,j} - I_{i,j}$ is the period or minimal inter-arrival time P_i of the task T_i. If $J_{i,j}$'s $X_{i,j}$ 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 of job execution time demand, instead of the worst-case, because our motivating applications 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.

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., [16]). 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 accrue 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 accrue at least ν_i percentage of its maximum possible utility with the probability ρ_i. This is also the requirement of each job of T_i. Thus, for example, if $\{\nu_i, \rho_i\} = \{0.7, 0.93\}$, then T_i must accrue 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.

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

2.4 Scheduling Objective

We consider a two-fold scheduling criterion: (1) assure that each task T_i accrues 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 attained utility. We also desire to obtain a lower bound on the system-level total attained utility. Also, 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 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 [5].

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, where the sojourn time is defined as the period from the job's release to its completion. Now, task T_i's statistical timeliness requirement can be represented as $Pr(U_i(s_{i,j}) \geq \nu_i \times U_i^{max}) \geq \rho_i$. 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 "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 accrue at least the utility percentage $\nu_i = U_i(D_i)/U_i^{max}$, with the probability ρ_i.

Note that the period or minimum inter-arrival time P_i and the critical time D_i of the task T_i have the following relationships: (1) $P_i = D_i$ for a binary-valued, downward step TUF; and (2) $P_i \geq D_i$, for other non-increasing TUFs.

3.2 Bounding Utility Accrual Probability

Since task execution time demands are stochastically specified, we need to determine the actual execution time that must be allocated to each task, such that the desired utility accrual 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:

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

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 accrue ν_i percentage of its maximum utility with a probability ρ_i. To satisfy this requirement, we let $\rho_i' = \frac{(C_i - E(Y_i))^2}{Var(Y_i) + (C_i - E(Y_i))^2} \geq \rho_i$ and obtain the minimum required execution time $C_i = E(Y_i) + \sqrt{\frac{\rho_i' \times Var(Y_i)}{1 - \rho_i'}}$.

Thus, gMUA allocates C_i execution time units to each job $J_{i,j}$, so that the probability that $J_{i,j}$ requires no more than the allocated C_i 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. Supposing that each task is allocated C_i time within its P_i, the actual demand of each task often vary. Some jobs of the task may complete its execution before using up its allocated time and the others may not. gMUA probabilistically schedules the jobs of a task T_i to provide assurance ρ_i' ($\geq \rho_i$) as long as they satisfy a certain schedulability test.

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 rear of schedule σ.
- `remove(`J_k`,`σ`)` removes job J_k from schedule σ.
- `removeLeastPUDJob(`σ`)` removes job with the least *potential utility density* (or PUD) from schedule σ. PUD is the ratio of the expected job utility (obtained when job is immediately executed to completion) to the remaining job allocated execution time, i.e., 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." 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  : 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(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, which is decreasing for running jobs and a constant for unscheduled jobs. The job PUDs are then computed.

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 for completing a job after assigning each job. 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 accrue 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.

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 schedulable utilization bounds for global EDF (GFB, BAK, BCL) in [4].

Theorem 1 *Suppose that only step shaped TUFs are allowed under conditions (1) and (2). Then, a schedule produced by global EDF is also produced by gMUA, yielding equal total 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 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, since gMUA's critical times correspond to EDF's deadlines. Under conditions (1) and (2), EDF meets all deadlines. Thus, each processor always has a feasible schedule, and the if-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 *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 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 accrues ν_i utility with a probability of at least ρ_i.

Theorem 4 *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* $\frac{\rho_1\nu_1 U_1^{max}/P_1+...+\rho_n\nu_n U_n^{max}/P_n}{U_1^{max}/P_1+...+U_n^{max}/P_n}$.

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 accrue 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 $\frac{\rho_1\nu_1 U_1^{max} m_1+...+\rho_n\nu_n U_n^{max} m_n}{U_1^{max} m_1+...+U_n^{max} m_n}$. Thus, the formula comes to $\frac{\rho_1\nu_1 U_1^{max}/P_1+...+\rho_n\nu_n U_n^{max}/P_n}{U_1^{max}/P_1+...+U_n^{max}/P_n}$.

4.2 Dhall Effect

The *Dhall effect* [9] shows that there exists a task set that requires nearly 1 total utilization demand, but cannot be scheduled to meet all deadlines under global EDF and RM even with infinite number of processors. Prior research has revealed that this is caused by the poor performance of global EDF and RM when the task set contains both high utilization tasks and low utilization tasks together. This phenomena, in general, can also affect UA scheduling algorithms' performance, and counter such algorithms' ability to maximize the total attained utility. We discuss this with an example inspired from [15]. We consider the case when the execution time demands of all tasks are constant with no variance, and gMUAi estimates them accurately.

Example A. Consider $M+1$ periodic tasks that are scheduled on M processors under global EDF. Let task τ_i, where $1 \leq i \leq M$, have $P_i = D_i = 1, C_i = 2\epsilon$, and task τ_{M+1} have $P_{M+1} = D_{M+1} = 1+\epsilon, C_{M+1} = 1$. We assume that each task τ_i has a step shaped TUF with height h_i and task τ_{M+1} has a step shaped TUF with height H_{M+1}. When all tasks arrive at the same time, tasks τ_i will execute immediately and complete their execution 2ϵ time units later. Task τ_{M+1} then executes from time 2ϵ to time $1+2\epsilon$. Since task τ_{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. When $M \to \infty$, $\epsilon \to 0$, $h_i \to 0$ and $H_{M+1} \to \infty$, we have a task set, whose total utilization demand is near 1 and the maximum possible total attained utility is infinite, but that finally accrues zero total utility even with infinite number of processors. We call this phenomena as the *UA Dhall effect*. Conclusively, one of the reasons why global EDF is inappropriate as a UA scheduler is that it is prone to suffer this effect. However, gMUA overcomes this phenomena.

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 τ_{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, τ_m and τ_{M+1}. τ_m is one of τ_i where $1 \leq i \leq M$. When $h_i \to 0$ and $H_{M+1} \to \infty$, τ_m's PUD is almost zero and that of task τ_{M+1} is infinite. Therefore, gMUA removes τ_m and eventually accrues infinite utility as expected.

Under the case when *Dhall effect* occurs, we can establish *UA Dhall effect* by assigning extremely high utility to the task that will be removed by global EDF. In this sense, *UA Dhall effect* is a special case of the *Dhall effect*. It also implies that the scheduling algorithm suffering from *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 *UA Dhall effect* than global EDF can be observed in our simulation experiments (see Section 5).

4.3 Sensitivity of Assurances

gMUA is designed under the assumption that task expected execution time demands and the variances on 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 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.

Theorem 5 *Let gMUA satisfies $\{\nu_i, \rho_i\}, \forall i$, under correct $E(Y_i)$'s and their correct $Var(Y_i)$'s. When incorrect expected values, $E'(Y_i)$'s, and variances, $Var'(Y_i)$'s, are given as inputs, 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 schedulable 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 $\rho_i' \geq \rho_i$,

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

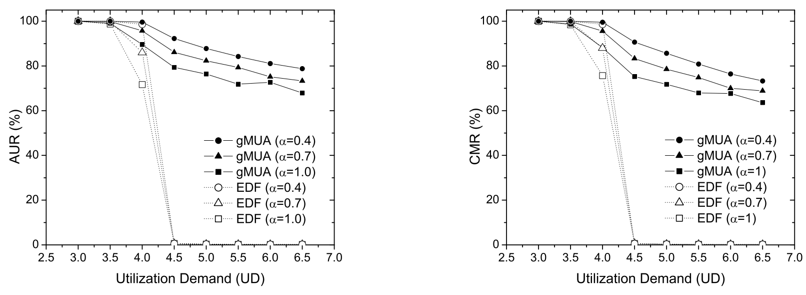
Hence, $C' \geq C_i$. From Equations 2 and 3,

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

5 Experimental Evaluation

Performance with Constant Demand. We consider an SMP machine with 4 processors. A task T_i's period $P_i(= X_i)$ and its expected execution time $E(Y_i)$ are randomly generated 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 [10], EDF's utilization bound depends on α and the number of processors, which means that irrespective of the number of processors, there exists task sets with total utilization demand (UD) close to 1.0, which cannot be feasibly scheduled under EDF. Generally, the performance of global schemes tends to decrease when α increases.

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



(a) AUR (b) CMR

Fig. 2: Performance Under Constant Demand, Step TUFs

The number of tasks are 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 1000,000 jobs are released. To see the generic performance of gMUA, we assume $\{\nu_i, \rho_i\} = \{0, 1\}$.

Figure 2 shows the AUR and CMR of gMUA and EDF, respectively, under increasing UD (from 3.0 to 6.5) and for the three α values. AUR (accrued utility ratio) is the ratio of total accrued utility to the total maximum utility, and CMR (critical time meet ratio) is the ratio of the number of jobs meeting their critical times to the total number of job releases. For a task with a step TUF, its AUR and CMR are identical. But the system-level AUR and CMR can be different due to the mix of different utility of tasks.

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

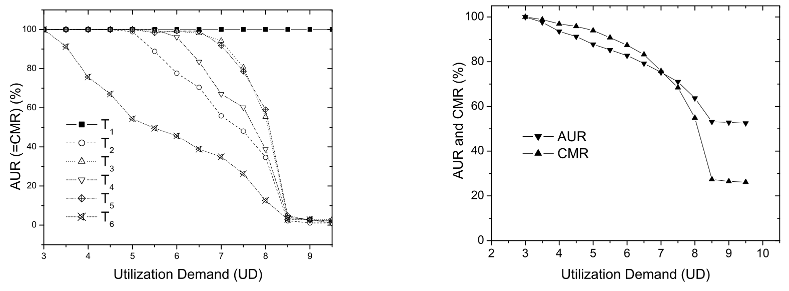
EDF's performance drops sharply after $UD = 4.0$ (for step TUFs), which corresponds to the number of processors in our experiments. This is due to EDF's domino effect that occurs when UD exceeds the number of processors. On the other hand, gMUA's performance gracefully degrades as UD increases and exceeds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

Observe that EDF begins to miss deadlines much earlier than when $UD = 4.0$, as indicated in [4]. 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 discussed in Section 4.2.

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

We observed similar and consistent trends for tasks with heterogeneous TUFs; these are omitted here for brevity.

Performance with Statistical Demand. We now evaluate gMUA's statistical timeliness assurances. 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 step and heterogeneous TUF shapes as before.



(a) Task-level (b) System-level

Fig. 3: Performance Under Statistical Demand, Step TUFs

Figures 3(a) shows AUR and CMR of each task under increasing total UD of gMUA. From the figure, we observe that all tasks under gMUA accrue 100% AUR and CMR within the global EDF's bound (i.e., UD<≈2.5 here), thus satisfying 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 3(b), as the previous experiment. In Figure 3(a), gMUA achieves 100% AUR and CMR for T_1 over all range of UD. This is because, T_1 has a 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 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 similar trend was observed for heterogeneous TUFs.

6 Conclusions and Future Work

We present a global UA scheduling algorithm for SMPs, called gMUA. The algorithm considers tasks that are subject to TUF time constraints, variable execution time demands, and resource overloads. We establish that gMUA achieves optimal total utility (for a special case), probabilistically satisfies task utility lower bounds, and lower bounds system-wide total accrued utility. We also show that gMUA'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. Our simulation experiments validate our analysis and confirm the algorithm's effectiveness.

Examples directions for further research include relaxing the sporadic task arrival model to allow a stronger adversary (e.g., the unimodal arbitrary arrival model) and allowing greater task utilizations for satisfying utility lower bounds.

References

1. J. Anderson, V. Bud, and U. C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *IEEE ECRTS*, pages 199–208, July 2005.
2. T. P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *IEEE RTSS*, pages 120–129, Dec. 2003.
3. S. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *Algorithmica*, volume 15, page 600, 1996.
4. M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *IEEE ECRTS*, pages 209– 218, 2005.
5. R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990.
6. R. K. Clark, E. D. Jensen, et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, April 1999.
7. R. K. Clark, E. D. Jensen, and N. F. Rouquette. Software organization to facilitate dynamic processor scheduling. In *IEEE WPDRTS*, April 2004.
8. U. C. Devi and J. Anderson. Tardiness bounds for global edf scheduling on a multiprocessor. In *IEEE RTSS*, 2005.
9. S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127140, 1978.
10. J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic tasks systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.
11. P. Holman and J. H. Anderson. Adapting pfair scheduilng for symmetric multiprocessors. In *Journal of Embedded Computing*, to appear.
12. 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.
13. D. P. Maynard et al. An example real-time command, control, and battle management application for alpha. Technical report, CMU CS Dept., Dec. 1988. Archons Project TR 88121.
14. A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *IEEE ECRTS*, pages 51–59, July 2003.
15. O. U. P. Zapata and 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.
16. X. Zhang, Z. Wang, et al. System support for automated profiling and optimization. In *ACM SOSP*, pages 15–26, October 1997.