Garbage Collector Scheduling in Dynamic, Multiprocessor Real-Time Systems

Hyeonjoong Cho, *Member*, *IEEE*, Binoy Ravindran, *Senior Member*, *IEEE*, and Chewoo Na, *Student Member*, *IEEE*

Abstract—We consider garbage collection (GC) in dynamic, multiprocessor real-time systems. We consider the time-based, concurrent GC approach and focus on real-time scheduling to obtain mutator timing assurances, despite memory allocation and garbage collection. We present a scheduling algorithm called GCMUA. The algorithm considers mutator activities that are subject to time/utility function time constraints, stochastic execution-time and memory demands, and overloads. We establish that GCMUA probabilistically lower bounds each mutator activity's accrued utility, lower bounds the system-wide total accrued utility, and upper bounds the timing assurances' sensitivity to variations in mutator execution-time and memory demand estimates. Our simulation experiments validate our analytical results and confirm GCMUA's effectiveness.

Index Terms—Real time, garbage collection, time/utility functions, scheduling, multiprocessors.

1 Introduction

Memory management in embedded real-time systems is traditionally limited to static partitioning. This approach is desirable (and likely efficient) when application's memory requirements are small and can be statically estimated, as is often the case for many hard real-time systems. However, it is inflexible for applications whose memory demands cannot be statically estimated, and consequently, desire runtime memory allocation. Dynamic, manual memory management is more flexible, but suffers from software engineering and product life-cycle disadvantages, e.g., programming becomes complex to avoid memory leaks and dangling pointers, 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 [1] for a survey).

Contributions. In this paper, we consider garbage collection in *dynamic* multiprocessor real-time systems. By dynamic systems, we mean those that operate in environments, where arrival and execution behaviors of mutator activities are subject to runtime uncertainties, causing resource overloads. Yet, such systems desire the strongest possible assurances on mutator timing behaviors—both that of individual activities' behavior and that of collective, system-wide behavior. Statistical assurances (e.g., meeting all deadlines with 80 percent probability; meeting 95 percent of deadlines) are appropriate for these systems.

Another distinguishing feature of these systems is that their time constraints include those with *nondeadline* timeliness semantics, e.g., "earlier the better, but before a certain time." Further, an activity's urgency is sometimes orthogonal to its relative importance. Yet another key distinguishing feature of these systems is their relatively long activity execution time magnitudes, compared to those of conventional real-time systems, e.g., milliseconds to minutes.

Some examples of such systems that motivate our work include [2], [3], [4]. For example, in [2], Clark et al. discuss an AWACS tracker application which collects radar sensor reports, identifies airborne objects (or "track objects") in them, and associates those objects to track states that are maintained in a track database. Here, each instance of a track association activity must complete as soon as possible, but before a certain time. In [3], Clark et al. discuss the Mission Data System (MDS) of the NASA/JPL Mars Science Lab Rover robot application with similar timeliness semantics. Additional key features of this application include processor cycle overloads and activity time scales (e.g., frequency of constructing MDS schedules) that are of the order of minutes (AWACS [2] also has these features). In [4], Yuan and Nahrstedt discuss an MPEG video decoder which decodes video frames of different types (e.g., decoding I, P, and B frames of an MPEG video). Here, each instance of a decoder activity has a deadline, and the application's collective timeliness objective is to meet a certain percentage (e.g., 95 percent) of deadlines of each activity.

We consider a mutator model that encompasses these application features. Key aspects of our model include 1) the *time/utility function* (or TUF) timeliness model [5] that allows the specification of a broad range of time constraints, including deadlines and nondeadline time constraints, and full decoupling of activity urgency from activity importance and 2) a *stochastic mutator model*, where mutator execution

- *H. Cho is with the Department of Computer and Information Science, Korea University, 339-700, Korea. E-mail: raycho@korea.ac.kr.*
- *B. Ravindran and C. Na are with the Bradley Depatment of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, 302 Whittemore Hall, Blacksburg, VA 24061. E-mail: {binoy, cwrha}@vt.edu.*

Manuscript received 11 June 2008; revised 23 Dec. 2008; accepted 6 Jan. 2009; published online 23 Jan. 2009.
Recommended for acceptance by X. Zhang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-06-0225.
Digital Object Identifier no. 10.1109/TPDS.2009.20.

time and memory demands are probabilistically expressed, to account for uncertainties in the execution time and memory allocation behaviors.

We consider garbage collection in this mutator model and on a symmetric multiprocessor (SMP) system with N identical processors. We consider the time-based GC approach [1], where the collector executes as a separate thread and is scheduled similar to a mutator thread (we explain our rationale for this approach in Section 2). Similar to [6], we fully decouple mutators and the garbage collector, allowing for any incremental GC algorithm with fine granularity (e.g., [7]). Unlike [8], we only consider a single GC thread, as we target embedded multiprocessor platforms which typically have only a few processors (e.g., 4-16). For such a mutator and system model, our objective is to 1) provide statistical assurances on individual activity timeliness behavior; 2) provide system-level timeliness assurances; and 3) maximize the total timeliness utility attained by all activities.

This problem has not been studied in the past and is $\mathcal{NP}$-hard. We present a polynomial-time, global multiprocessor scheduling algorithm for the problem called the *garbage collector multiprocessor utility accrual scheduling algorithm* (or GCMUA). We prove several properties of GCMUA including optimal total utility for downward step TUFs, probabilistically satisfied lower bounds on each activity's accrued utility, and a lower bound on the total activity attained utility. We also show that GCMUA has bounded sensitivity for its assurances to variations in execution-time and memory demand estimates, in the sense that the assurances hold as long as the variations satisfy a sufficient condition that we present. Our simulation experiments validate our analytical results and confirm GCMUA's effectiveness.

Garbage collection on multiprocessors has a long history [8], [9], [10], [11], [12], [13], [14], [15] that started with Halstead's modification [9] of Baker's semispace copying collector algorithm [16], which is one of the earliest real-time GC works. Halstead partitions the shared memory into distinct regions for each processor, and allocation and garbage collection proceeds within each region through a semispace copying scheme. As Cheng and Blelloch show in [8], this approach can imbalance the GC work distribution across processors. In [17], Endo balances the Halstead-collector's work through a mark-and-sweep, nonincremental collector. Flood et al. extend Endo's work for a copying collector [18]. Herlihy and Moss also improve the Halstead-collector's performance through lock-free synchronization [11]. Doligez and Gonthier present a concurrent mark-and-sweep collector in [12] with no read overheads. Bacon et al. present a concurrent reference counting collector in [15].

These nonreal-time GC efforts became the basis for real-time GC on multiprocessors, which started with Blelloch and Cheng's collector [14]. Unlike Halstead's, their memory model is similar to Baker's in that the whole shared memory is divided into two spaces (from and to) and are shared among all processors. To avoid the expensive cost of reads in Baker's algorithm due to Baker's read-barrier, they use the replication scheme of Nettles and O'Toole's copying collector [13], which simultaneously updates object copies in the from and to spaces. Though this substantially increases the cost for writes, writes are less frequent than reads. Fixed bounds on pause times and memory usage are established in [14]. Cheng and Blelloch parallelizes this collector in [8] by running multiple GC threads in parallel, so that garbage collection can keep up with the increased allocation rate when mutator threads and processors increase.

None of the past real-time GC efforts, except [19], [20], consider our mutator and system model, which includes TUF time constraints, stochastic execution-time and memory demands, resource overloads, and SMPs. Both [19] and [20] consider some aspects of our model (e.g., TUFs, overloads), but are restricted to one processor. Moreover, [19] does not provide any mutator timing assurances such as lower bounds on individual and collective activity utility, and [20] is restricted to deterministic memory demands. In contrast, our work precisely provides such assurances, and allows stochastic memory demands and SMPs.

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

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

2 Background and Related Work

Real-time GC works can be broadly classified into *work-based* and *time-based* [1]. In the work-based approach, the GC work is distributed over that of the application activities (called "mutators") by performing a bounded amount of GC work at each mutator allocation request. Baker's algorithm [16] is the basis for most of the work-based approaches. Baker's algorithm (and its numerous derivatives [8], [9], [10], [13], [21], [22]) is a variant of the *semispace copying collector* approach [23]. 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-space to the to-space, compacts the objects in 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 [16], the GC work is distributed over the mutator operation to minimize individual pause times due to 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 algorithm [22], a variant of Baker's algorithm, reduces the increased cost of Baker's reads, 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 cost of the write barrier was further reduced in [7]. The Appel-Ellis-Li collector [10] 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 [13] 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 [8].

Although the work-based approach reduces individual pause times, timing assurances are generally difficult to obtain, because, worst-case time bounds on each of the GC work attached to the mutator allocations must be established. As Detlefs shows in [1], such 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 Baker's algorithm and in its derivatives, scanning thread stacks to identify reachable objects and copying them from the from-space to the to-space during a flip is expensive, though flips are rare. Such pessimistic analysis will likely result in infeasible real-time schedules.

In the time-based approach, the collector executes as a separate thread and is scheduled by the scheduler just as another 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 GC now becomes a real-time scheduling problem: How to schedule the mutator and the GC thread to satisfy mutator time constraints while not exhausting memory?

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 [10], Nettles and O'Toole's [13], North and Reppy's [24]. However, one of the first efforts where this was done for real-time GC with mutator timing assurances is Henriksson's work [7]. In [7], Henriksson reduces the increased cost of writes in Brook's algorithm by delaying the expensive object copying until the collector runs. Furthermore, the collector is run in the background, to avoid interfering with the hard real-time threads, and performs an amount of work proportional to the memory allocated by the hard real-time threads.

In [25], Kim et al. reduce the amount of memory reserved in Henriksson's work, which can potentially be large (since the collector is scheduled in the background), by modeling the collector as an aperiodic thread whose worst-case sojourn time decides the worst-case memory requirement.

Robertz and Henriksson's work [6] further decouples the collector from the mutator by allowing any fine-grained, incremental GC algorithm (e.g., [7]). However, the collector's work which must be performed before memory is exhausted must be bounded. Using worst-case mutator allocation needs, they derive a deadline for the GC thread, such that satisfying the collector's deadline ensures that there will always be enough memory for mutator allocation.

Our previous work [20] builds upon Robertz and Henriksson's work [6]. While [6] focuses on systems with deterministic mutator execution-time behaviors and desire hard real-time assurances (i.e., always satisfying all deadlines), [20] targets dynamic systems that are subject to uncertainties in mutator execution-time behaviors and desire statistical timing assurances. However, [20] is restricted to single-processor systems and deterministic memory demands. In this paper, we extend our prior work in [20] for SMPs and stochastic memory demands.

The time-based approach is also considered by Bacon et al. in [26] (for a single processor). Here, fixed time quanta are assigned to the collector and the mutator, which are then scheduled in an interleaved manner for their allocated quanta. This ensures consistent CPU utilization for the mutator, yielding timing assurances. This is in contrast to our work and [6], [7], [20], [25], where such assurances are obtained through real-time schedule construction.

3 Models and Objective

3.1 Mutator and GC Model

The application consists of a set of mutator tasks, denoted $\mathbf{M} = \{M_1, M_2, \ldots, M_n\}$. Each mutator task M_i has a number of instances, called jobs. Jobs may be released periodically or sporadically with a known minimum interarrival time. The jth job of mutator M_i is denoted as $J_{i,j}$. A mutator M_i's period or minimum interarrival time is denoted as P_i.

All mutator tasks are assumed to be independent, i.e., they do not have dependencies (e.g., due to synchronization). Note that this is true in all our motivating applications [2], [3], [4] and past works [6], [7], [26] on which our work builds upon. Our basic scheduling entity is the job abstraction. We use J to denote a mutator job without being task specific.

Similar to [14], we consider a memory model where the entire shared memory is divided into from- and to-spaces and shared among all processors. Similar to [6], we consider time-based GC, where the collector is run periodically, allowing for any fine-grained incremental GC algorithm, e.g., [7]. (In Section 4.3, we show how the collector's period is determined.)

3.2 Timeliness Model

We consider the TUF model [5] for specifying a mutator task's time constraint. A task's TUF specifies the utility of completing it as a function of its completion time. The classical deadline is a TUF's special case—a binary-valued, downward "step" shaped TUF; Fig. 1a shows an example. Note that a task's TUF decouples its 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.

As previously mentioned, our motivating applications also have tasks with *nondeadline* time constraints, such as those where the utility attained for task completion *varies* (e.g., decreases, increases) with completion time. Figs. 1b and 1c show example such time constraints from applications in the defense domain (see [2], [27] for application details).

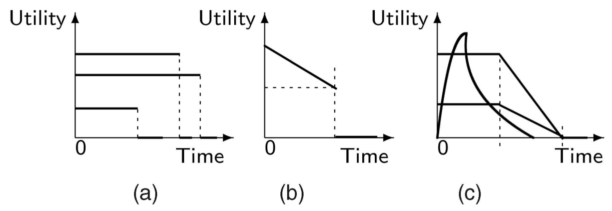


Fig. 1. Example TUFs: (a) step TUFs; (b) TUF of an Airborne Warning And Control System (AWACS) [2]; and (c) TUFs of a Coastal Air Defense System [27].

Jobs of the same mutator task have the same TUF time constraint. Mutator M_i's TUF is denoted as U_i. Thus, job $J_{i,j}$'s completion at time t will yield a utility $U_{i,j}(t)$. We focus on *nonincreasing* unimodal TUFs (e.g., Figs. 1a and 1b; two TUFs in Fig. 1c) as they include most of our motivating time constraints.

Each TUF $U_{i,j}$ 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. $I_{i,j}$ is the arrival time of job $J_{i,j}$, and $X_{i,j} - I_{i,j}$ is the period or minimum interarrival time P_i of M_i.

If a job has not completed by its termination time, a time constraint violation exception is raised, and the job is immediately aborted by executing an (application-supplied) exception handler. Before aborting the job, the handler is assumed to perform compensations and recovery actions for avoiding system inconsistencies and for ensuring the safety and stability of the external state.

3.3 Execution Time and Memory Demands

We estimate the statistical properties (e.g., mean, variance) of job execution-time and memory allocation demands rather than the worst-case demands because our motivating applications exhibit a large variation in their *actual* workload. Thus, the statistical estimation of the demand is more stable, and hence, more predictable than the actual workload.

Let Y_i and X_i be the random variables of a mutator M_i's execution-time demand and memory allocation demand, respectively. Estimating the execution-time and memory demand distributions of the task involve 1) profiling its execution-time and memory allocation usage and 2) deriving the probability distribution of that usage. We assume that the means and variances of Y_i and X_i are finite and determined through offline or online profiling (e.g., [4]).

We denote the *expected* execution-time and memory allocation demands of a mutator M_i as $E(Y_i)$ and $E(X_i)$, respectively, and the corresponding variance on the demands as $Var(Y_i)$ and $Var(X_i)$.

3.4 Statistical Timeliness Requirement

Each mutator task has a statistical timeliness requirement. For a 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 utility with the probability ρ_i. This is also the requirement of each job of M_i. For example, if $\{\nu_i, \rho_i\} = \{0.7, 0.93\}$, then M_i must accrue at least 70 percent of its maximum utility with a probability no less than 93 percent. For step TUFs, ν can only be 0 or 1. Thus, the objective of always meeting all task deadlines is the special case: $\{\nu_i, \rho_i\} = \{1.0, 1.0\}$.

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 nonincreasing unimodal TUFs, upper-bounding task sojourn times will lower bound task utilities.

3.5 System and Scheduling Models

We consider a SMP with N number of processors and the global multiprocessor scheduling approach. In that scheduling approach, a single shared scheduling queue is maintained for all processors and a processor-wide scheduling decision is made by a global scheduling algorithm, allowing arbitrary job migration across processors. This paradigm has several advantages over other approaches (e.g., partitioned scheduling) that are of interest to us, including the potential for greater scheduling flexibility during overloads and, thus, the potential for greater accrued utility, and lower migration overheads on chips with shared on-chip caches (such chips are anticipated in future embedded multiprocessor platforms) [28].

3.6 Scheduling Objective

We consider a twofold scheduling criterion: 1) assure that each mutator task M_i accrues ν_i percentage of its maximum utility with at least the probability ρ_i and 2) maximize the total task attained utility. We also desire to obtain a lower bound on the total task attained utility. Also, when it is not possible to satisfy ρ_i (e.g., due to CPU/memory overloads), our objective is to maximize the total attained utility.

This problem is $\mathcal{NP}$-hard because it subsumes the $\mathcal{NP}$-hard problem of scheduling step TUF-shaped tasks on one processor [29].

4 The GCMUA Algorithm

4.1 Bounding Accrued Utility

Let $s_{i,j}$ be the sojourn time of the jth job of mutator task M_i. M_i's statistical timing 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 nonincreasing and all jobs of the same mutator task have the same TUF, it is sufficient to have $Pr(s_{i,j} \leq D_i) \geq \rho_i$, where D_i is the upper bound on M_i's sojourn time. We call D_i "critical time," and is calculated as $D_i = U_i^{-1}(\nu_i \times U_i^{max})$, where $U_i^{-1}(x)$ denotes the inverse function of the 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 interarrival 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 nonincreasing TUFs.

4.2 Estimating Execution and Memory Demands

Since execution time demands are statistically specified, we need to estimate the execution time and memory demands that must be allocated to each mutator, such that the desired utility accrual probability ρ_i is satisfied. Further, each mutator's demand must account for the uncertainty in both execution time and memory demand specifications (i.e., the variance factors).

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

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

From a probabilistic point of view, (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 $\epsilon_i = \frac{(C_i - E(Y_i))^2}{Var(Y_i)+(C_i-E(Y_i))^2}$ and obtain the minimum required execution time $C_i = E(Y_i) + \sqrt{\frac{\epsilon_i \times Var(Y_i)}{1-\epsilon_i}}$.

Thus, GCMUA allocates C_i execution time units to each job $J_{i,j}$ of mutator M_i, so that the probability that $J_{i,j}$ requires no more than the allocated C_i time units is at least ϵ_i.

The memory demand can be similarly estimated. Given the mean and the variance of a mutator M_i's memory allocation demand X_i, we have

$$Pr[X_i < A_i] \geq \frac{(A_i - E(X_i))^2}{Var(X_i) + (A_i - E(X_i))^2}. \quad (2)$$

To satisfy a mutator's memory demand, we let $\mu_i = \frac{(A_i-E(X_i))^2}{Var(X_i)+(A_i-E(X_i))^2}$ and obtain the upper bound of memory demand $A_i = E(X_i) + \sqrt{\frac{\mu_i \times Var(X_i)}{1-\mu_i}}$. Hence, the probability that each M_i requires no more than the allocated A_i bytes is at least μ_i.

Lemma 1. *With probabilities ϵ_i and μ_i for execution-time and memory demands, respectively, if $\prod_i \epsilon_i \times \prod_i \mu_i \geq \max_i\{\rho_i\}$, then M_i accrues at least ν_i percentage of the utility with the probability ρ_i.*

Proof. For a mutator M_i with a probability ϵ_i and a probability μ_i, the least probability satisfying all execution time demands is given by $\prod_i \epsilon_i$, since each probability is exclusive. Likewise, the least probability satisfying all memory demands is given by $\prod_i \mu_i$. Let $\prod_i \epsilon_i \times \prod_i \mu_i = \Phi_i$. Φ_i produces the least probability satisfying both execution-time and memory demands. Thus, M_i accrues at least ν_i percentage of its utility with the probability ρ_i, when Φ_i is greater than or equal to the maximum ρ_i. □

4.3 Bounding GC Cycle Time

In [6], Robertz and Henriksson develop an upper bound on the GC cycle time that ensures that the application never runs out of memory:

Lemma 2 (from [6]). *For a mutator M_i with a period P_i (or frequency $f_i = 1/P_i$) which allocates no more than a_i bytes per job and F bytes of available memory at the start of the GC cycle, a GC cycle time upper bound that ensures that a_i is reclaimed is*

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

Since it may not be possible to use all F bytes in the current cycle due to floating garbage, Robertz and Henriksson bound the memory that can be safely allocated during a cycle.

Lemma 3 (from [6]). *With a heap size H and maximum live memory L_{max}, the maximum memory that can be safely allocated during a GC cycle is given by $a_{max} = \frac{H-L_{max}}{2}$.*

a_{max} is then used to bound the GC cycle time.

Theorem 4 (from [6]). *An upper bound on the GC cycle time that ensures no memory exhaustion is*

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

In [6], the T_{GC} upper bound is used as the collector's deadline and period, since satisfying that deadline for the collector ensures no memory exhaustion.

We compute T_{GC} using Theorem 4, but use the probabilistic demand A_i determined from (2) instead of the worst-case demand a_i assumed in [6], since we consider statistical properties of memory demands instead of worst case. By doing so, and satisfying the resulting T_{GC} deadline for the collector at runtime ensures that the actual mutator memory demand is satisfied at runtime, as long as that demand does not exceed A_i.

Equation (2) ensures that the probability that each M_i requires no more than A_i bytes is at least μ_i. Thus, there is $(1 - \mu_i)$ probability that M_i requires more than A_i bytes. When the actual memory demand at runtime exceeds A_i, causing "memory overloads," an appropriate T_{GC} would not be found. Hence, in order to make the collector task satisfy its deadline T_{GC}, we need to dynamically calculate T_{GC} using the actual runtime demand and check for the collector's feasibility. If infeasible, some mutator tasks will have to be aborted (by executing its exception handler) to ensure the collector's feasibility. The process of selecting the mutator tasks for abortion is described in the following section.

4.4 Algorithm Rationale and Design

To assure that each mutator M_i accrues ν_i percentage of its utility with a probability ρ_i, a schedule must be constructed such that it will ensure the completion of M_i before its critical time D_i, where the execution time and memory demands for M_i are assumed to be no larger than C_i time units and A_i bytes, respectively. Such a schedule will automatically satisfy $\{\nu_i, \rho_i\}, \forall M_i$ (per Section 4.1 and Lemma 1).

A reasonable approach for constructing such a schedule is to first approximate the global EDF schedule, where job critical times correspond to global EDF's deadline. By doing so, global EDF's schedulable utilization bound can be exploited to meet job critical times, thereby satisfying $\{\nu_i, \rho_i\}, \forall M_i$.

However, it is possible that the total expected mutator utilization demand $\sum_i C_i/D_i$, or the total actual mutator utilization demand at runtime can exceed global EDF's schedulable bound.[1] Further, the (expected or actual) total utilization demand can even exceed the total capacity of all processors. Moreover, memory overloads can occur, when some mutators' memory demand at runtime exceed their

1. The total actual mutator utilization demand at runtime can exceed the total expected demand because a mutator's actual execution time demand can exceed its allocated execution time. By (1), each M_i needs no more than C_i time units with ϵ_i probability; thus M_i needs more than C_i time units with $(1 - \epsilon_i)$ probability.

allocation, causing the collector to become infeasible, and thereby the mutators become infeasible.

Recall that when it is not possible to satisfy ρ_i for each M_i, our objective is to maximize the total task accrued utility. A reasonable heuristic towards doing so is a "greedy" strategy such as favoring "high return" jobs—jobs that can significantly contribute toward the total utility—over low return ones, and completing as many such high return jobs as possible before the job termination times. This will increase the likelihood of maximizing the total accrued utility.

The potential utility that can be accrued by executing a job defines a measure of its "return on investment." GCMUA measures this using a metric called the *Potential Utility Density* (or PUD). The PUD of a job measures the amount of utility that can be accrued per unit time by executing the job, and is computed as the ratio of the job's attained utility (obtained when the 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, GCMUA adopts the strategy of approximating global EDF until a utilization demand overload or memory overload occurs. To approximate global EDF, the algorithm examines jobs in the earliest termination time order, and assigns each job to a processor that yields the shortest sum of allocated execution times of all jobs in that processor's local schedule. 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. This will establish the same schedule as that of global EDF.

When an overload occurs, the algorithm discards low PUD jobs until the overload is removed and a feasible schedule is obtained. For each job that is discarded due to memory overloads, the algorithm immediately aborts the job by executing its exception handler (as otherwise the collector will be unable to feasibly complete, potentially causing a "domino" effect by jeopardizing the execution of all mutator tasks). For jobs that are discarded due to utilization demand overloads, GCMUA appends them onto the feasible schedule, and reconsiders them for execution, by seeking to exploit slack that may become available when some mutator tasks need less than their allocated execution times and complete early. Such jobs are aborted when they still do not complete by their termination times.

4.5 Algorithm Description

GCMUA's scheduling events include job arrival, job completion, the expiration of a TUF termination time, and memory allocation request.

To facilitate GCMUA's description, we define a set of variables. The set of mutator tasks is denoted by $\mathbf{M} = \{M_1, .., M_n\}$. Let ζ_r denote the current job set in the system including running jobs and unscheduled jobs. Let σ_{tmp} and σ_a denote temporary schedules. σ_i denotes the schedule for processor i, where $i \leq N$ and N is the number of processors of the SMP system.

T_{GC} denotes the GC cycle time upper bound.

$J_k.X$ denotes job J_k's termination time, and $J_k.C(t)$ denotes J_k's remaining allocated execution time at time t. For the collector, we assume that its worst-case execution time, C_{GC}, is known. C_{GC} depends upon the particular GC algorithm (see [25] for C_{GC} calculation, for a variant of Brook's algorithm [22]). Note that $T_{GC} < C_{GC}$ due to memory overloads.

For convenience, we also define a set of auxiliary functions. `offlineComputing(M)` is a procedure that is invoked at time $t = 0$ once. For each mutator $M_i \in \mathbf{M}$, this procedure computes $C_i = E(Y_i) + \sqrt{\frac{\epsilon_i \times Var(Y_i)}{1-\epsilon_i}}$. The procedure also computes T_{GC} using Theorem 4 and using probabilistic memory demand A_i determined from (2).

`UpdateRAET`(ζ_r) updates the remaining allocated execution time of all jobs in set ζ_r, and `UpdateRAMD`(ζ_r) updates the remaining allocated memory demand of all jobs in set ζ_r.

`ComputeTgc`(σ_{tmp}) calculates T_{GC} with estimated memory demand A_i and actual memory demand a_i.

`feasible`(σ_i) returns a boolean value denoting schedule σ_i's feasibility, and `feasible`(J_k) denotes job J_k's feasibility. For a schedule σ_i (or job J_k) to be feasible, the predicted completion time of each job in σ (or J_k) must not exceed its termination time.

`findProcessor()` returns the ID of that processor on which the currently assigned mutator tasks have the shortest sum of allocated execution times.

`removeLeastPUDJob`(σ_i) removes a job with the least PUD from schedule σ_i. PUD of a job J_k is $\frac{U_k(t+J_k.C(t))}{J_k.C(t)}$. The procedure returns the removed job. Note that when $T_{GC} < C_{GC}$, the collector task is not removed. Instead, the next least PUD job is removed.

`append`(J_r,σ_a) appends J_r at the rear of schedule σ_a. `headOf`(σ_i) returns the set of jobs that are at the head of schedule σ_i, $1 \leq i \leq N$.

A description of GCMUA 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 $t = 0$. The procedure computes T_{GC} with the memory demand A_i.

When GCMUA is invoked, it updates the remaining allocated execution-times and memory demands of each job (lines 6 and 7). The remaining allocated execution times and memory demands of running jobs are decreasing, while those of unscheduled jobs remain constant. The algorithm then computes the PUDs of all jobs. The jobs are then sorted in the order of earliest termination time first (or EXF), in line 10. In each step of the *while-loop* from line 12 to line 14, T_{GC} is computed and checked for memory overloads (i.e., $T_{GC} \leq 0$). The algorithm removes the overload by removing the least PUD job until $T_{GC} > 0$.

Algorithm 1: GCMUA

```
1  Input    : $\mathbf{M}$={$M_1$,..,$M_n$}, $\zeta_r$={$J_1$,..,$J_m$}, $N$: # of
              processors
2  Output   : array of dispatched jobs to processor $p$,
              $Job_p$
3  Data: {$\sigma_1, .., \sigma_N$}, $\sigma_{tmp}$, $\sigma_a$, $T_{GC}$, $J_r$
4  offlineComputing($\mathbf{M}$);
5  Initialization: $\{\sigma_1, ..., \sigma_N\} = \{0, ..., 0\}$;
6  UpdateRAET($\zeta_r$);
7  UpdateRAMD($\zeta_r$);
8  for $\forall J_k \in \zeta_r$ do
9      $J_k.PUD = \frac{U_k(t+J_k.C(t))}{J_k.C(t)}$;
10 $\sigma_{tmp}$ = sortByEXF($\zeta_r$);
11 $T_{GC}$ = computeTgc($\sigma_{tmp}$);
12 while $T_{GC} < 0$ and !IsEmpty($\sigma_{tmp}$) do
13     removeLeastPUDJob($\sigma_{tmp}$);
14     $T_{GC}$ = computeTgc($\sigma_{tmp}$);
15 for each $J_k \in \sigma_{tmp}$ from head to tail do
16     if $J_k.PUD > 0$ or $J_k$ is garbage collection task
       then
17         $p$ = findProcessor();
18         append($J_k$, $\sigma_p$);
19 for $i = 1$ to $N$ do
20     $\sigma_a$ = null;
21     while !feasible( $\sigma_i$) and !IsEmpty($\sigma_i$) do
22         $J_r$ = removeleastPUD($\sigma_i$);
23         append($J_r$, $\sigma_a$);
24     sortByEXF($\sigma_a$);
25     $\sigma_i$ += $\sigma_a$;
26 {$Job_1, ..., Job_N$} = headOf({$\sigma_1, ..., \sigma_N$});
27 return {$Job_1, ..., Job_N$};
```

In each step of the *for-loop* from line 15 to line 18, the job with the earliest termination 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 (`findProcessor()`). As previously explained, this processor results in the nearest scheduling event for completing a job after assigning each job, establishing a global EDF schedule. Then, the job J_k with the earliest termination time is inserted into the local schedule σ_p of the selected processor p.

In the *for-loop* from line 19 to line 25, GCMUA attempts to make each local schedule feasible by removing the lowest PUD job. In line 21, if σ_i is not feasible, then GCMUA removes the job with the least PUD from σ_i until σ_i becomes feasible. All removed jobs are temporarily stored in a schedule σ_a and then appended to each σ_i in EXF order. Note that simply aborting the removed jobs may result in decreased accrued utility. This is because the algorithm may decide that a job is feasible which is estimated to have a longer allocated execution time than its actual one, causing the job to complete earlier than expected. To exploit such slacks, GCMUA gives removed jobs another chance to complete instead of aborting it, which eventually makes the algorithm more robust.

Each job at the head of $\sigma_i, 1 \leq i \leq N$, is dispatched for execution on the respective processor.

5 Algorithm Properties

5.1 Timeliness Assurances

We establish GCMUA's timeliness assurances under the conditions of: 1) independent tasks that arrive periodically/sporadically and 2) task utilization demand satisfies any of global EDF's schedulable utilization bounds, i.e., GFB, BAK, or BCL in [30].

Theorem 5. *Suppose that only binary-valued step TUFs are allowed under conditions 1) and 2). Then, a schedule produced by global EDF is also produced by GCMUA, yielding equal total utilities. This is a termination time-ordered schedule.*

Proof. We prove this by examining Algorithm 1. In line 10, the queue σ_{tmp} is sorted in a nondecreasing termination time order. In line 17, `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) $n \leq N$ and 2) $n > N$.

When $n \leq N$, the result is trivial. GCMUA's schedule of tasks on each processor is identical to EDF (every processor has a single task or none assigned). When $n > N$, task M_i ($N < i \leq n$) 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 M_{i-1}, so that the event that will assign M_i will occur by this completion. Note that tasks in σ_{tmp} are selected to be assigned to processors according to EXF. This is precisely the global EDF schedule, as a TUF termination time is equivalent to EDF's deadline. Under conditions 1) and 2), EDF meets all deadlines. Thus, each processor always has a feasible schedule, and the while-loop from line 21 to line 23 will never be executed. Thus, GCMUA produces the same schedule as global EDF. □

An important corollary about GCMUA's timeliness behavior can be deduced from EDF's behavior.

Corollary 6. *Under conditions 1) and 2), GCMUA always completes the allocated execution time of all tasks before their critical times.*

Theorem 7 (Statistical Task-Level Assurance). *Under conditions 1) and 2), GCMUA meets the statistical timeliness requirement $\{\nu_i, \rho_i\}, \forall M_i$.*

Proof. From Corollary 6, all allocated execution times of tasks can be completed before their critical times. Further, based on the results of (1), among the actual processor time of mutator M_i's jobs, at least ϵ_i of them have lesser actual execution time than the allocated execution time. Thus, GCMUA satisfies $\{\nu_i, \rho_i\}, \forall M_i$ at least with a probability of $\epsilon_i = max\{\rho_i\}$, since all tasks must have lesser actual execution time than the allocated. $\prod \epsilon_i = max\{\rho_i\} \geq \rho_i, \forall M_i$, i.e., GCMUA accrues ν_i utility with at least ρ_i probability. □

Theorem 8 (System-Level Utility Assurance). *Under conditions 1) and 2), if a mutator M_i has the highest utility U_i^{max}, then the ratio of the total utility accrued by GCMUA to the maximum possible total utility is at least $\frac{\sum_{i=1}^{n} \rho_i \nu_i U_i^{max}}{\sum_{i=1}^{n} U_i^{max}}$.*

Proof. Let the number of jobs released by M_i be m_i. M_i can accrue at least ν_i percent of its maximum utility with the probability $max\{\rho_i\} \geq \rho_i$. Thus, the ratio of the total accrued utility to the maximum total utility is $\frac{max\{\rho_i\}\nu_1 U_1^{max} l_1 + \cdots + max\{\rho_i\}\nu_n U_n^{max} l_n}{U_1^{max} l_1 + \cdots + U_n^{max} l_n}$. This is greater than or equal to $\frac{\rho_1\nu_1 U_1^{max} l_1 + \cdots + \rho_n\nu_n U_n^{max} l_n}{U_1^{max} l_1 + \cdots + U_n^{max} l_n}$. 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 Dhall Effect

The Dhall effect [31] shows that there exists a task set that requires a total utilization demand of nearly 1, but cannot be scheduled to meet all deadlines under global EDF and RM even with infinite number of processors. Previous work has shown 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. This phenomenon, in general, can also affect TUF schedulers. We discuss this with an example inspired from [32] that considers tasks with constant execution time demands, and GCMUA accurately estimating those demands.

Example 1. Consider N+1 periodic tasks that are scheduled on N processors under global EDF. Let task τ_i, where $1 \leq i \leq N$, have $P_i = D_i = 1$, $C_i = 2\epsilon$, and task τ_{N+1} have $P_{N+1} = D_{N+1} = 1 + \epsilon$, $C_{N+1} = 1$. We assume that each task τ_i has a step-shaped TUF with maximum utility U_i^{max} and task τ_{N+1} has a step-shaped TUF with maximum utility U_{N+1}^{max}. When all tasks arrive at the same time, tasks τ_i will execute immediately and complete their execution 2ϵ time units later. Task τ_{N+1} then executes from time 2ϵ to time $1 + 2\epsilon$. Since task τ_{N+1}'s critical time—we assume here it is the same as its period—is $1 + \epsilon$, it begins to miss its critical time. By letting $N \rightarrow \infty$, $\epsilon \rightarrow 0$, $U_i^{max} \rightarrow 0$, and $U_{N+1}^{max} \rightarrow \infty$, 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 phenomenon as the *Utility Accrual Dhall effect* (or UA Dhall effect). Conclusively, one of the reasons why global EDF is inappropriate as a TUF scheduler is that it is prone to suffer this effect. However, GCMUA overcomes this phenomena.

Example 2. Consider the same scenario as in Example 1, but now, let the task set be scheduled by GCMUA. In Algorithm 1, GCMUA first tries to schedule tasks like global EDF, but it will fail to do so as we saw in Example 1. When GCMUA finds that τ_{N+1} will miss its critical time on processor m (where $1 \leq m \leq N$), the algorithm will select a task with lower PUD on processor m for removal. On processor m, there are two tasks, τ_m and τ_{N+1}. τ_m is one of τ_i, where $1 \leq i \leq N$. When $U_i^{max} \rightarrow \infty$ and $U_{N+1}^{max} \rightarrow \infty$, the PUD of task τ_m is almost zero and that of task τ_{N+1} is infinite. Therefore, GCMUA 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 selected (and will miss its deadline) by global EDF. It also implies that the scheduling algorithm suffering from Dhall effect will likely suffer from UA Dhall effect, when it schedules tasks with TUF time constraints. The fact that GCMUA is more robust against UA Dhall effect than global EDF can be observed in our experiments.

5.3 Sensitivity of Timeliness Assurances

GCMUA is designed in the same probabilistic framework as the algorithms in [20], [33]. Consequently, GCMUA inherits the sensitivity property on timeliness assurances that these prior algorithms provide. That is, GCMUA assumes that $\{E(Y_i), Var(Y_i)\}$ and $\{E(X_i), Var(X_i)\}$ are correct. However, it is possible that these inputs may change over time (e.g., due to changes in application's execution context). To understand GCMUA's behavior when this happens, we assume that $E(Y_i)$'s and $Var(Y_i)$'s are erroneous, and present the sufficient condition under which the algorithm satisfies $\{\nu_i, \rho_i\}, \forall M_i$.

Our previous work [33] established sensitivity of the timeliness assurances to variations in execution-time demand estimates—GCMUA also inherits this feature. Here, we develop an extension to that property to variations in memory demand estimates.

Let a mutator M_i's correct, expected memory demand be $E(X_i)$ and its correct variance be $Var(X_i)$, and let an erroneous expected demand $E''(X_i)$ and an erroneous variance $Var''(X_i)$ be specified as the input to GCMUA. Let M_i's timeliness requirement be $\{\nu_i, \rho_i\}$. We show that if GCMUA can satisfy $\{\nu_i, \rho_i\}$ with $E(X_i)$ and $Var(X_i)$, then there exists a sufficient condition under which the algorithm can still satisfy $\{\nu_i, \rho_i\}$ even with $E''(X_i)$ and $Var''(X_i)$.

Theorem 9. *Assume that GCMUA satisfies* $\{\nu_i, \rho_i\}, \forall M_i$, *under correct, expected memory demand estimates,* $E(X_i)$*'s, and their correct variances,* $Var(X_i)$*'s. When incorrect expected values,* $E''(X_i)$*'s, and variances,* $Var''(X_i)$*'s, are given as inputs instead of* $E(X_i)$*'s and* $Var(X_i)$*'s, GCMUA satisfies* $\{\nu_i, \rho_i\}, \forall M_i$, *if* $E''(X_i) + (A_i - E(X_i))\sqrt{\frac{Var''(X_i)}{Var(X_i)}} \geq A_i, \forall M_i$.

Proof. We assume that if GCMUA has correct $E(X_i)$'s and $Var(X_i)$'s as inputs, then it satisfies $\{\nu_i, \rho_i\}, \forall M_i$. This implies that the A_i's determined by (2) are feasibly scheduled by GCMUA, satisfying all task critical times:

$$\mu_i' = \frac{(A_i - E(X_i))^2}{Var(X_i) + (A_i - E(X_i))^2}. \quad (3)$$

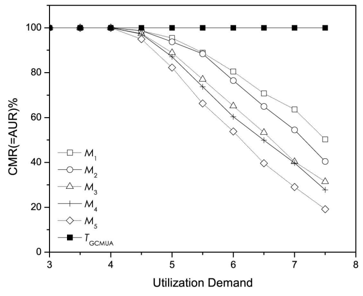
However, GCMUA has incorrect inputs, $E''(X_i)$'s and $Var''(X_i)$, and based on these, it determines A_i''s by (2) to obtain the probability $\rho_i', \forall M_i$:

$$\rho_i' = \frac{(A_i'' - E''(X_i))^2}{Var''(X_i) + (A_i'' - E''(X_i))^2}. \quad (4)$$

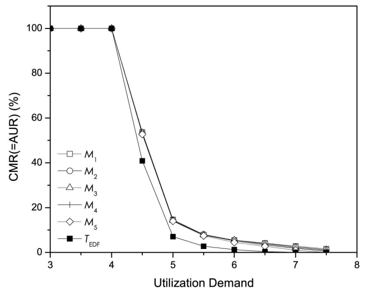
Unfortunately, A_i'' that is calculated from the erroneous $E''(X_i)$ and $Var''(X_i)$ leads GCMUA to another probability ρ_i'' by (2). Thus, although we expect the utility assurance with the probability ρ_i', we can only obtain assurance with the probability ρ_i'' because of the error. ρ'' is given by

$$\rho_i'' = \frac{(A_i'' - E(X_i))^2}{Var(X_i) + (A_i'' - E(X_i))^2}. \quad (5)$$

Note that we also assume that tasks with A_i'' satisfy global EDF's schedulable utilization bound; otherwise, GCMUA cannot provide the assurances. To satisfy

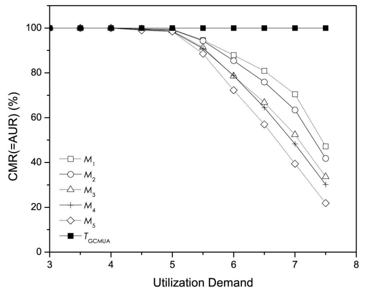


(a)

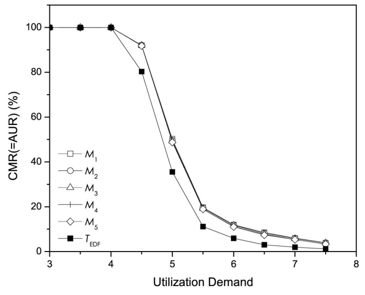


(b)

Fig. 2. Performance under constant demand (normally distributed), step TUFs. (a) GCMUA and (b) EDF.



(a)



(b)

Fig. 3. Performance under constant demand (exponentially distributed), step TUFs. (a) GCMUA and (b) EDF.

$\{\nu_i, \rho_i\}, \forall M_i$, the actual probability ρ''_i must be greater than the desired probability ρ'_i. Since $\rho''_i \geq \rho'_i\ (\geq \rho_i)$,

$$\frac{(A''_i - E(X_i))^2}{Var(X_i) + (A''_i - E(X_i))^2} \geq \frac{(A_i - E(X_i))^2}{Var(X_i) + (A_i - E(X_i))^2}.$$

Hence, $C'' \geq C_i$. From (3) and (4),

$$A''_i = E''(X_i) + (A_i - E(X_i))\sqrt{\frac{Var''(X_i)}{Var(X_i)}} \geq A_i. \qquad (6)$$

□

6 Experimental Results

We conducted simulation studies to validate our analytical results and to compare GCMUA's performance against global EDF-based GC scheduling, considering four processors. We considered two cases: 1) the mutator demands are constant and GCMUA exactly estimates the execution time allocation and 2) the mutator demands statistically vary and GCMUA probabilistically estimates the execution time allocation for each mutator. We considered two TUF shape patterns: 1) all mutators with step TUFs and 2) a heterogeneous TUF shape class, which included step, linear, and parabolic TUF shapes.

6.1 Performance with Constant Demand

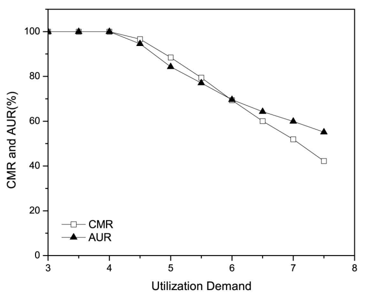
We considered a set of five mutator tasks with $\{\nu_i, \rho_i\} = \{1.0, 1.0\}, i = \{1, \ldots, 5\}$, and a GCMUA-scheduled GC task, denoted T_{GCMUA}. Mutator M_i's period P_i, expected execution time demand $E(Y_i)$, and expected memory demand $E(X_i)$ were randomly generated in the range of [10, 40], $[1, \alpha \cdot P_i]$, and [64, 1024], respectively, where $\alpha = \max\{\frac{C_i}{P_i} | i = 1, \ldots, 5\} = 0.4$ and $Var(Y_i) = Var(X_i) = 0$. We generated normally and exponentially distributed execution time demands. Mutator execution times were changed along with the total utilization demand (or *UD*).

According to [34], EDF's schedulable utilization bound depends on α as well as the number of processors. Thus, no matter how many processors the system has, there exists tasks with *UD* close to 1.0, which cannot be feasibly scheduled under EDF. The number of mutators depends on the given *UD*. In our experiments, the *UD* ranged from 3.0 to 7.5, including when it exceeded the number of processors.

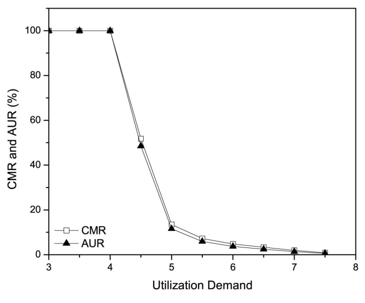
Figs. 2 and 3 show the critical-time meet ratio (or CMR) and the accrued utility ratio (or AUR) of each mutator under increasing *UD*, for step TUFs, under normally and exponentially distributed mutator execution times, respectively. CMR is the ratio of the number of jobs satisfying their critical times to the total number of job releases, and AUR is the ratio of the total accrued utility to the maximum possible total utility. For mutators with step TUFs, we show CMR and AUR in the same plot since they are identical. When all mutators have step TUFs and the *UD* satisfies global EDF's utilization bound (i.e., $UD <\approx$ 2.5 here, by the GFB bound [30]), GCMUA performs the same as EDF. This validates Theorem 5.

Figs. 2a and 3a show that all mutators obtain 100 percent of CMR and AUR when $UD \leq 4.0$ and T_{GCMUA} maintains 100 percent of CMR and AUR under all *UDs*. Thus, GCMUA ensures that memory is never exhausted during both underloads and overloads. When $UD > 4.0$, we observe that GCMUA-scheduled mutators gracefully degrade their timeliness as GCMUA schedules as many (feasible) high-PUD mutators as possible, in contrast to EDF's earliest-deadline mutator ordering (irrespective of utility).

From Figs. 2b and 3b, we observe that the CMR/AUR of EDF-scheduled mutators and the collector, denoted T_{EDF}, sharply drops beyond $UD = 4.0$, implying that the system

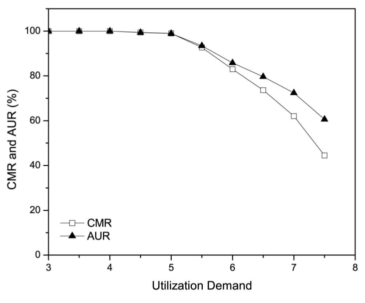


(a)

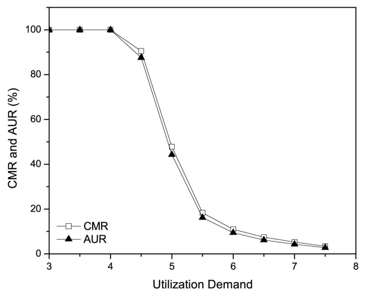


(b)

Fig. 4. System-level CMR/AUR under constant demand (normally distributed), step TUFs. (a) GCMUA and (b) EDF.



(a)



(b)

Fig. 5. System-level CMR/AUR under constant demand (exponentially distributed), step TUFs. (a) GCMUA and (b) EDF.

cannot satisfy the mutator memory demands. This is due to EDF's *domino effect* that occurs when *UD* exceeds the number of processors. Observe that EDF misses deadlines much earlier than when $UD = 4.0$, as indicated in [30]. Note that the trends in Figs. 2 and 3 are consistent, demonstrating GCMUA's robustness against different execution time distributions.

Figs. 4 and 5 show the system-level CMR and AUR for step TUFs, under normally and exponentially distributed execution times, respectively. Unlike Figs. 2 and 3, the system-level CMR and AUR are different, as satisfying the critical times of different mutators can result in different total accrued utility. In Figs. 4 and 5, we observe similar trends (at the system-level) as in Figs. 2 and 3: GCMUA gracefully degrades performance, while EDF suffers from the domino effect. The trends are consistent across different execution demand distributions.

6.2 Performance with Statistical Demand

We evaluated GCMUA's statistical timeliness assurances for heterogeneous TUF shapes. We assigned step TUF to M_1 and the collector, parabolic TUF to M_2 and M_5, and linearly decreasing TUF to M_3 and M_4. Table 1 shows the mutator settings. For each mutator M_i's demand Y_i, we generated normally and exponentially distributed execution time demands. Mutator execution times were varied with the *UD*.

Figs. 6 and 7 show the task-level AUR, task-level CMR, and system-level AUR and CMR under increasing *UD*, for heterogeneous TUFs, under normally and exponentially distributed mutator execution times, respectively. Figs. 6a and 7a show that all mutators under GCMUA accrue 100 percent AUR within global EDF's bound (i.e., $UD <\approx\ 2.5$ here), thus satisfying the desired $\{\nu_i, \rho_i\}$ described in Table 1. This validates Theorem 7.

We also observe that GCMUA achieves 100 percent AUR and CMR for M_1 under all *UDs*. This is because, M_1 has a step TUF with the highest maximum utility. Thus, GCMUA favors M_1 over others to obtain more utility when it cannot meet all mutator critical times.

As defined in Theorem 8, the system-level AUR under GCMUA can be calculated as $(0.98 \times 0.80 \times 92 + 0.95 \times 0.75 \times 63 + 0.94 \times 0.85 \times 75 + 0.97 \times 0.90 \times 69 + 0.96 \times 0.92 \times 54) \div 353 = 80.69$ percent. In Figs. 6c and 7c, we observe that the system-level AUR under GCMUA is more than 80.69 percent. This validates Theorem 8. We also observe that GCMUA's system-level AUR and CMR degrade gracefully as the algorithm differentially favors mutators according to their contribution to the total accrued utility.

Similar to the results under constant demand, we also observe that the trends in Figs. 6 and 7 are consistent, illustrating GCMUA's robustness.

7 Conclusions, Future Work

We consider garbage collection in dynamic multiprocessor real-time systems, and present a scheduling algorithm called GCMUA. We show that GCMUA probabilistically satisfies task utility lower bounds, and lower bounds

TABLE 1
Mutator Settings

	U_i^{max}	P_i	ρ_i	ν_i	μ_i
M_1	92	12	0.98	0.80	0.98
M_2	63	15	0.95	0.75	0.96
M_3	75	17	0.94	0.85	0.95
M_4	69	20	0.97	0.90	0.94
M_5	54	25	0.96	0.92	0.92

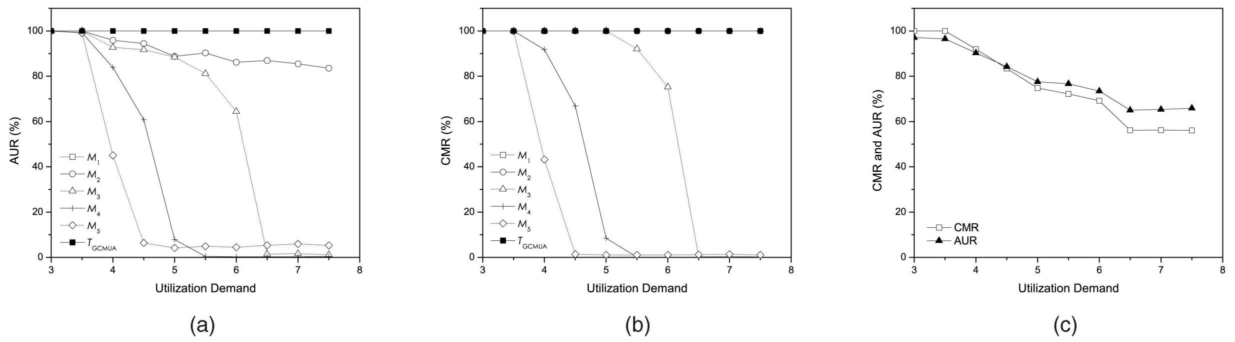


Fig. 6. Performance under statistical demand (normally distributed), heterogeneous TUFs. (a) Task-level AUR, (b) task-level CMR, and (c) system-level AUR and CMR.

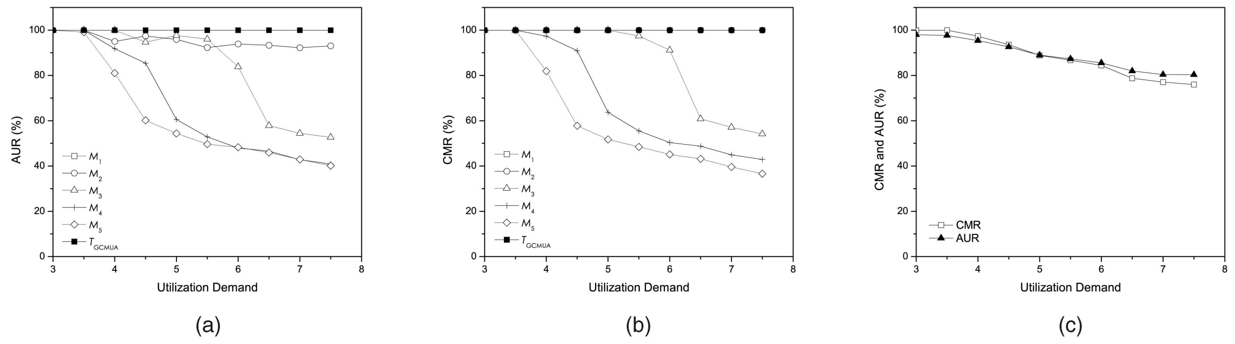


Fig. 7. Performance under statistical demand (exponentially distributed), heterogeneous TUFs. (a) Task-level AUR, (b) task-level CMR, and (c) system-level AUR and CMR.

system-wide total accrued utility. When task utility bounds cannot be met due to overloads, GCMUA maximizes total utility by completing a subset of tasks which yields high total utility, and thereby, gracefully degrades timeliness.

Future directions include allowing collectors with non-negligible atomic segments, allowing multiple GC threads for greater scalability (like [8]), relaxing the task arrival model (e.g., unimodal), and allowing mutators with synchronization dependencies. Implementation of the algorithm in an OS/virtual machine for obtaining further experimental insights is also important.

Acknowledgments

This work was sponsored by the US Office of Naval Research (ONR) (Grant N00014-00-1-0549) and by The MITRE Corporation (Grant 52917). Preliminary results appeared in [29].

References

[1] D. Detlefs, "A Hard Look at Hard Real-Time Garbage Collection," *Proc. IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing (ISORC '04)*, pp. 23-32, May 2004.

[2] R. Clark, E.D. Jensen, A. Kanevsky, and J. Maurer, "An Adaptive, Distributed Airborne Tracking System," *Proc. IEEE Int'l Workshop Parallel and Distributed Real-Time Systems (WPDRTS '99)*, pp. 353-362, Apr. 1999.

[3] R.K. Clark, E.D. Jensen, and N.F. Rouquette, "Software Organization to Facilitate Dynamic Processor Scheduling," *Proc. IEEE Int'l Workshop Parallel and Distributed Real-Time Systems (WPDRTS '04)*, p. 122b, Apr. 2004.

[4] W. Yuan and K. Nahrstedt, "Energy-Efficient CPU Scheduling for Multimedia Applications," *ACM Trans. Computer Systems*, vol. 24, no. 3, pp. 292-331, 2006.

[5] E.D. Jensen, C.D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Systems," *Proc. IEEE Real-Time Systems Symp. (RTSS '85)*, pp. 112-122, Dec. 1985.

[6] S.G. Robertz and R. Henriksson, "Time-Triggered Garbage Collection: Robust and Adaptive Real-Time GC Scheduling for Embedded Systems," *Proc. ACM Conf. Language, Compiler, and Tool for Embedded Systems (LCTES '03)*, pp. 93-102, 2003.

[7] R. Henriksson, "Scheduling Garbage Collection in Embedded Systems," PhD dissertation, Lund Inst. of Technology, July 1998.

[8] P. Cheng and G.E. Blelloch, "A Parallel, Real-Time Garbage Collector," *Proc. ACM Programming Language Design and Implementation (PLDI '01)*, pp. 125-136, 2001.

[9] R.H. Halstead, Jr., "MULTILISP: A Language for Concurrent Symbolic Computation," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 4, pp. 501-538, 1985.

[10] A.W. Appel, J.R. Ellis, and K. Li, "Real-Time Concurrent Collection on Stock Multiprocessors," *Proc. ACM Programming Language Design and Implementation (PLDI '88)*, pp. 11-20, 1988.

[11] M.P. Herlihy and J.E.B. Moss, "Lock-Free Garbage Collection for Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 3, pp. 304-311, May 1992.

[12] D. Doligez and G. Gonthier, "Portable, Unobtrusive Garbage Collection for Multiprocessor Systems," *Proc. ACM Principles of Programming Languages (POPL '94)*, pp. 70-83, 1994.

[13] S. Nettles and J. O'Toole, "Real-Time Replication Garbage Collection," *Proc. ACM Programming Language Design and Implementation (PLDI '93)*, pp. 217-226, 1993.

[14] G.E. Blelloch and P. Cheng, "On Bounding Time and Space for Multiprocessor Garbage Collection," *Proc. ACM Programming Language Design and Implementation (PLDI '99)*, pp. 104-117, 1999.

[15] D.F. Bacon, C.R. Attanasio, H.B. Lee, V.T. Rajan, and S. Smith, "Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector," *Proc. ACM Programming Language Design and Implementation (PLDI '01)*, pp. 92-103, 2001.

[16] H.G. Baker, "List Processing in Real Time on a Serial Computer," *Comm. ACM*, vol. 21, no. 4, pp. 280-294, 1978.

[17] T. Endo, "A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines," Master's thesis, Univ. of Tokyo, Feb. 1998.

[18] C. Flood, D. Detlefs, N. Shavit, and C. Zhang, "Parallel Garbage Collection for Shared Memory Multiprocessors," *Proc. USENIX Java Virtual Machine Research and Technology Symp. (JVM),* 2001.

[19] S. Feizabadi and G. Back, "Java Garbage Collection Scheduling in Utility Accrual Scheduling Environments," *Proc. Java Technologies for Real-time and Embedded Systems (JTRES '05),* Oct. 2005.

[20] H. Cho, C. Na, B. Ravindran, and E.D. Jensen, "On Scheduling Garbage Collector in Dynamic Real-Time Systems with Statistical Timing Assurances," *Proc. IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing (ISORC '06),* pp. 215-223, Apr. 2006.

[21] H.G. Baker, "The Treadmill: Real-Time Garbage Collection without Motion Sickness," *ACM SIGPLAN Notices,* vol. 27, no. 3, pp. 66-70, 1992.

[22] R.A. Brooks, "Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware," *Proc. ACM Symp. LISP and Functional Programming (LFP '84),* pp. 256-262, 1984.

[23] R.R. Fenichel and J.C. Yochelson, "A LISP Garbage-Collector for Virtual-Memory Computer Systems," *Comm. ACM,* vol. 12, no. 11, pp. 611-612, 1969.

[24] S.C. North and J.H. Reppy, "Concurrent Garbage Collection on Stock Hardware," *Proc. Conf. Functional Programming Languages and Computer Architecture,* pp. 113-133, 1987.

[25] T. Kim, N. Chang, N. Kim, and H. Shin, "Scheduling Garbage Collector for Embedded Real-Time Systems," *Proc. ACM Conf. Language, Compiler, and Tool for Embedded Systems (LCTES '99),* pp. 55-64, 1999.

[26] D.F. Bacon, P. Cheng, and V.T. Rajan, "A Real-Time Garbage Collector with Low Overhead and Consistent Utilization," *Proc. ACM Principles of Programming Languages (POPL '03),* pp. 285-298, 2003.

[27] D.P. Maynard et al. "An Example Real-Time Command, Control, and Battle Management Application for Alpha," CMU CS Dept., Technical Report 88121 (Archons Project), Dec. 1988.

[28] B. Brandenburg, J. Calandrino, and J. Anderson, "On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study," *Proc. IEEE Real-Time Systems Symp. (RTSS),* 2008.

[29] C. Na, H. Cho, B. Ravindran, and E.D. Jensen, "Garbage Collector Scheduling in Dynamic, Multiprocessor Real-Time Systems," *Proc. IEEE Int'l Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA '06),* pp. 101-105, 2006.

[30] M. Bertogna, M. Cirinei, and G. Lipari, "Improved Schedulability Analysis of EDF on Multiprocessor Platforms," *Proc. IEEE Euromicro Conf. Real-Time Systems (ECRTS '05),* pp. 209-218, July 2005.

[31] S.K. Dhall and C.L. Liu, "On a Real-Time Scheduling Problem," *Operations Research,* vol. 26, no. 1, pp. 127-140, 1978.

[32] 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, Oct. 2005.

[33] H. Cho, H. Wu, B. Ravindran, and E.D. Jensen, "On Multiprocessor Utility Accrual Real-Time Scheduling with Statistical Timing Assurances," *Proc. IFIP Int'l Conf. Embedded and Ubiquitous Computing (IFIP EUC '06),* pp. 274-286, Aug. 2006.

[34] J. Goossens, S. Funk, and S. Baruah, "Priority-Driven Scheduling of Periodic Tasks Systems on Multiprocessors," *Real-Time Systems,* vol. 25, nos. 2-3, pp. 187-205, 2003.

Hyeonjoong Cho received the BS degree in electronic engineering from Kyungpook National University in 1996, the MS degree in electronic and electrical engineering from the Pohang University of Science and Technology in 1998, and the PhD degree in computer engineering from Virginia Polytechnic Institute and State University in 2006. He is an assistant professor in the Department of Computer and Information Science at Korea University (KU). His research focuses on real-time software on various platforms including single/multiprocessors, sensor networks, real-time operating systems, embedded systems, and industrial field bus. Before he joined KU in 2009, he worked as a senior researcher at the Electronics and Telecommunications Research Institute, South Korea. He is a member of the IEEE.

Binoy Ravindran is an associate professor in the Electrical and Computer Engineering Department at Virginia Polytechnic Institute and State University. The central theme of his research is adaptive, real-time embedded software, i.e., real-time embedded software that can dynamically adapt to uncertainties in their operating environments, and satisfy time constraints acceptably well with acceptable predictability. This theme is reflected in different aspects of real-time embedded systems research that he conducts, which include real-time scheduling and resource management (single processors, multiprocessors, distributed systems), real-time operating systems, real-time networking, and real-time middleware. He and his PhD students have recently developed several new results in this area, some of which have also been transitioned to US Department of Defense (DoD) programs. He currently serves as an IEEE distinguished visitor, ACM distinguished speaker, and as an associate editor of *ACM Transactions on Embedded Computing Systems*. He is a senior member of the IEEE and the IEEE Computer Society.

Chewoo Na received the BS and MS degrees in computer science and engineering from Inha University, South Korea, in 1996 and 1998, respectively. He has been working toward the PhD degree in the Electrical and Computer Engineering Department at Virginia Tech since 2005. In 1998, he joined Interlink Systems Co., South Korea, and was a research engineer. In 2000, he was a senior researcher for Teleware Network Systems Co., South Korea. In 2002, he joined LG Industrial Systems Co., South Korea, as a senior research engineer. His current research interests include wireless sensor networks and wireless network optimization and resource allocation. He is a student member of the IEEE.