Noname manuscript No.
(will be inserted by the editor)

On Scheduling Garbage Collector in Dynamic Real-Time Systems With Statistical Timing Assurances

Hyeonjoong Cho[1], Chewoo Na[1], Binoy Ravindran[1], and E. Douglas Jensen[2]

[1] ECE Dept., Virginia Tech, Blacksburg, VA 24061, USA e-mail: {hjcho, cwrha, binoy}@vt.edu
[2] The MITRE Corporation, Bedford, MA 01730, USA e-mail: jensen@mitre.org

Abstract We consider garbage collection (GC) in dynamic real-time systems. We consider the time-based GC approach of running the collector as a separate, concurrent thread, and focus on real-time scheduling to obtain assurances on mutator timing behavior, while ensuring that memory is never exhausted. We present a scheduling algorithm called GCUA. The algorithm considers mutator activities that are subject to time/utility function time constraints, variable execution time demands, the unimodal arbitrary arrival model that allows a strong adversary, and resource overloads. We establish several properties of GCUA including probabilistically-satisfied utility lower bounds for each mutator activity, a lower bound on the system-wide total accrued utility, bounded sensitivity for the assurances to variations in mutator execution time demand estimates, and no memory exhaustion at all times. Our simulation experiments validate our analytical results and confirm the algorithm's effectiveness and superiority.

1 Introduction

Traditionally, memory management in embedded real-time systems is limited to static partitioning. Although this approach is desirable (and efficient) when application's memory requirements are small and can be statically estimated (as is typically the case for many hard real-time systems), it is inflexible for dynamic application systems, which desire run-time memory allocation. Dynamic memory management is more flexible, but suffers from software engineering and product life-cycle disadvantages—e.g., programming becomes complex reducing code robustness and increasing maintenance costs. Dynamic, automatic memory management or garbage collection (GC) overcomes these problems, but introduces unpredictability on GC-pause times, which is antagonistic to timeliness optimization in real-time systems. This drawback has motivated research on real-time GC (see [10] for an excellent survey).

Real-time GC efforts can be classified into *work-based* (or *allocation-based*) and *time-based* [10]. In the work-based approach, the GC work is distributed over that of the application activities (called "mutator") by performing a bounded amount of GC work at each mutator allocation request. Examples include Baker's incremental collector [14] — one of the earliest real-time GC efforts and the basis for a number of subsequent efforts [1,4–6,18,24,30].

Though the work-based approach reduces the individual pause times, timing assurances are generally difficult to obtain with the approach, because, worst-case time bounds on each of the GC work attached to the mutator allocations must be determined, for analyzing mutator timing behavior. As Detlefs shows in [10], such time bounds are likely to be highly pessimistic. This is because, rare, but expensive GC operations must be accounted for in each allocation, to account for the worst-case GC cost. For example, in the semi-space copying collector used in Baker's work [14] and its derivative works, scanning thread stacks to identify reachable objects and copying the objects from the from-space to the to-space during a flip operation is expensive, though flips are rare. Further, mutator reads occurring immediately after a flip are likely to trigger the read-barrier, resulting in the worst-case read cost; reads at other times will cost much less, since objects will be in the to-space. Nevertheless, the worst-case GC cost must be considered for each allocation to upper bound the GC work. Such overly pessimistic analysis will likely result in infeasible real-time schedules.

In the time-based approach, the GC executes as a separate thread and is scheduled by the real-time scheduler just as another application real-time thread (i.e., mutator thread). The advantage of this approach is that the GC work is no longer coupled with each allocation, and is directly exposed to the scheduler. Thus, a mutator thread's execution time does not include GC time, since all GC operations are encapsulated in the GC thread, and consequently tightens mutator execution times to that in a system without GC. Further, the problem of obtaining timing assurances in the presence of a GC, now becomes a real-time scheduling problem: How to schedule the mutator and the GC to satisfy application time constraints, while not exhausting memory? Example works in this paradigm include [3,11,13,19,28].

In this paper, we consider garbage collection in *dynamic* real-time systems. By dynamic systems, we mean those that operate in environments, where arrival and execution time behaviors of mutator activities are subject to run-time uncertainties, causing resource overloads. Yet, such systems desire the strongest possible assurances on mutator timing behaviors — both that of individual activities and that of collective, system-wide behavior. Hard real-time assurances — i.e., all activity deadlines will always be met

— are generally impossible for these systems, as their dynamic operating conditions violate the pre-requisites needed for obtaining those assurances. Another important distinguishing feature of most of these systems (at least the ones of primary interest to us) is their relatively long activity execution time magnitudes, compared to those of conventional hard real-time subsystems—e.g., in the order of milliseconds to minutes. Some examples of such dynamic systems that motivate our work include [8, 9].

Statistical assurances (e.g., meeting all deadlines with 80% probability, meeting 95% of deadlines, missing deadlines with none more than 30% tardy, missing deadlines with a maximum mean tardiness of 20%) are appropriate for these systems, and possible, when activities exhibit non-deterministic, but stochastic behaviors.

An activity's urgency 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 resource 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 algorithms exist that can meet all deadlines (e.g., EDF on one processor).

Deadlines by themselves cannot express both urgency and importance. Thus, we consider the abstraction of time/utility functions (or TUFs) [17] that express the utility of completing an application activity as a function of that activity's completion time. We specify deadline as a binary-valued, downward "step" shaped TUF — i.e., a positive utility is accrued for completing the activity anytime before the deadline time, after which zero (or infinitively negative utility) is accrued. 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.

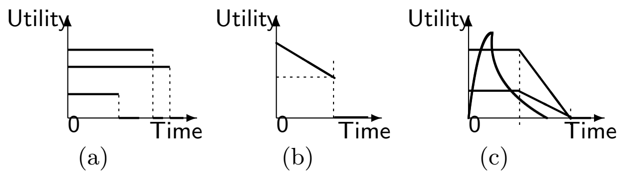


Fig. 1 Example TUF Time Constraints: (a) Step TUFs; (b) AWACS TUF [8]; and (c) Coastal Air defense TUFs [23]

Our motivating applications 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(b) and 1(c) show example such time constraints from two real applications in the defense domain [8, 23].

In this paper, we consider repeatedly occurring mutator activities that are subject to TUF time constraints and variable execution times. Activities may constitute resource overloads. To account for uncertainties in activity execution behaviors, we consider a stochastic model, where activity execution demand is stochastically expressed. To account for the variability in activity arrivals, we describe arrival behaviors using the *unimodal arbitrary arrival model* (or UAM) [15], which specifies the maximum number of arrivals that can occur (with any frequency) during any time interval. Consequently, the model subsumes most traditional arrival models (e.g., frame-based, periodic, sporadic) as special cases. We consider garbage collection in this mutator model, and in particular, time-based GC, due to its previously mentioned advantages. Similar to [28], we completely decouple the mutator and the GC, allowing for any incremental GC algorithm with fine granularity—e.g., [13].

For such a mutator model, our scheduling 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 assured lower bound on the sum of activities' attained utilities; and (3) maximize the sum of activities' attained utilities, while guaranteeing that memory is never exhausted.

This problem has not been studied in the past and is $\mathcal{NP}$-hard. We present a polynomial-time, heuristic scheduling algorithm for the problem called the *garbage collector utility accrual scheduling algorithm* (or GCUA). We prove several properties of GCUA including optimal total utility for activities during under-loads with step TUFs, lower bounds on each activity's accrued utility that are probabilistically satisfied, a lower bound on the sum of activities' attained utilities, and no memory exhaustion during both under-loads and overloads. We also show that the algorithm's assurances have bounded sensitivity to variations in activities' execution time demand estimates. Our simulation-based experiments validate our analytical results and confirm GCUA's effectiveness and superiority.

Except for [11], none of the past real-time GC efforts consider dynamic real-time systems with variable execution demand, arbitrary arrivals, overloads, and TUF time constraints. The work in [11] considers overloads and TUFs, however, no assurances on mutator timing behavior such as lower bounds on individual and collective accrued activity utility are provided. In contrast, our work precisely provides such assurances.

Thus, the contribution of the paper is the GCUA scheduling algorithm that provides assurances on (individual and collective) mutator timing behavior in dynamic real-time systems. To the best of our knowledge, we are not aware of any other efforts that solve the problem solved by GCUA.

The rest of the paper is organized as follows: In Section 2, we provide background on the GC techniques that form the basis of our work. Section 3 describes our models and scheduling objective. In Section 4, we discuss the rationale behind GCUA and present the algorithm. We establish the algorithm's properties in Section 5 and report our simulation-based experimental studies in Section 6. The paper concludes in Section 7.

2 Background

Baker's algorithm [14], one of the earliest real-time GC works, is the basis for most of the work-based approaches. The GC algorithm in Baker's work (and in its derivative works) is a variant of the *semi-space copying collector* approach [12]. In this GC paradigm, memory is partitioned into two regions called *from-space* and *to-space*, and allocation proceeds from the from-space until it is exhausted, which triggers collection. The collector copies all live objects from the from- to the to-space, compacts the objects in the to-space while copying (thereby generally occupying only a small portion of the to-space), and thus frees up the from-space. The two regions then flip their roles, and subsequent allocation proceeds from the new from-space, and the process repeats in the reverse direction.

In Baker's algorithm [14], the GC work is distributed over the mutator operation to minimize individual pause times due to the GC. This is done by ensuring that the mutator is exposed only to the to-space after a flip through a *read-barrier*: When the mutator attempts to read an object, the barrier checks whether the object is in the from-space, and if so, it is copied to the to-space, and a forwarding pointer that points to the object in the to-space from the from-space is returned (consequently increasing the cost of the original read).

Baker's approach was built upon by many others. Brooks's GC [5], a variant of Baker's algorithm, reduces the increased cost of reads (due to Baker's read-barrier), which are generally frequent, by doing the costly barrier operation on writes, which are less frequent. Thus, object copying is done only when mutator writes to objects. The Appel-Ellis-Li collector [1] uses virtual memory protection, instead of a read-barrier, to ensure that mutator always reads from, and writes to, the to-space region. Nettles and O'Toole's replicating copying collector [24] avoids a read-barrier by simultaneously updating the object copies in the two spaces. But this substantially increases the cost for writes. Cheng and Blelloch present a parallel extension of this collector in [6].

The idea of running the collector as a separate, concurrent thread has its roots in many work-based approaches—e.g., the Appel-Ellis-Li [1], Nettles and O'Toole's [24], North and Reppy's [25]. However, one of the first efforts where this was done for real-time GC with assurances on mutator timing behavior is Henriksson's work [13]. In [13], Henriksson reduces the increased cost of writes in Brooks' algorithm (due to the write-barrier), by delaying the expensive object copying until the GC runs: When the mutator writes to a from-space object, the barrier allocates space for the object in the to-space without copying the object contents, and sets up forwarding pointers that point to the object's from-space location through the allocated space in to-space. The object contents are later copied by the GC when it is scheduled and executes. This moves the copying overhead to the GC, which tightens mutator execution times and makes it closer to that in a system without GC.

Henriksson's mutator model consists of hard real-time threads and non-real-time threads. To ensure that hard real-time threads do not suffer any interference from the GC, the GC is run only when those threads are not executing, and performs an amount of work proportional to the memory allocated by them. Running the GC (essentially) as a background thread implies that the GC may not be able to, at least temporarily, "keep up" with the hard real-time threads, potentially causing memory starvation. To avoid this, an amount of memory is reserved for those threads, which is determined using modified generalized RMS [29].

In [19], Kim *et. al.* reduces the amount of memory reserved in Henriksson's work, which can potentially be large (since the GC is scheduled in the background), by modelling the GC as an aperiodic thread whose worst-case sojourn time[1] decides the worst-case memory requirement. To minimize the GCs' worst-case sojourn time, the GC is scheduled using the sporadic server (at the highest priority), while hard real-time threads are scheduled using RMS.

Robertz and Henriksson's work [28] further decouples the GC from the mutator by allowing any fine-grained, incremental GC, provided the GC's work which must be performed before memory is exhausted can be bounded (e.g., [13]). Using worst-case mutator allocation needs or online tuning techniques, they derive a deadline for the GC thread, such that satisfying the GC's deadline ensures that there will always be enough memory for mutator allocation. The GC thread is run periodically with a period equal to its deadline. Scheduling of mutator threads and the GC thread is done using RMS or EDF [16].

Our work builds upon Robertz and Henriksson's work [28]. While [28] focuses on systems with deterministic mutator arrival and execution behaviors and desire hard real-time assurances, we target dynamic systems that are subject to uncertainties in mutator behaviors and desire statistical timing assurances. In Sections 3–5, we show that [28] is a special-case of our work, both in terms of input models and in terms of output behaviors and assurances.

The time-based approach is also considered by Bacon *et. al.* in [3]. Here, fixed time quanta are assigned to the GC and the mutator, which are then scheduled in an interleaved manner for their allocated quanta. This ensures consistent CPU utilization for the mutator, toward obtaining assurances on mutator timing behavior. This in contrast to our work and [13], [19] [28], where such assurances are obtained through schedule construction by the real-time scheduler.[2] Further, [3] also reduces the GC's space overhead by

[1] The sojourn time of an activity is the time between the activity's arrival and its completion.

[2] Note that our algorithm GCUA, explicitly constructs schedules, whereas EDF and RMS (used in [13], [19] [28]) implicitly do so.

using an incremental mark-sweep algorithm with on-demand compaction, which can support high heap occupancies as there is no memory partitioning (unlike the semi-space copying collector).

3 Models and Objective

3.1 Mutator and GC Model

We consider the application to consist of a set of mutator tasks, denoted $\mathbf{M}$=$\{M_1, M_2, ..., M_n\}$, running on one processor. Each mutator task M_i has a number of instances, called jobs. Under the UAM, we associate a tuple $< m_i, W_i >$ with a mutator task M_i, which means that the maximal number of job arrivals of M_i during any sliding time window of length W_i is m_i. This model allows jobs to arrive at the same time. Note that the periodic task arrival model is a special case of the UAM with $< 1, W_i >$, where 1 is both the upper/lower bound and W_i is the period.

The j^{th} job of mutator M_i is denoted as $J_{i,j}$. All mutator tasks are assumed to be independent—i.e., they do not share any non-CPU resources or have any precedences. The basic scheduling entity that we consider is the job abstraction. Thus, we use J to denote a mutator job without being task specific, as seen by the scheduler at any scheduling event.

Similar to [28], we consider time-based GC, where the GC is run periodically, allowing for any fine-grained incremental GC algorithm. Later, in Section 4.3, we show how the GC's period is determined.

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

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 $U_{i,j}()$ has an initial time $I_{i,j}$ and a termination time $X_{i,j}$. Initial and termination times 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 mutator M_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.

3.2 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 [8,9] 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 [2], resulting in resource over-supply; and (3) allocating execution times based on the statistical estimation of tasks' demands can provide statistical timing assurances, which is sufficient for our motivating applications.

Let Y_i be the random variable of a mutator M_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., [31]). 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 mutator M_i as $E(Y_i)$, and the variance on the demand as $Var(Y_i)$.

3.3 Statistical Timeliness Requirement

We consider a task-level statistical timeliness requirement: Each mutator task must accrue some percentage of its maximum possible utility with a certain probability. For a mutator task M_i, this requirement is specified as $\{\nu_i, \rho_i\}$, which implies that M_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 M_i. Thus, for example, if $\{\nu_i, \rho_i\} = \{0.7, 0.93\}$, then M_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. Note that the hard real-time 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 mutator implies a corresponding requirement on the range of mutator sojourn times. Since we focus on non-increasing unimodal TUFs, upper-bounding task sojourn times will lower-bound task utilities.

3.4 Scheduling Objective

We consider a two-fold scheduling criterion: (1) assure that each mutator task M_i accrues the specified percentage ν_i of its maximum possible utility with at least the specified probability ρ_i; and (2) maximize the sum of the mutator tasks' attained utility, while guaranteeing that the system never runs out of memory, at all times (during both under-loads and overloads). We also desire to obtain a lower bound on the sum of the mutator tasks' attained utility, as a form of "output assurance." Also, when it is not possible to satisfy ρ_i for each mutator task (e.g., due to overloads), our objective is to maximize the 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 [7].

4 The GCUA Algorithm

4.1 Bounding Accrued Utility

Let $s_{i,j}$ be the sojourn time of the j^{th} job of mutator task M_i. Now, mutator M_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 mutator M_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, M_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 mutator M_i have the following relationships: (1) $P_i = D_i$ for a binary-valued, downward step TUF; and (2) $P_i > D_i$, for other non-increasing TUFs.

4.2 Bounding Utility Accrual Probability

Since mutator 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 mutator 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 mutator M_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 mutator M_i's execution time demands—i.e., $F_i(y) = Pr[Y_i \leq y]$. Recall that each job of mutator M_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 $C_i = E(Y_i) + \sqrt{\frac{\rho_i' \times Var(Y_i)}{1 - \rho_i'}}$.

The GCUA algorithm allocates C_i execution time units to each job $J_{i,j}$ of mutator M_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 $\rho'_i = (max\{\rho_i\})^{\frac{1}{n}}$, $\forall i$, where n is the number of mutators, so that each mutator accrues ν_i percentage of its maximum utility with a probability ρ'_i as long as all mutators satisfy the schedulability test.

4.3 Bounding GC Cycle Time Under the UAM

In [28], Robertz and Henriksson show how an upper bound on the GC cycle time can be calculated to guarantee that the application never runs out of memory under the periodic task arrival model. Building upon this result, we show how to calculate the GC cycle time under the UAM that guarantees that the GC reclaims sufficient amount of memory for the mutator. We will show that the Robertz and Henriksson's GC cycle time bound under the periodic model is a special case of our result under the UAM.

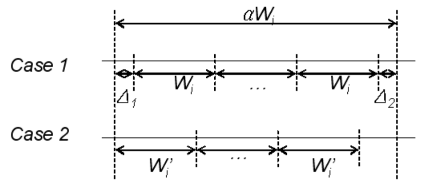


Fig. 2 UAM

Lemma 1 *If each task T_i is subject to $< m_i, W_i >$ under UAM, it is also subject to $< \lceil\alpha\rceil m_i, \alpha W_i >$ where $\alpha \geq 1$.*

Proof To compute the maximum task arrival during the extended sliding window αW_i, an array of sliding windows W_i's can be placed as the Case 1 in Figure 2 to maximize the number of W_i within the extended sliding window. We assume N number of W_i's within the αW_i. During each W_i, the maximum task arrival is m_i, but it is not clear during Δ_1 and Δ_2. Since the number of maximum task arrivals during Δ_1 and Δ_2 is m_i each, we can say that T_i is subject to $< (\lceil\alpha\rceil + 1)m_i, \alpha W_i >$. However, a much tighter bound can be calculated. For this, we consider the Case 2 where the sliding window W'_i starts with αW_i. Since $W_i = W'_i$, the UAM condition also holds during each W'_i, so that if the last window $\Delta_1 + \Delta_2$ in Case 2 is smaller than W'_i, this lemma is proved. We can prove that $\Delta_1 + \Delta_2 \geq W_i$ by contradiction. If $\Delta_1 + \Delta_2 \geq W_i$, $N + 1$ number of sliding windows are possible in Case 1. This leads to contradiction, since we assume that N number of W_i's within αW_i as the maximum.

We use the same notation as in [28]: H denotes the Heapsize, F denotes the free memory, f denotes the frequency (or period), L denotes live object, G denotes floating garbage, and a_i denotes the allocation per each job of M_i.

Lemma 2 *For a mutator task M_i, subject to $< m_i, W_i >$ under the UAM, with allocation requirement a_i bytes per job, and F bytes of available memory at the start of the GC cycle, an upper bound on the GC cycle time that guarantees that total a_i is reclaimed is given by:*

$$T_{GC} \leq \frac{F - \sum a_i m_i}{\sum f_i a_i m_i}.$$

Proof To ensure that a GC cycle completes before the available memory F has been allocated, it must hold by Lemma 1 that:

$$\sum_i \left\lceil \frac{T_{GC}}{W_i} \right\rceil \cdot m_i \cdot a_i \leq F.$$

A stronger condition is:

$$\sum_i \left(\frac{T_{GC}}{W_i} + 1 \right) \cdot m_i \cdot a_i \leq F.$$

The final equation is obtained by substituting $f_i = \frac{1}{W_i}$.

Note that Lemma 2 covers the periodic mutator case in [28]. We introduce another lemma for deriving our final solution:

Lemma 3 *Let H be the heap size and L_{max} be the maximum amount of live memory. Then, the maximum amount of memory that can be safely allocated during a GC cycle is $a_{max} = \frac{H - L_{max}}{2}$.*

Proof See [28].

Theorem 1 *An upper bound on the GC cycle time that guarantees that sufficient memory is available for allocation is given by:*

$$T_{GC} \leq \frac{\frac{H - L_{max}}{2} - \sum a_i m_i}{\sum f_i a_i m_i}.$$

Proof The theorem is proved by Lemmas 2 and 3.

In Theorem 1, when each mutator's allocation demand a_i is higher, GC cycle time becomes shorter, since GC should be invoked more frequently to reclaim more memory. If the mutator's memory allocation demand is extremely high, an appropriate T_{GC} would not be found. Thus, there exists a bound where Theorem 1 is valid. We assume that $T_{GC} > 0$. It implies that $F_{min} > \sum_i a_i m_i$ in Lemma 2, where F_{min} is the worst-case amount of memory available at the start of a GC cycle. Implicitly, we also assume that the worst-case execution time of the GC, denoted C_{GC}, does not exceed T_{GC}. Otherwise, the GC itself will constitute an overload and will not be feasible. Note that C_{GC} depends upon the particular GC algorithm, and we assume that it is given (see [19] for C_{GC} calculation, for a variant of Brooks' algorithm). Therefore, we consider the case $T_{GC} > C_{GC} > 0$.

4.4 Algorithm Description

GCUA's scheduling events include job arrival, job completion, and the expiration of a time constraint such as the arrival of a TUF's termination time. To describe GCUA, we define the following variables and auxiliary functions:

- ζ_r: current job set in the system including running jobs and unscheduled jobs.
- $\sigma_{tmp}, \sigma_{pud}, \sigma_a$: a temporary schedule; σ: the final schedule
- $J_k.C(t)$: J_k's remaining allocated execution time.
- `offlineComputing()` is computed at time $t = 0$ once. For a mutator task M_i, it computes C_i as $C_i = E(Y_i) + \sqrt{\frac{\rho_i' \times Var(Y_i)}{1-\rho_i'}}$, where $\rho_i' = (max\{\rho_i\})^{\frac{1}{n}}$. For the collector, it computes T_{GC} according to Theorem 1.
- `UpdateRAET(`ζ_r`)` updates the remaining allocated execution time of all jobs in the set ζ_r.
- `sortByPUD(`σ`)` sorts jobs in σ in the *potential utility density* (or PUD) order. 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."
- `insert(`J_c`, `σ`)` inserts the collector into schedule σ.
- `insertByECF(`J_k`, `σ`)` inserts the job J_k into σ in the earliest critical time first (or ECF) order.
- `feasible(`σ`)` returns a boolean value denoting schedule σ's feasibility
- `append(`σ`,`J_k`)` appends job J_k at rear of schedule σ; `append(`σ`,`σ_a`)` appends a schedule σ_a at rear of schedule σ.
- `headOf(`σ`)` returns the job that is at the head of schedule σ.

A description of GCUA 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 GCUA 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 largest PUD first, in line 9. Then, it inserts the collector job first into the schedule in line 10. This is to ensure that the collector job is never aborted even during overloads. But this does not mean that the collector is never preempted by mutator tasks. When a mutator job has an earlier critical time than the collector, and they are schedulable together, the mutator job will be selected by GCUA. Similar to [28], we call this GCUA's non-intrusiveness property.

In each step of the for-loop from line 11 to line 17, the job with the largest PUD is selected to be assigned to the final schedule σ. If the job inserted into the schedule in the ECF order is not feasible, then it is appended to a schedule σ_a. All jobs not inserted into the final schedule σ are temporarily stored in a schedule σ_a and then appended to σ in ECF order, in line 19.

Algorithm 1: GCUA

```
1  Input   : M={$M_1$,...,$M_n$}, $\zeta_r$={$J_1$,...,$J_N$}
2  Output: dispatched jobs, $Job_{exe}$
3  Data: $\sigma_{pud}$, $\sigma_{tmp}$, $\sigma_a$, $\sigma$
4  offlineComputing(M);
5  Initialization: $\{\sigma_{pud}, \sigma_{tmp}, \sigma_a, \sigma\} = \{0, ..., 0\}$;
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  $\sigma_{pud}$ = sortByPUD( $\zeta_r$ );
10 if $J_c \in \zeta_r$ then insert($J_c$, $\sigma$);
11 for $\forall J_k \in \sigma_{pud}$ from head to tail do
12     if $J_k.PUD > 0$ then
13         $\sigma_{tmp} = \sigma$;
14         insertByECF($J_k$, $\sigma_{tmp}$);
15         if feasible($\sigma_{tmp}$) then $\sigma = \sigma_{tmp}$;
16         else append( $J_k$, $\sigma_a$ );
17     else break;
18 sortByECF( $\sigma_a$ );
19 append( $\sigma$, $\sigma_a$);
20 $Job_{exe}$ = headOf( $\sigma$ );
21 return $Job_{exe}$;
```

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, GCUA gives the job another chance to be scheduled instead of aborting it, which eventually makes the algorithm more robust. Finally, the job at the head of σ is selected for execution.

4.5 Algorithm Time Complexity

GCUA's time cost depends upon that of the procedures `sortByPUD()`, `insertByECF()`, `feasible()`, `append()`, and `sortByECF()`. With n jobs, `sortByPUD()` and `sortByECF()` costs $O(nlogn)$. While `append()` costs $O(1)$ time, both `feasible()` and `insertByECF()` costs $O(n)$. The for-loop in line 11 iterates at most n times, costing the entire loop $O(n^2)$. Thus, the algorithm costs $O(n^2)$.

GCUA's $O(n^2)$ cost is similar to that of many past utility accrual (or UA) real-time scheduling algorithms [27]. Our prior implementation experience with UA scheduling at the middleware-level have shown that the overheads are in the magnitude of sub-milliseconds [21] (sub-microsecond overheads may be possible at the kernel-level). We anticipate a similar overhead magnitude for GCUA. Though this cost is higher than that of many traditional algorithms, the cost is justified for applications with longer execution time magnitudes (e.g., milliseconds to minutes) such as those that we focus here. Of course, this high cost cannot be justified for every application.[3]

4.6 Extension to UAGC

We calculated T_{GC} with the worst-case memory demands and periods from mutators. However, memory demands vary on run-time applications and may lead to overloaded GC work–i.e., $T_{GC} < C_{GC}$. Under overloaded GC work, the system cannot satisfy memory allocation demands from mutators. That means, that the GC will never complete before its deadline and therefore, mutators will run out of memory. One possible way is to consider that the scheduler aborts mutators with the least PUD until $T_{GC} >= C_{GC}$. In this paper, we do not evaluate this extension. Our GC scheduling algorithm assume that a GC is based on a fine-grained incremental GC algorithm and can be arbitrarily preempted at an atomic interval. We do not consider GCUA's overhead in this analysis.

5 Algorithm Properties

5.1 Timeliness Assurances

We establish GCUA's timeliness assurances under the conditions of (1) independent mutator tasks that arrive under the UAM, and (2) there is a sufficient processor capacity for meeting all task critical times—i.e., there is no overload.

Theorem 2 (Optimal Performance with Step-Shaped TUFs) *Suppose that only step-shaped TUFs are allowed under conditions (1) and (2). Then, a schedule produced by EDF [16] is also produced by GCUA, yielding equal total utilities. This is a critical time-ordered schedule.*

Proof We prove this by examining Algorithm 1. In line 14, the queue σ_{tmp} is sorted in a non-decreasing critical time order. In line 15, if σ_{tmp} is feasible, $\sigma = \sigma_{tmp}$. If no overload exists, this is precisely the EDF schedule, as GCUA's critical time is equivalent to EDF's deadline. EDF is optimal with respect to meeting all task deadlines during under-loads, yielding maximum total utility with step-shaped TUFs. Thus, GCUA produces the same schedule, and consequently, the same maximum total utility as EDF.

[3] When UA scheduling is desired with low overhead, solutions and tradeoffs exist. Examples include linear-time stochastic UA scheduling [20], and using special-purpose hardware accelerators for UA scheduling (analogous to floating-point co-processors) [26].

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

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

Corollary 2 *Under conditions (1) and (2), GCUA minimizes the maximum lateness.*

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

Proof From Corollary 1, 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 mutator task M_i's jobs, at least ρ_i' of them have lesser actual execution time than the allocated execution time. Thus, GCUA can satisfy job critical times at least with a probability of $\Pi\rho_i'$, since all mutator tasks must have lesser actual execution time than the allocated. $\Pi\rho_i' = max\{\rho_i\} \geq \rho_i, \forall i$—i.e., the algorithm accrues ν_i utility with a probability of at least ρ_i.

Theorem 4 (System-Level Utility Assurance) *Under conditions (1) and (2), if a mutator task M_i's TUF has the highest height U_i^{max}, then the system-level utility ratio, defined as the utility accrued by GCUA with respect to the system's maximum possible utility, is at least $\frac{\sum_{i=1}^{n}\rho_i\nu_i U_i^{max}}{\sum_{i=1}^{n}U_i^{max}}$.*

Proof We denote the number of jobs released by mutator M_i as l_i. Mutator M_i can accrue at least ν_i percentage of its maximum possible utility with the probability $max\{\rho_i\} \geq \rho_i$. Thus, the ratio of the system-level accrued utility to the system's maximum possible utility is $\frac{max\{\rho_i\}\nu_1 U_1^{max} l_1 + \ldots + max\{\rho_i\}\nu_n U_n^{max} l_n}{U_1^{max} l_1 + \ldots + U_n^{max} l_n}$ and it is greater than or equal to $\frac{\rho_1\nu_1 U_1^{max} l_1 + \ldots + \rho_n\nu_n U_n^{max} l_n}{U_1^{max} l_1 + \ldots + U_n^{max} l_n}$. Thus, when l_i approaches ∞, the formula converges to $\frac{\sum_{i=1}^{n}\rho_i\nu_i U_i^{max}}{\sum_{i=1}^{n}U_i^{max}}$.

5.2 Sensitivity of Assurances

GCUA 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 GCUA'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 mutator tasks M_i.

Let a mutator M_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 GCUA. Let the mutator's statistical timeliness requirement be $\{\nu_i, \rho_i\}$. We show that if GCUA 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 GCUA 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, GCUA 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 the EDF's timeliness.*

Proof We assume that if GCUA 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 GCUA 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, GCUA 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 GCUA 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 EDF's schedulability test; otherwise GCUA 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 $\rho_i'' \geq \rho_i' (\geq \rho_i)$,

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

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 3 () *Assume that GCUA 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 $Var(Y_i)$, GCUA 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 schedulability test for EDF.*

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

Corollary 3, a special case of Theorem 5, is intuitively straightforward: It essentially states that if overestimated demands are schedulable, then GCUA 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.

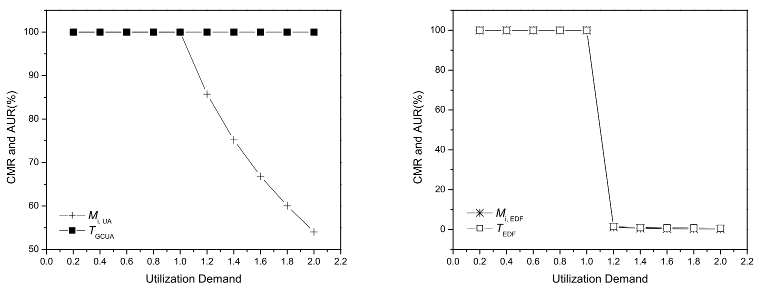
Corollary 4 () *Assume that GCUA satisfies $\{\nu_i, \rho_i\}, \forall i$, under correct, expected execution time demand estimates, $E(Y_i)$'s, and their correct variances, $Var(Y_i)$'s. When incorrect variances, $Var''(Y_i)$'s, are given as inputs instead of correct $Var(Y_i)$'s but with correct expectations $E(Y_i)$'s, GCUA 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 schedulability test for EDF.*

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

6 Experimental Evaluation

We conducted simulation-based experimental studies to validate our analytical results and to compare GCUA's performance with Robertz and Henriksson's EDF-based GC scheduling [28]. We consider two cases: (1) the execution time demand of all mutator tasks are constant (i.e., no variance) and GCUA exactly estimates the execution time allocation, and (2) the demand of all tasks statistically varies and GCUA probabilistically estimates the execution time allocation for each task. The former experiment is conducted to evaluate GCUA's generic performance as opposed to EDF, while the latter is conducted to validate the algorithm's assurances.

We consider two TUF shape patterns: (1) a homogenous TUF shape class in which all tasks have step shaped TUFs, and (2) a heterogeneous TUF shape class, including step, linearly decreasing, and parabolic shapes.



(a) GCUA (b) EDF GC Scheduling [28]

Fig. 3 Performance Under Constant Demand, Step TUFs

6.1 Performance with Constant Demand

We consider mutator tasks M_i with the timing requirement of $\{\nu_i, \rho_i\} = \{1, 0.96\}$, $i = \{1, ..., 5\}$, and a UA-scheduled garbage collection task, T_{GCUA}. On the other hand, a EDF-scheduled garbage collection task is T_{EDF}.

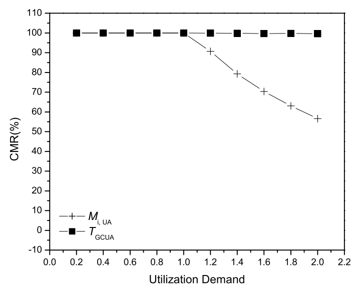
Figure 3 and 4 show the *accrued utility ratio* (or AUR) and the *critical-time meet ratio* (or CMR) of each mutator task under increasing *utilization demand* (or UD), for homogeneous and heterogenous TUF shape classes, respectively. AUR is the ratio of the total accrued utility to the maximum possible total utility, and CMR is the ratio of the number of jobs meeting their critical times to the total number of job releases. We vary the UD from 0.2 to 2.0. For tasks with step TUFs, we show AUR and CMR in the same plot since they are identical, as satisfying the critical time implies the accrual of a constant utility.

As Figure 3(a) shows, all tasks maintain 100% of CMR during underloads (UD $\leq$ 1.0), since GCUA is equivalent to EDF during this load region. This validates Theorem 2.

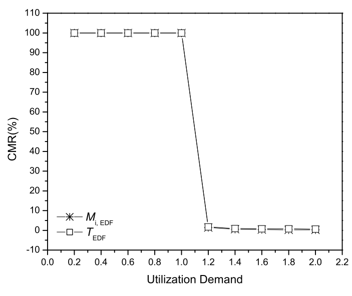
During overloads (UD $>$ 1), the CMR of all tasks gracefully decreases, except that of T_{GCUA}. GCUA ensures that the CMR of T_{GCUA} does not drop less than 100% to prevent mutator tasks from running out of memory. As described in Algorithm 1, when UD $>$ 1, GCUA selects as many, feasible, high-PUD tasks as possible, instead of selecting earlier deadline tasks. When T_{GCUA} is not released, the algorithm schedules one of the mutator tasks to maximize total utility.

In Figure 3(b), the CMR of all tasks drops sharply under EDF during overloads due to EDF's domino effect [22].

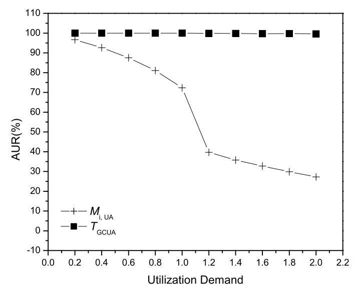
For the heterogenous TUF class (Figure 4), we observe consistent results: GCUA ensures 100% CMR (and AUR) for T_{GCUA} at all loads, obtains 100% CMR and AUR for mutator tasks at UD $\leq$ 1.0, and gracefully degrades CMR and AUR of mutator tasks at UD $>$ 1. We also observe that GCUA outperforms EDF for all TUF shapes considered, as it finds schedules that yield greater total utility than EDF.



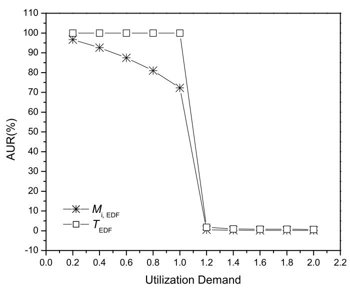
(a) CMR of GCUA



(b) CMR of EDF GC Scheduling



(c) AUR of GCUA



(d) AUR of EDF GC Scheduling

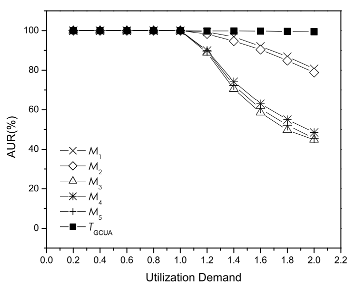
Fig. 4 Performance Under Constant Demand, Heterogeneous TUFs

6.2 Performance with Statistical Demand

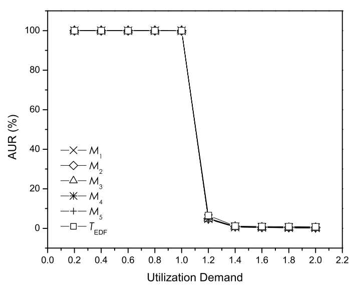
We now evaluate GCUA's statistical timeliness assurances. For each mutator task M_i's execution time demand Y_i, we generate normally distributed demands. Task execution times are changed along with the total UD. We consider both homogeneous and heterogeneous TUF shapes as before. The task settings used for homogeneous TUFs (including step TUFs only) are summarized in Table 1.

Table 1 Mutator Task Settings for Step TUFs

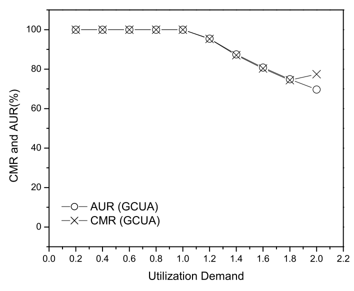
Mutators	M_1	M_2	M_3	M_4	M_5
ρ_i	0.96	0.96	0.96	0.96	0.96
ν_i	1.0	1.0	1.0	1.0	1.0
$Var(Y_i)$	0.01	0.01	0.01	0.01	0.01
U_i^{max}	386	374	386	389	360



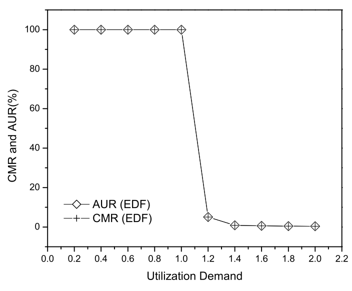
(a) GCUA



(b) EDF GC Scheduling



(c) System-level (GCUA)



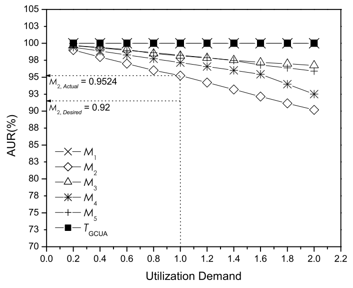
(d) System-level (EDF GC)

Fig. 5 Performance Under Statistical Demand, Step TUFs

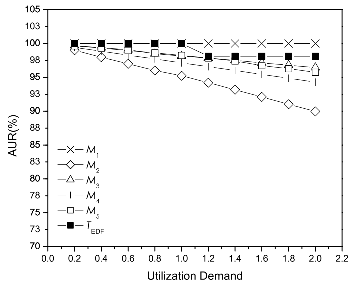
Figures 5(a) and 5(b) show AUR of each mutator task under increasing UD, of GCUA and EDF, respectively. Figure 5(c) and 5(d) shows the system-level AUR and CMR of both algorithms. From Figure 5(a), we observe that T_{GCUA} obtains 100% of AUR during under-loads and overloads, implying that no mutator task will run out of memory. On the other hand, in Figure 5(b), we observe that T_{EDF}'s AUR is nearly close to 0 during overloads, implying that memory exhaustion will likely occur (during overloads). In Figure 5(a), GCUA achieves higher AUR for M_1 over all range of UD. On the other hand, EDF achieves less AUR for M_1. This is because, M_1 has a higher U_i^{max} (i.e., a step TUF with higher height) and therefore, GCUA generally favors M_1 over others to obtain more utility when it cannot satisfy the critical time of all tasks.[4] EDF cannot make such a scheduling decision, as it favors only shorter deadline tasks. Thus, in Figure 5(b), we observe that the algorithm is unable to make utility-sensitive decisions.

As defined in Theorem 4, the system-level AUR under GCUA can be calculated as $100 \times 0.96 \times (386+374+386+389+360)/1895 = 96\%$. (For each

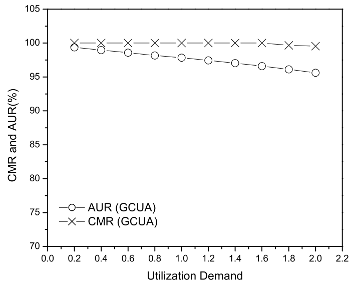
[4] Note that GCUA selects that subset of feasible high-PUD tasks (during overloads) which can yield a high collective utility. This does not mean that tasks with high U_i^{max}'s will always have high AUR/CMR, but the likelihood is generally high.



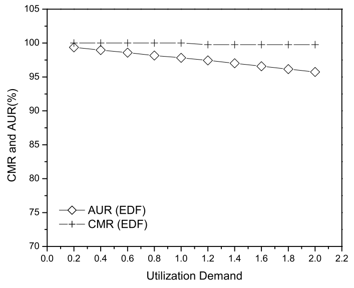
(a) GCUA



(b) EDF GC Scheduling



(c) System-level (GCUA)



(d) System-level (EDF GC)

Fig. 6 Performance Under Statistical Demand, Heterogeneous TUFs

mutator task M_i, $\nu_i = 1$, since all TUFs are step-shaped.) From Figure 5(c), we observe that the AUR of GCUA, under the condition of Theorem 4, are above 99%. This validates Theorem 4. Further, from Figure 5(c) and 5(d), we observe that GCUA accrues significantly more system-wide utility than EDF over all UD range. A similar trend is observed in Figure 6 for heterogeneous TUFs. We assign step-shaped TUF to M_1, linearly decreasing TUF to M_3 and M_4, and parabolic TUF to M_2 and M_5. The task settings used for heterogeneous TUFs are summarized in Table 2.

Table 2 Mutator Task Settings for Heterogeneous TUFs

Mutators	M_1	M_2	M_3	M_4	M_5
ρ_i	0.90	0.92	0.94	0.96	0.98
ν_i	0.85	0.85	0.85	0.85	0.85
$Var(Y_i)$	0.01	0.01	0.01	0.01	0.01
U_i^{max}	360	324	334	332	333
$TUFshape$	Step	Parabolic	Linear	Linear	Parabolic

Figures 6(a) and 6(b) show the AUR of GCUA and EDF under heterogeneous TUFs, respectively, while Figure 6(c) and 6(d) show the system-level AUR and CMR of both algorithms. From Figure 6(a), we observe that task-level assurances are ensured under GCUA. For example, $\rho_2 = 0.92$. From Figure 6(a), we observe that M_2's actual AUR is 95%. The other mutator tasks similarly meet their desired AURs during under-loads. This validates Theorem 3.

According to Theorem 4, the system-level AUR must be at least $(0.9\times360 +0.92\times324+0.94\times334+0.96\times332+ 0.98\times333)\times0.85\times100/1683 = 79.8\%$. In Figure 6(c), we observe that the actual system-level AUR of GCUA is above 79.8%. This validates Theorem 4 for non step-shaped TUFs.

7 Conclusions, Future Work

In this paper, we consider garbage collection in dynamic real-time systems. We consider the time-based GC approach of running the collector as a separate, concurrent thread, and focus on real-time scheduling to obtain assurances on mutator timing behavior, while ensuring that memory is never exhausted. We present a scheduling algorithm called GCUA. The algorithm considers mutator tasks that are subject to TUF time constraints, variable execution time demands, and resource overloads, and allows any fine-grained incremental GC algorithm. GCUA considers the two-fold scheduling objective of probabilistically satisfying utility lower bounds for each task and maximizing the total accrued utility, while ensuring that memory is never exhausted.

We establish that GCUA achieves optimal total utility for step TUFs during under-loads, probabilistically satisfies task utility lower bounds, lower bounds system-wide total accrued utility, and never exhausts memory during under-loads and overloads. We also show that the algorithm's utility lower bound satisfactions have bounded sensitivity to variations in execution time demand estimates. When task utility lower bounds cannot be satisfied during overloads, GCUA maximizes total utility by completing a subset of tasks which yields high total utility, and thereby gracefully degrades timeliness. Our simulation experiments validate our analytical results and confirm the algorithm's effectiveness and superiority.

Several aspects of our work are directions for further research. Examples include reducing the algorithm overhead, allowing mutator tasks to share non-CPU resources, and considering multiprocessor and distributed system architectures.

References

1. A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 11–20, New York, NY, USA, 1988. ACM Press.

2. H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *IEEE RTSS*, pages 95 –105, December 2001.
3. D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.
4. H. G. Baker. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Not.*, 27(3):66–70, 1992.
5. R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262, New York, NY, USA, 1984. ACM Press.
6. P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 125–136, New York, NY, USA, 2001. ACM Press.
7. R. K. Clark. *Scheduling Dependent Real-Time Activities.* PhD thesis, Carnegie Mellon University, 1990.
8. R. K. Clark, E. D. Jensen, et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, April 1999.
9. R. K. Clark, E. D. Jensen, and N. F. Rouquette. Software organization to facilitate dynamic processor scheduling. In *IEEE WPDRTS*, April 2004.
10. D. Detlefs. A hard look at hard real-time garbage collection. In *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 23– 32, May 2004.
11. S. Feizabadi and G. Back. Java garbage collection scheduling in utility accrual scheduling environments. In *Workshop on Java Technologies for Real-time and Embedded Systems*, October 2005.
12. R. R. Fenichel and J. C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, 1969.
13. R. Henriksson. *Scheduling Garbage Collection in Embedded Systems.* PhD thesis, Lund Institute of Technology, July 1998.
14. J. Henry G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.
15. J.-F. Hermant and G. L. Lann. A protocol and correctness proofs for real-time high-performance broadcast networks. In *IEEE ICDCS*, pages 360–369, 1998.
16. W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quaterly*, 21:177–185, 1974.
17. 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.
18. M. S. Johnstone. *Non-compacting memory allocation and real-time garbage collection.* PhD thesis, University of Texas at Austin, 1997. Supervisor-Paul R. Wilson.
19. T. Kim, N. Chang, N. Kim, and H. Shin. Scheduling garbage collector for embedded real-time systems. *SIGPLAN Not.*, 34(7):55–64, 1999.
20. P. Li and B. Ravindran. Fast, best effort real-time scheduling algorithms. *IEEE Transactions on Computers*, 53(9):1159 – 1175, 2004.

21. P. Li, B. Ravindran, et al. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Trans. Software Engineering*, 30(9):613 – 629, Sept. 2004.
22. C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, Carnegie Mellon University, 1986.
23. D. P. Maynard, S. E. Shipman, et al. An example real-time command, control, and battle management application for alpha. Technical report, CMU CS Dept., Dec. 1988. Archons Project TR 88121.
24. S. Nettles and J. O'Toole. Real-time replication garbage collection. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 217–226, New York, NY, USA, 1993. ACM Press.
25. S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 113–133, London, UK, 1987. Springer-Verlag.
26. J. D. Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems – The Alpha Kernel.* Academic Press, 1987.
27. B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *IEEE ISORC*, pages 55 – 60, May 2005.
28. S. G. Robertz and R. Henriksson. Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 93–102, New York, NY, USA, 2003. ACM Press.
29. L. Sha, R. Rajkumar, and J. P. Lehoczky. Generalized rate-monotonic scheduling theory. *Proceedings of the IEEE*, 82(1), 1994.
30. T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, 1990.
31. X. Zhang, Z. Wang, et al. System support for automated profiling and optimization. In *ACM SOSP*, pages 15–26, October 1997.