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

Chewoo Na*, Hyeonjoong Cho*, Binoy Ravindran*, and E. Douglas Jensen‡

*ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{cwrha,hjcho,binoy}@vt.edu

‡The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

Abstract

We consider garbage collection in dynamic, multiprocessor real-time systems, and 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 and superiority.

I. 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 typically the case for many hard real-time subsystems. However, it is inflexible for applications whose memory demands cannot be statically estimated and consequently desire run-time 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.

Real-time GC works can be 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 "mutator") by performing a bounded amount of GC work at each mutator allocation request. Examples include Baker's GC [2], and its numerous derivatives [3]–[10].

Though the work-based approach reduces the individual pause times, timing assurances are generally difficult to obtain with this approach, 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 semi-space copying collector [2] 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. Further, mutator reads occurring immediately after a flip are likely to trigger the read-barrier, resulting in the worst-case cost; reads at other times will cost much less, as objects will be in the to-space. Nevertheless, the worst-case GC cost must be considered for each allocation. 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 scheduler just as another application 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 mutator time constraints, while not exhausting memory? Example works in this paradigm include [11]–[16].

A. Multiprocessor Embedded Systems

Multiprocessor architectures (e.g., Symmetric Multi-Processors or SMPs, Single Chip Heterogeneous Multi-processors or SCHMs) are becoming more attractive for embedded real-time systems primarily because major processor manufacturers (Intel, AMD) are making them decreasingly expensive. This makes such architectures very desirable for embedded real-time applications with high computational workloads, where additional cost-effective processing capacity is often needed. Responding to this trend, RTOS vendors are increasingly providing multiprocessor platform support (e.g., QNX Neutrino [17]).

Garbage collection on multiprocessors has a long history [3], [4], [7], [10], [18]–[24], that started with Halstead's modification of Baker's algorithm [3]. Halstead partitions the shared-memory into distinct regions for each processor, and allocation and garbage-collection proceeds within each region through a semi-space copying scheme. As Cheng and Blelloch shows in [10], this approach can imbalance the GC work distribution across processors. In [25],

Endo balances Halstead-collector's work through a mark-and-sweep, but non-incremental collector. Flood *et. al.* extend Endo's work for a copying collector [26]. Herlihy and Moss also improve Halstead-collector's performance through lock-free synchronization [20]. Doligez and Gonthier's concurrent mark-and-sweep GC [22] is a derivative of Dijkstra *et. al.*'s collector [27], with no read overheads. Bacon *et. al.* present a concurrent reference counting collector in [24].

These non real-time GC efforts became the basis for real-time GC (on multiprocessors), which started with Blelloch and Cheng's collector [23]. 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. Fixed bounds on pause times and memory usage are established in [23]. Cheng and Blelloch parallelizes this collector in [10], by running multiple GC threads in parallel, so that collection can keep up with increased allocation rate, when mutator threads and processors increase. Further, they analyze real-time performance in terms of the percentage of processor time available for mutator threads during any time interval.

B. Our 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 run-time 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% probability, meeting 95% of deadlines) are appropriate for these systems, and possible, when activities exhibit non-deterministic, but stochastic behaviors.

Another distinguishing feature of these systems is that their time constraints include those with *non-deadline* semantics such as "earlier the better, but before a certain time" and "best before a certain time, after which earlier is the better." Further, an activity's urgency is often 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. Yet another important distinguishing feature of these systems, is their relatively long activity execution time magnitudes, compared to those of conventional real-time subsystems—e.g., in the order of milliseconds to minutes. Some example such systems that motivate our work include [28], [29].

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 [30] that allows the specification of a broad range of time constraints, including deadlines and (aforementioned) non-deadline time constraints, and full decoupling of activity urgency from activity importance; and (2) a *stochastic mutator model*, where mutator execution time and memory demand are stochastically expressed, to account for uncertainties in execution and memory allocation behaviors.

We consider garbage collection in this mutator model, and on an SMP system with M identical processors. We consider time-based GC, due to its previously mentioned advantages. Similar to [14], we fully decouple mutator and the GC, allowing for any incremental GC algorithm with fine granularity (e.g., [11]). Unlike [10], 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 assurances on system-level timeliness behavior; and (3) maximize the total activity attained utility.

This problem has not been studied in the past and is $\mathcal{NP}$-hard. We present a polynomial-time, global 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 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. Our simulation-based experiments validate our analytical results and confirm GCMUA's effectiveness and superiority.

None of the past real-time GC efforts, except [15], [16], consider our mutator and system model, which include TUF time constraints, stochastic execution-time and memory demands, resource overloads, and SMPs. Both [15], [16] consider some aspects of our model (e.g., TUFs, overloads), but are restricted to one processor. Moreover, [15] does not provide any assurances on mutator timing behavior such as lower bounds on individual and collective activity utility, and [16] is restricted to deterministic memory demand. In contrast, our work precisely provides such assurances, and allows stochastic memory demand (on SMPs).

Thus, the contribution of the paper is the GCMUA scheduling algorithm that provides assurances on (individual and collective) mutator timing behavior 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: Section II describes our models and scheduling objective. In Section III, we discuss the rationale and design of GCMUA. We establish the algorithm's properties in Section IV and report our simulation studies in Section V. The paper concludes in Section VI.

II. Models and Objective

A. Mutator and GC Model

We consider the application to consist of a set of mutator tasks, denoted $\mathbf{M}=\{M_1, M_2, ..., M_n\}$. Each mutator task M_i has a number of instances, called jobs. Jobs may be released either periodically or sporadically with a

known minimal inter-arrival time. The j^{th} job of mutator M_i is denoted as $J_{i,j}$. A mutator M_i's period or minimal inter-arrival time is denoted as P_i.

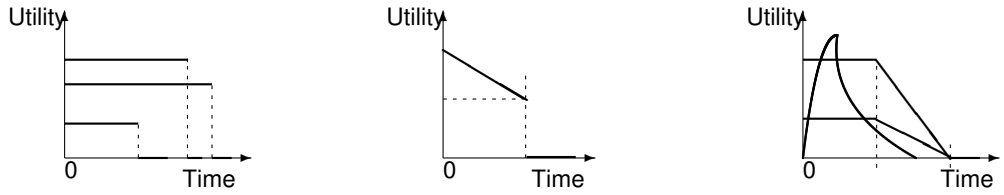
All mutator tasks are assumed to be independent—i.e., they do not share any non-CPU resources or have any precedences. The basic scheduling entity that we consider is the job abstraction. Thus, we use J to denote a mutator job without being task specific, as seen by the scheduler at any scheduling event.

Similar to [23], we consider a memory model where the entire shared-memory is divided into two spaces (from and two) and shared among all processors. Similar to [14], we consider time-based GC, where the GC is run periodically, allowing for any fine-grained incremental GC algorithm—e.g., [11]. Later, in Section III-C, we show how the GC's period is determined.

B. Timeliness Model

We consider the TUF model [30] for specifying a mutator task's time constraint. A task's TUF specifies the utility of completing the task as a function of that task's completion time. The classical deadline is a TUF's special case — a binary-valued, downward "step" shaped TUF, implying that a positive utility is accrued for completing the task 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.

As previously mentioned, our motivating applications also have tasks with *non-deadline* time constraint semantics, such as those where the utility attained for task 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 (see [29], [31] for application details).



(a) (b) (c)

Fig. 1: Example TUF Time Constraints: (a) Step TUFs; (b) AWACS TUF [29]; and (c) Coastal Air defense TUFs [31]

Jobs of the same mutator task have the same TUF. Mutator M_i's TUF is denoted as $U_i()$. Thus, job $J_{i,j}$'s completion at time t will yield an utility of $U_{i,j}(t)$. 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 $X_{i,j}$, which are the earliest and the latest times for which the TUF is defined, respectively. We assume that $I_{i,j}$ is the arrival time of job $J_{i,j}$, and $X_{i,j} - I_{i,j}$ is the period

or minimal inter-arrival time P_i of the 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.

C. Execution and Memory Demands

We estimate the statistical properties—e.g., distribution, mean, variance, of job execution-time and memory allocation demands rather than the worst-case demands because: (1) our motivating applications [28], [29] 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 [32], resulting in resource over-supply; and (3) allocating execution-times and memory based on the statistical estimation of the corresponding task demands can provide statistical timing assurances, which are sufficient for our motivating applications.

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. A number of measurement-based, off-line and online profiling mechanisms exist (e.g., [33]). We assume that the means and variances of Y_i and X_i are finite and determined through such profiling.

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 variance on the demands as $Var(Y_i)$ and $Var(X_i)$, respectively.

D. Statistical Timeliness Requirement

We consider a statistical timeliness requirement for *each* mutator task. 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% of its maximum utility with a probability no less than 93%. For step TUFs, ν can only be 0 or 1. Note that the hard real-time objective of always meeting 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.

E. Scheduling Objective

We consider a two-fold scheduling criterion: (1) assure that each 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 for each M_i (e.g., due to CPU or memory overloads), our objective is to maximize the total task attained 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 [34].

III. The GCMUA Algorithm

A. Bounding Accrued Utility

Let $s_{i,j}$ be the sojourn time of the j^{th} job of mutator task M_i. Mutator 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 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.

B. Estimating Execution Time and Memory Demands

Since execution time demands are stochastically specified (through means and variances), 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 of mutator's demand must account for the uncertainty in both time and memory demand specifications (i.e., the variance factors). 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 $C_i \geq E(Y_i)$, we have:

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

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 $\varepsilon_i = \frac{(C_i - E(Y_i))^2}{Var(Y_i) + (C_i - E(Y_i))^2}$ and obtain the minimum required execution time $C_i = E(Y_i) + \sqrt{\frac{\varepsilon_i \times Var(Y_i)}{1 - \varepsilon_i}}$.

GCMUA 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 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:

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

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 $\prod_i \varepsilon_i \times \prod_i \mu_i \geq \max_i\{\rho_i\}$, 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 $\prod_i \varepsilon_i$, since each probability is exclusive. Likewise, the least probability satisfying all memory demands is given by $\prod_i \mu_i$. Let $\prod_i \varepsilon_i \times \prod_i \mu_i = \Phi_i$. Φ_i produces the least probability satisfying both execution-time and memory demands. Thus, M_i accrues at least ν_i percentage of the utility with the probability ρ_i, when Φ_i is greater than or equal to the maximum ρ_i. ■

C. Bounding GC Cycle Time

In [14], Robertz and Henriksson develop an upper bound on the GC cycle time that guarantees that the application never runs out of memory:

Lemma 2: For a mutator M_i that arrives with a period P_i (or frequency $f_i = 1/P_i$) and 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 total 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, they bound the memory that can be safely allocated during a cycle:

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

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

Theorem 4: An upper bound on the GC cycle time that guarantees no memory exhaustion is:

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

In [14], the T_{GC} upper bound is then used as the GC's deadline and period, since satisfying that deadline for the GC ensures no memory exhaustion.

We compute T_{GC} using Theorem 4, but use the stochastic demand A_i determined from Equation 2, instead of the worst-case demand a_i assumed in [14], since we consider statistical properties of memory demands instead of worst-case. Doing so, and satisfying the resulting T_{GC} deadline for the collector at run-time ensures that the actual mutator memory demand is satisfied at run-time, 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 run-time exceeds A_i, causing "memory overloads," an appropriate T_{GC} would not be found. Hence, in order make the GC task schedulable, we need to dynamically calculate T_{GC} using the actual run-time demand and check for GC's feasibility. If infeasible, some mutator tasks will have to be aborted to ensure feasibility. We explain this process in detail in the next section.

D. Algorithm Description

GCMUA's scheduling events include job arrival, job completion, the expiration of a time constraint including the arrival of a TUF's termination time, and memory allocation request. To describe GCMUA, we define the following variables and auxiliary functions:

- ζ_r: current job set in the system including running jobs and unscheduled jobs.
- σ_{tmp}, σ_a: a temporary schedule; σ_m: schedule for processor m, where $m \leq N$.
- $J_k.X$: job J_k's termination time; $J_k.C(t)$: J_k's remaining allocated execution time.
- `offlineComputing()` is computed at time $t = 0$ once. For a mutator M_i, it computes C_i as $C_i = E(Y_i) + \sqrt{\frac{\varepsilon_i \times Var(Y_i)}{1-\varepsilon_i}}$ and T_{GC} computed with memory demand upper bound A_i.
- `UpdateRAET(`ζ_r`)` updates the remaining allocated execution time of all jobs in the set ζ_r.
- `ComputeTgc(t)` calculates the GC cycle time with estimated memory demand A_i and actual memory demand a_i except the memory demand of J_t.
- `feasible(`σ`)` returns a boolean value denoting schedule σ's feasibility; `feasible(`J_k`)` denotes job J_k's feasibility. For σ (or J_k) to be feasible, the predicted completion time of each job in σ (or J_k), must not exceed its termination time. For the GC task, we assume that its worst-case execution time, denoted C_{GC} is known. C_{GC} depends upon the particular GC algorithm (see [12] for C_{GC} calculation, for a variant of Brooks' algorithm [6]). Note that $T_{GC} < C_{GC}$ due to memory overloads.
- `findProcessor()` returns the ID of the processor on which the currently assigned tasks have the shortest sum of allocated execution times.
- `append(`J_k`,`σ`)` appends J_k at rear of schedule σ.
- `removeLeastPUDJob(`σ`)` removes job with the least *potential utility density* (or PUD) from schedule σ. PUD is the ratio of the expected job utility (obtained when job is immediately executed to completion) to the remaining job allocated execution time, i.e., PUD of a job J_k is $\frac{U_k(t+J_k.C(t))}{J_k.C(t)}$. Thus, PUD measures the job's "return on investment." Function returns the removed job. In case of the job equals to Note that when $T_{GC} < C_{GC}$, the GC task is not removed. Instead, the next least PUD job is removed.
- `headOf(`σ_m`)` returns the set of jobs that are at the head of schedule σ_m, $1 \leq m \leq N$.

Algorithm 1: GCMUA

```
1  Input  : M={M_1,...,M_n}, ζ_r={J_1,...,J_N}, N:# of processors
2  Output : array of dispatched jobs to processor p, Job_p
3  Data: {σ_1,...,σ_N}, σ_tmp, σ_a
4  offlineComputing(M);
5  Initialization: {σ_1,...,σ_N} = {0,...,0};
6  UpdateRAET(ζ_r);
7  for ∀J_k ∈ ζ_r do
8      J_k.PUD = U_k(t+J_k.C(t)) / J_k.C(t);
9  σ_tmp = sortByEXF( ζ_r );
10 T_GC = computeTgc(null);
11 while T_GC < 0 and !IsEmpty( σ_tmp ) do
12     t = removeLeastPUDJob( σ_tmp );
13     T_GC = computeTgc( t );
14 for ∀J_k ∈ σ_tmp from head to tail do
15     if J_k.PUD > 0 or J_k == J_c then
16         m = findProcessor();
17         append(J_k, σ_m);
18 for m = 1 to N do
19     σ_a = null;
20     while !feasible( σ_m) and !IsEmpty( σ_m ) do
21         t = removeleastPUD( σ_m );
22         append( t, σ_a );
23     sortByEXF( σ_a );
24     σ_m += σ_a;
25 {Job_1,...,Job_N} = headOf( {σ_1,...,σ_N} );
26 return {Job_1,...,Job_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. 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 9. In each step of the *while-loop* from line 10 to line 13, 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 $T_{GC} > 0$. In each step of *for-loop* from line 14 to line 17, 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 (procedure `findProcessor()`). The rationale for this choice is that the shortest summed execution time processor results in the nearest scheduling event for completing a job after assigning each job. This will establish the same schedule as that of the global EDF. Then, the job J_k with the earliest termination time is inserted into the local schedule σ_m of the selected processor m.

In the *for-loop* from line 18 to line 24, GCMUA attempts to make each local schedule feasible by removing the lowest PUD job. In line 20, if σ_m is not feasible, then GCMUA removes the job with the least PUD from σ_m until σ_m becomes feasible. All removed jobs are temporarily stored in a schedule σ_a and then appended to each σ_m in EXF order. Note that simply aborting the removed jobs may result in decreased accrued utility. This is because, the algorithm may decide to remove a job which is estimated to have a longer allocated execution time than its actual one, even though it may be able to accrue some utility. For this case, GCMUA gives the job another chance to be scheduled instead of aborting it, which eventually makes the algorithm more robust. Finally, each job at the head of $\sigma_m, 1 \leq m \leq N$ is selected for execution on the respective processor.

IV. Algorithm Properties

A. Timeliness Assurances

We establish GCMUA's timeliness assurances under the conditions of (1) independent tasks that arrive periodically and sporadically, and (2) task utilization demand satisfies any of the schedulable utilization bounds for global EDF (GFB, BAK, BCL) in [35].

Theorem 5 (Optimal Performance with Step Shaped TUFs): Suppose that only step shaped 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.

We prove this by examining Algorithm 1. In line 9, the queue σ_{tmp} is sorted in a non-decreasing termination time order. In line 12, the function `findProcessor()` returns the index of the processor on which the summed execution time of assigned tasks is the shortest among all processors. Assume that there are n tasks in the current ready queue. We consider two cases: (1) $n \leq N$ and (2) $n > MN$. When $n \leq N$, the result is trivial: GCMUA's schedule of tasks on each processor is identical to that produced by EDF (every processor has a single mutator or none assigned). When $n > N$, task T_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 T_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 the deadline in EDF scheduling. Under conditions (1) and (2), EDF meets all deadlines. Thus, each processor always has a feasible schedule, and the while-loop from line 16 to line 18 will never be executed. Thus, GCMUA produces the same schedule as global EDF.

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

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 $\{\nu_i, \rho_i\}$ for each mutator 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 Equation 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 $\{\nu_i, \rho_i\}, \forall i$ at least with a probability of $\varepsilon_i = max\{\rho_i\}$, since all mutators must have lesser actual execution time than the allocated. $\prod \varepsilon_i = max\{\rho_i\} \geq \rho_i, \forall i$—i.e., the algorithm accrues ν_i utility with a probability of at least ρ_i. ■

Theorem 8 (System-Level Utility Assurance): Under conditions (1) and (2), if a mutator M_i's TUF has the highest height U_i^{max}, then the system-level utility ratio, defined as the ratio of the total utility accrued by GCMUA to the system's 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: We denote the number of jobs released by mutator M_i as m_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 total accrued utility to the maximum possible total utility is $\frac{max\{\rho_i\}\nu_1 U_1^{max} l_1 + ... + max\{\rho_i\}\nu_n U_n^{max} l_n}{U_1^{max} l_1 + ... + U_n^{max} l_n}$. This is greater than or equal to $\frac{\rho_1 \nu_1 U_1^{max} l_1 + ... + \rho_n \nu_n U_n^{max} l_n}{U_1^{max} l_1 + ... + U_n^{max} l_n}$. Thus, when l_i approaches ∞, the formula converges to $\frac{\sum_{i=1}^{n} \rho_i \nu_i U_i^{max}}{\sum_{i=1}^{n} U_i^{max}}$. ■

B. Sensitivity of Assurances

GCMUA is designed in the same stochastic framework as two of our past algorithms [16], [36]. Consequently, the algorithm inherits the sensitivity property on assurances that the 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 [36] establishes sensitivity of the assurances to variations in execution-time demand – GCMUA also inherits this feature. Here, we develop an extension to that property to variations in memory demand. Let a mutator M_i's correct expected memory demand 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 the mutator's statistical timeliness requirement be $\{\nu_i, \rho_i\}$. We show that if GCMUA can satisfy $\{\nu_i, \rho_i\}$ with the correct expectation $E(X_i)$ and the correct variance $Var(X_i)$, then there exists a sufficient condition under which the algorithm can still satisfy $\{\nu_i, \rho_i\}$ even with the incorrect expectation $E''(X_i)$ and incorrect variance $Var''(X_i)$.

Theorem 9 (): Assume that GCUA satisfies $\{\nu_i, \rho_i\}, \forall i$, 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, GCUA satisfies $\{\nu_i, \rho_i\}, \forall i$, if $E''(X_i) + (A_i - E(X_i))\sqrt{\frac{Var''(X_i)}{Var(X_i)}} \geq A_i, \forall i$.

Proof: We skip the proof for brevity, but it is similar to that for execution time demands (see [36]). ■

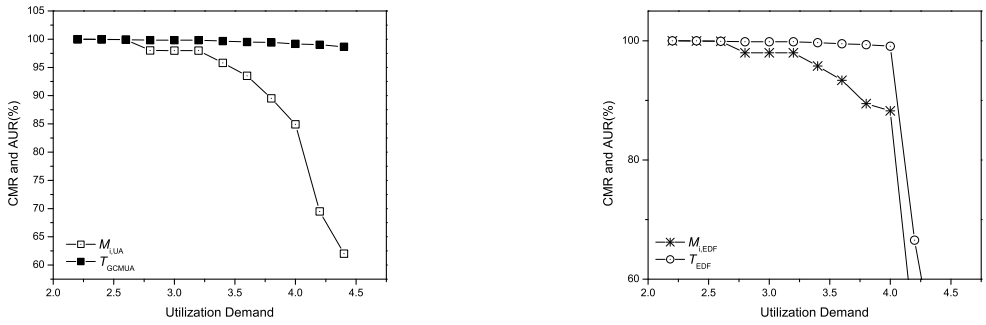
V. Simulation-Based Experiments

We conducted simulation-based experimental studies to validate our analytical results and to compare GCMUA's performance with Robertz and Henriksson's EDF-based GC scheduling [14]. We considered two cases: (1) the execution time demand of all mutator tasks are constant (i.e., no variance) and GCMUA exactly estimates the execution time allocation, and (2) the demand of all tasks statistically varies and GCMUA probabilistically estimates the execution time allocation for each task. The former experiment is conducted to evaluate GCMUA'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) all tasks have step-shaped TUFs, and (2) a heterogeneous TUF shape class, including step, linearly decreasing, and parabolic shapes.

A. Performance with Constant Demand

According to [37], EDF's schedulable utilization bound (or EUB) depends on α as well as the number of processors. It implies that there exists task sets with utilization demand(or UD) close to 1.0 no matter how many processors the system has. We consider an SMP machine with 4 processors and mutator tasks with period P_i, expected execution time demand $E(Y_i)$ and memory demand $E(X_i)$, $[1, \alpha \cdot P_i]$, respectively, where α is given $max\{\frac{C_i}{P_i}|i = 1,..,5\}$ and $Var(Y_i) = Var(X_i) = 0$.



(a) GCMUA

(b) Robertz and Henriksson's EDF GC Scheduling

Fig. 2: Performance Under Constant Demand, Step TUFs

Figure 2 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 step 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.

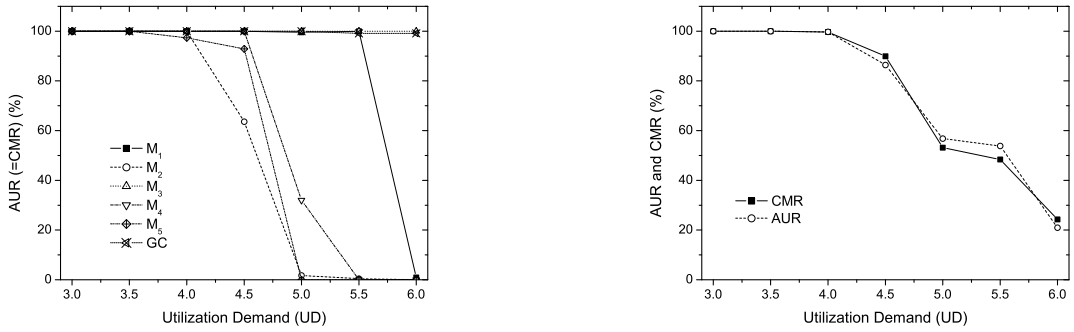
We vary the UD from 2.2 to 4.6. For tasks with step TUFs, we show AUR and CMR in the same plot since they are identical, as satisfying the critical time implies the accrual of a constant utility.

B. Performance with Statistical Demand

We now evaluate GCMUA's statistical timeliness assurance. Table I summarizes the mutator settings used in our study. The table shows the mutator periods and the maximum utilities (or U_{max}) of the TUFs. Note that the period of the GC is dynamically calculated. For each mutator's demand, normally distributed execution time demands are generated. Average execution times are changed along with the total UD.

TABLE I: Mutator Settings

Mutators and GC	P_i	U_i^{max}	ρ_i	$Var(Y_i)$	μ_i	$Var(X_i)$
M_1	23	347.4	0.96	1.6	0.96	1.0
M_2	22	372.6	0.96	1.6	0.96	1.0
M_3	50	188.3	0.96	1.6	0.96	1.0
M_4	40	143.2	0.96	1.6	0.96	1.0
M_5	45	20.0	0.96	1.6	0.96	1.0
GC	N/A	243.9	0.96	1.6	0.96	1.0



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

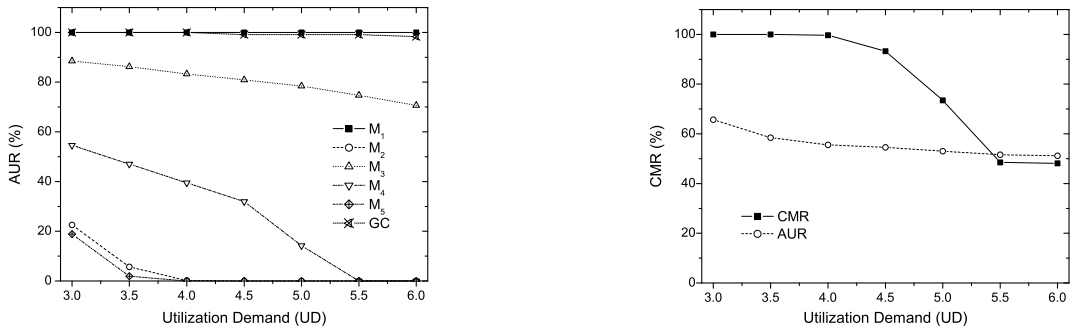
Fig. 3: Performance Under Statistical Demand, Step TUFs

Figures 3 shows AUR and CMR of each mutator under increasing total UD of GCMUA. For a mutator with step TUFs, task-level AUR and CMR are identical, as satisfying the critical time implies the accrual of a constant utility. Whereas, the system-level AUR and CMR are different as satisfying the critical time of each mutator does not always yield the same amount of utility.

Figure 3 shows that all mutators under GCMUA accrue 100% AUR and CMR within global EDF's bound (i.e., UD$<\approx$2.0 here), thus satisfying the desired $\{\nu_i, \rho_i\} = \{1, 0.96\}, \forall i$. This validates Theorem 7.

Under the condition beyond what Theorem 7 indicates, GCMUA achieves graceful performance degradation in both AUR and CMR in Figure 3, as the previous experiment in Section V-A implies.

According to Theorem 8, the system-level AUR must be at least 96%. (For each mutator $M_i, \nu_i = 1$, because all TUFs are step shaped.) We observe that AUR and CMR of GCMUA under the condition of Theorem 8 are above 99.0%. This validates Theorem 8.



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

Fig. 4: Performance Under Statistical Demand, Heterogeneous TUFs

We observe a similar trend in Figure 4 for heterogeneous TUFs. We assign step TUFs for M_1 and GC, linearly decreasing TUFs for M_3 and M_4, and parabolic TUFs for M_2 and M_5. For each mutator M_i, ν_i is set as $\{1.0, 0.2, 0.2, 0.2, 0.2\}$ and for GC, it is set as 1.0.

According to Theorem 8, the system-level AUR must be at least $0.96 \times (1.0 \cdot 347.4 + 0.2 \cdot 372.6 + 0.2 \cdot 188.3 + 0.2 \cdot 143.2 + 0.2 \cdot 20.0 + 1.0 \cdot 243.9)/(347.4 + 372.6 + 188.3 + 143.2 + 20.0 + 243.9) = 53.72\%$. In Figure 4, we observe that the system-level AUR under GCMUA is above 53.72% within global EDF's bound. This further validates Theorem 8 for non step-shaped TUFs.

VI. Conclusions, Future Work

In this paper, we consider garbage collection in dynamic, multiprocessor real-time systems, and present a scheduling algorithm called GCMUA. The algorithm considers mutator tasks that are subject to TUF time constraints, variable execution-time and memory demands, and resource overloads, and allows any fine-grained incremental GC algorithm. We establish that GCMUA probabilistically satisfies task utility lower bounds, and lower bounds system-wide total accrued utility. The algorithm's lower bound satisfactions have bounded sensitivity to variations in execution-time and memory demand estimates. When task utility lower bounds cannot be satisfied due to CPU and/or memory overloads, GCMUA 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.

Our work can be extended in several ways. To allow GC algorithms with larger granularity, GCMUA can be extended to allow GCs with non-negligible atomic segments. The work can also be extended in the direction of [10] by allowing multiple GC threads so that collection can scale with increased allocation rate. Other directions include relaxing our task arrival model (e.g., unimodal arrival) and allowing mutator threads to share non-CPU resources.

References

[1] D. Detlefs, "A hard look at hard real-time garbage collection," in *IEEE ISORC*, May 2004, pp. 23– 32.

[2] J. Henry G. Baker, "List processing in real time on a serial computer," *CACM*, vol. 21, no. 4, pp. 280–294, 1978.

[3] J. Robert H. Halstead, "Multilisp: a language for concurrent symbolic computation," *ACM TOPLAS*, vol. 7, no. 4, pp. 501–538, 1985.

[4] A. W. Appel, J. R. Ellis, and K. Li, "Real-time concurrent collection on stock multiprocessors," in *ACM PLDI*, 1988, pp. 11–20.

[5] H. G. Baker, "The treadmill: real-time garbage collection without motion sickness," *ACM SIGPLAN Not.*, vol. 27, no. 3, pp. 66–70, 1992.

[6] R. A. Brooks, "Trading data space for reduced time and code space in real-time garbage collection on stock hardware," in *ACM LFP*, 1984, pp. 256–262.

[7] S. Nettles and J. O'Toole, "Real-time replication garbage collection," in *ACM PLDI*. ACM Press, 1993, pp. 217–226.

[8] T. Yuasa, "Real-time garbage collection on general-purpose machines," *J. Syst. Softw.*, vol. 11, no. 3, pp. 181–198, 1990.

[9] M. S. Johnstone, "Non-compacting memory allocation and real-time garbage collection," Ph.D. dissertation, University of Texas at Austin, 1997.

[10] P. Cheng and G. E. Blelloch, "A parallel, real-time garbage collector," in *ACM PLDI*, 2001, pp. 125–136.

[11] R. Henriksson, "Scheduling garbage collection in embedded systems," Ph.D. dissertation, Lund Institute of Technology, July 1998.

[12] T. Kim, N. Chang, N. Kim, and H. Shin, "Scheduling garbage collector for embedded real-time systems," *ACM SIGPLAN Not.*, vol. 34, no. 7, pp. 55–64, 1999.

[13] D. F. Bacon, P. Cheng, and V. T. Rajan, "A real-time garbage collector with low overhead and consistent utilization," in *ACM POPL*, 2003, pp. 285–298.

[14] S. G. Robertz and R. Henriksson, "Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems," in *ACM LCTES*, 2003, pp. 93–102.

[15] S. Feizabadi and G. Back, "Java garbage collection scheduling in utility accrual scheduling environments," in *JTRES*, October 2005.

[16] H. Cho, C. Na, B. Ravindran, and E. D. Jensen, "On scheduling garbage collector in dynamic real-time systems with statistical timing assurances," in *IEEE ISORC*, 2006, to appear. Available: http://www.ee.vt.edu/~realtime/isorc06.pdf.

[17] QNX, "Symmetric multiprocessing," http://www.qnx.com/products/rtos/smp.html, last accessed October 2005.

[18] J. Crammond, "A garbage collection algorithm for shared memory parallel processors," *Int. J. Parallel Program.*, vol. 17, no. 6, pp. 497–522, 1989.

[19] I. Foster, "A multicomputer garbage collector for a single-assignment language," *Int. J. Parallel Program.*, vol. 18, no. 3, pp. 181–203, 1990.

[20] M. P. Herlihy and J. E. B. Moss, "Lock-free garbage collection for multiprocessors," *IEEE TPDS*, vol. 3, no. 3, pp. 304–311, 1992.

[21] L. Huelsbergen and J. R. Larus, "A concurrent copying garbage collector for languages that distinguish (im)mutable data," in *ACM PPOPP*, 1993, pp. 73–82.

[22] D. Doligez and G. Gonthier, "Portable, unobtrusive garbage collection for multiprocessor systems," in *ACM POPL*, 1994, pp. 70–83.

[23] G. E. Blelloch and P. Cheng, "On bounding time and space for multiprocessor garbage collection," in *ACM PLDI*, 1999, pp. 104–117.

[24] 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," in *ACM PLDI*, 2001, pp. 92–103.

[25] T. Endo, "A scalable mark-sweep garbage collector on large-scale shared-memory machines," Master's thesis, University of Tokyo, February 1998.

[26] C. Flood, D. Detlefs, N. Shavit, and C. Zhang, "Parallel garbage collection for shared memory multiprocessors," in *USENIX JVM Conference*, April 2001.

[27] E. W. Dijkstra, L. Lamport, *et al.*, "On-the-fly garbage collection: an exercise in cooperation," in *Language Hierarchies and Interfaces, International Summer School*. London, UK: Springer-Verlag, 1976, pp. 43–56.

[28] R. K. Clark, E. D. Jensen, and N. F. Rouquette, "Software organization to facilitate dynamic processor scheduling," in *IEEE WPDRTS*, April 2004.

[29] R. K. Clark, E. D. Jensen, *et al.*, "An adaptive, distributed airborne tracking system," in *IEEE WPDRTS*, April 1999.

[30] E. D. Jensen, C. D. Locke, and H. Tokuda, "A time-driven scheduling model for real-time systems," in *IEEE RTSS*, December 1985, pp. 112–122.

[31] D. P. Maynard, S. E. Shipman, *et al.*, "An example real-time command, control, and battle management application for alpha," CMU CS Dept., Tech. Rep., Dec. 1988, archons Project TR 88121.

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

[33] X. Zhang, Z. Wang, *et al.*, "System support for automated profiling and optimization," in *ACM SOSP*, October 1997, pp. 15–26.

[34] R. K. Clark, "Scheduling dependent real-time activities," Ph.D. dissertation, Carnegie Mellon University, 1990.

[35] M. Bertogna, M. Cirinei, and G. Lipari, "Improved schedulability analysis of edf on multiprocessor platforms," in *IEEE ECRTS*, 2005, pp. 209– 218.

[36] H. Cho, H. Wu, B. Ravindran, and E. D. Jensen, "On multiprocessor utility accrual real-time scheduling with statistical timing assurances," 2005, http://www.ee.vt.edu/~realtime/rtas06_RTL_VT.pdf.

[37] J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic tasks systems on multiprocessors," *Real-Time Systems*, vol. 25, no. 2-3, pp. 187–205, 2003.